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

Nils Nilsson

March 11, 1976

A

QLISP

Reference Manual

By

B. Michael Wilber

Artificial Intelligence Center

Technical Note 118

The work reported herein was sponsored by the Advanced Research Projects Agency under Contract DAHC04-72-C-0008 and by the National Aeronautics and Space Administration under Contract NASW-2086.

ABSTRACT

A mature version of QLISP is described. QLISP permits free intermingling of advanced language constructs with those of INTERLISP. It provides an associative data base, viewed from perspectives controlled by a powerful context mechanism. Access to the data base can be selectively aided by teams of programs from which members are chosen by a pattern matching operation. Tentative computation is facilitated by a backtracking facility which can be triggered by directed (or undirected) failure. It also provides for partial evaluation and includes special features to expedite handling of transitive relations and equivalence relations. All these language features are embedded in a programming system which is very smoothly integrated into the programming environment provided by INTERLISP.

PREFACE

QLISP is many things: a collection of concepts, a programming language embodying the concepts, and a programming system providing a friendly environment in which to use the programming language. This document will describe QLISP several times in succession, once from each of these perspectives. The first major section will describe the concepts as they are peculiar to QLISP; it will describe only as much of the programming language as is necessary to fix ideas. Then there is a description of QLISP as a programming language; it assumes that you know the concepts involved and merely attempts to describe the language in a clear and concise way. The last major section tells how the programming environment of INTERLISP has been extended and modified so that it is friendly and sympatnetic to QLISP.

To greatly expedite matters, I assume you are familiar with the basic concepts of list processing and with LISP as a family of list processing languages and systems. QLISP happens to be implemented in INTERLISP, and some references to the host language of QLISP must be particular to INTERLISP as a particular dialect of LISP. For instance, INTERLISP treats strings and numbers in ways not necessarily shared by other LISP dialects, and any such description of QLISP must specifically allude therein to INTERLISP. Moreover, INTERLISP provides a unique programming environment which lends great power and flexibility to QLISP; many implementational details of QLISP stem directly from particular aspects of that environment. For these reasons, INTERLISP is specifically mentioned quite often in this manual; if you are unfamiliar with INTERLISP as a specific LISP, you will miss a rich contact with the details of QLISP. This is especially true of Section IV, which describes QLISP as a system and so necessarily depends quite strongly on INTERLISP being the specific host language. Otherwise though, you will not cut yourself off from the broad sweep of QLISP by reading LISP for INTERLISP.

I have made a habit of using pronouns in the first and second persons because this is not a tome left by an impersonal "the author" for some impersonal "the user"; down at the bottom of it, there are only two people involved: you and me.

ACKNOWLEDGMENTS

This work was largely supported by the Advanced Research Projects Agency through Contract DAHC04-72-C-0008 and also by the National Aeronautics and Space Administration through Contract NASW-2086. The QA4 project was lead by Jeff Rulifson and included Jan Derksen and Richard Waldinger. The early QLISP work was done by Earl Sacerdoti and Rene Reboh, and Daniel Sagalowicz and I did the later work. Malcolm Newey wrote the unification program which performs all pattern matching in QLISP. Warren Teitelman gave generously of his time and energy, providing us with support, insight and encouragement concerning the host language. Bert Raphael, as head of SRI Artificial Intelligence Center, gave fundamental encouragement to the QLISP development beyond simply providing a properly conducive atmosphere. Bernie Elspas, Rich Fikes and Georgia Sutherland, as well as Earl and Richard and many others helped debug the system and provided direction for its evolution by being eager and willing to use it in spite of its shortcomings. Writing the manual has been my task; Earl encouraged me, Bert has been patient with me, and they and Daniel have read and thoughtfully criticized previous drafts of this manual.

OUTLINE

I. Introduction

QLISP is a programming language and a programming system; it extends the INTERLISP programming environment in ways that are especially useful for the applications of artificial intelligence. It includes a data base that can store composite data items, such as sets, which are awkward to handle in INTERLISP. The data base is organized to simulate associative retrieval, to permit property lists to be attached to the composite items, and, by means of a powerful context mechanism, to coherently manage separate views into the data base. It also includes a unification program to decompose complex data items, to aid in their associative storage and retrieval, and on occasion to select applicable functions from a large team. Finally, it adds a third mode of interpretation to the two occurring in LISP, where a given form can be interpreted literally (quoted) or fully evaluated; in QLISP it can also be partially evaluated ("instantiated").

The central construct of the QLISP language is what we call a "statement", which is conceptually similar to a LISP form but differs from it in several respects. The parts of a statement are usually instantiated rather than being evaluated. A statement usually has one principal operand and several optional operands which significantly modify the way the statement handles its principal operand. The optional operands are delimited by key words and can come in any order. The parsing method is also significant from considerations of efficiency: it is parsed the first time it is encountered, and the result is saved as a LISP function call which is subsequently used directly.

QLISP is an outgrowth of QA4, a programming language developed for artificial intelligence at SRI around 1970. In fact, it was originally an alternative implementation of the QA4 concepts which sacrificed some generality to gain a factor of 30 in efficiency. From there, it has proceeded in evolutionary steps for several years. The concepts have changed under the pressure of experience and the changing needs of the Artificial Intelligence Center, but they are still fundamentally those of QA4.

II. Concepts

A. Data Objects

QLISP operates on all data types provided by INTERLISP. Moreover, it provides additional data types, together with the means for constructing and analyzing them. Section III.A enumerates the constructors and tells how to use them; this section merely discusses the concepts involved.

QLISP extends the INTERLISP notion of property lists. INTERLISP provides property lists for literal atoms, and the INTERLISP property list is global to all references. QLISP provides its own version of property lists for all of its own data types and for most of those of INTERLISP. It also provides a powerful context mechanism by which the user can control access to its property lists. The following sections on instantiation, the context mechanism and the statements which store and retrieve properties tell how all this can be used.

The inset paragraphs outline the special treatment QLISP gives various of the data types.

> A literal atom (in the INTERLISP sense) whose first character is either _ or $ is a QLISP variable. The next section expands on their special role and treatment.
>
> If numbers and strings are equal (in the INTERLISP sense), QLISP reckons them as identical. They can have QLISP property lists.
>
> A tuple is equivalent to an INTERLISP list, but it can also have a QLISP property list. A tuple of the form (f a1 a2... an) will normally be stored internally as (TUPLE f a1 a2... an). See section III.E.4 on DEFTYPE for exceptions.
>
> A vector is similar to a tuple. It also contains an ordered sequence of elements. However, it is closer to the ordinary mathematical conception of n-tuples in its evaluation rule. A tuple EVALs like the corresponding INTERLISP list: its first element is applied as a function to the values of the succeeding elements as arguments. However, the result of EVALing a vector is simply a list of the values of the elements. A vector containing the elements e1, e2,... en is stored internally as (VECTOR e1 e2... en).
>
> A bag implements the conventional mathematical

concept of a multiset: it is an unordered collection of elements that may be duplicated. For example, (BAG A A B) is equivalent to (BAG A B A) and distinct from (BAG A B).

A class is an unordered collection of elements without duplication (in mathematical terms, a set). For example, (CLASS A A B) is equivalent to (CLASS A B) or (CLASS B A).

B. Values, Instantiation, and Evaluation

QLISP statements typically instantiate their parameters (rather than evaluating them, as do LISP functions). That is, variables are replaced by their values, but functions are not applied to their arguments as happens in LISP evaluation. Any symbols which are not QLISP variables remain unchanged. The distinction between QLISP variables and other symbols is made by giving variables a $ prefix, so that, for example, $X is a variable with the $ prefix, while X is just the symbol X. Then, if $X had the value B, instantiating (CONS X $X) would give (CONS X B). Evaluating a QLISP $ variable produces its value, just the same as instantiating it. If $Y had the value D, then evaluating (CONS $Y $X) would give (D . B), while instantiating it would simply produce (CONS D B) without applying the function to its arguments.

This of course raises the question of how QLISP variables obtain their values. I'll answer it in the next section but one: it's normally done by the pattern matching operation. However, I can say here that a variable signifies its willingness to accept a value by sporting the _ prefix (e.g., _X). The distinction between the prefixes _ and $ is even stronger than that: when an expression containing an _ variable is instantiated, the _ variable does not produce a value but instead remains unchanged. The gravity of that fact is just this: preparatory to matching one expression against another, one or both of them is typically instantiated, and you should understand that any _ variables will survive that process unchanged. Finally, there is the question of evaluation: when an _ variable is evaluated, it still remains unchanged. There are special modes, operators and other prefixes to modify this simple picture, but it's correct as far as it goes and will suffice for the moment.

We should note in passing that QLISP variables have definite scopes, implemented via the context mechanism (discussed in sec. II.F). That is, their values are stored on their property lists and can differ from one computational context to another. Thus, a given variable can independently take different values in different contexts and, when instantiated, produce the value visible

in the appropriate context. While the motivation was to avoid naming conflicts between different parts of a (possibly large) system of programs, the upcoming discussion of contexts implicitly shows that this locality can be used to advantage in conjunction with contexts generated explicitly by context designators as well as those generated implicitly by the structure of the program.

C. Backtracking and Failure

Often, a program acting on a complex situation (or reasoning about one) comes to a pass where several alternative actions (or lines of reasoning) present themselves with no clear criteria to choose among them. It is sometimes useful in such a case to arbitrarily pick one of them and tentatively proceed onward; if further analysis shows the choice to be clearly wrong, we would like the program to forget those tentative actions, return to the choice point and continue with a different alternative. That is essence of "backtracking"; QLISP contains a slightly modified concept.

There are other applications for this notion of tentative execution with backtracking. We generalize this notion of a choice point to what we call a "backtrack point"; the description (in later sections) of the QLISP programming language will point out other circumstances under which a backtrack point is established. At any epoch in the execution of a program, there will typically be a many backtrack points to which control can return. They are hierarchically related in this sense: suppose a backtrack point to be established and the computation to be at work on one of the alternatives; further backtrack points can be established "inside" this one, but when control returns to the outer backtrack point, all the subordinate ones will vanish.

The mechanism by which a computation returns to a backtrack point is something we call "failure" just because that kind of return is usually invoked when a computation discovers that its current choice fails to be appropriate. A failure usually takes the computation back to the nearest ("innermost") backtrack point but can be directed to return to any currently open backtrack point. A backtrack point previously established by an ambiguous choice also generates a failure when all its alternatives fail. Thus, we also have a notion of failures propagating back from one backtrack point to another, with the propagation stopping at the nearest (i.e., innermost) backtrack point which still has some alternatives to try. Propagating failure is different from directed failure: directed failure passes over (and thus closes off) backtrack points until it finds its target regardless of whether they still have untested alternatives. Only then does the failure start propagating

(possibly further) back until it finds a backtrack point with further alternatives to try out. Directed failure, then, is intended for situations in which a computation can assuredly say that a wrong choice was taken at a level (possibly) much higher than the backtrack points which surround it very closely.

There is a final notion implicit in the idea of backtracking - and this is the one from which it derives its name. When a computation fails back to a backtrack point, it does more than just returning control there to try another possible choice. The side effects of the execution over which it is backtracking will also vanish. This isn't always desirable, though: sometimes the computation will discover something worth remembering in spite of possible backtracking over the branch that discovered it. There is provision for just such a case: the context mechanism (discussed in the section after next) provides a way to record data in a manner immune to erasure by backtracking.

D. Pattern Matching

The QLISP pattern matcher is one focal point of the concepts outlined so far. It understands and can disassemble tuples, vectors, bags and classes. It is the principal means by which QLISP variables attain the values they produce on instantiation. It can generate a QLISP failure if it is unable to perform an assigned matching. It also uses backtracking in two slightly different ways: often it will decompose an assigned matching into several subordinate matchings and try a different decomposition if one of the subordinate attempts fails; or it can deliver a successful match to the user's program for deeper analysis and try to generate an alternate matching if the user's program fails.

The pattern matcher is used under various circumstances QLISP. Retrieval from the QLISP data base is directed by the pattern matcher. Functions can be defined with a QLAMBDA construct, which uses a pattern to be matched with the arguments supplied. Details are forthcoming in Part IV, which describes QLISP as a programming language; this section merely outlines the concepts involved.

The essence of the pattern matcher is this: given two data, it determines how (and whether) they match one another; if they can't be completely unified, the pattern matcher fails (in the QLISP sense). If both data contain only constants, then the pattern matcher simply determines whether they are equivalent. The story changes considerably if one or both of the data happens to contain QLISP variables with the _ prefix: the pattern matcher will substitute anything at all for such a variable if that substitution will make the two data unify. If the match

succeeds, any _ variable will be given its required substitution as a value. (Keep clear in your mind, however, that the substitution takes place only to the extent that _ variables are assigned values; the data themselves are not altered in the unification process.)

Consider some examples:

_X will unify with anything.

FOO will unify only with (the INTERLISP literal atom) FOO (or, of course, with an _ variable).

(BAG A) and (CLASS A) won't unify because they are of different types.

(BAG A B) and (BAG A C) won't unify because their elements are different.

(BAG A B) and (BAG A _X), on the other hand, will unify. That unification would assign the value B to _X, so that (BAG A $X) would then instantiate to (BAG A B). (Of course, (BAG $X A) will then also instantiate to (BAG A B) because bags are indifferent to the order of their elements.)

(BAG A _X) and (BAG B _Y) will unify, assigning the values A to _Y and B to _X, respectively.

(CLASS _X _Y) and (CLASS 1) will unify, assigning the value 1 to both _X and _Y because class elements can be repeated without significance.

(BAG _X _Y) and (BAG 1) won't unify because they contain different numbers of elements.

(BAG _X _X) and (BAG A B) won't unify.

(BAG _X _X) and (BAG A A), however, will. The first bag will unify with precisely those bags whose two elements are the same.

E. Teams of Functions

Teams of functions give QLISP a great deal of power. Many QLISP statements take a team of QLAMBDA functions which may be applied to the statement's primary operand. The statement will attempt to match a team member's QLAMBDA pattern with the operand; if the match succeeds, then the function's body will be supplied the bindings resulting from the match. In any given QLISP statement, the team can be any QLISP expression, so that the statement need only embody enough information to permit calculation of the set of team members. Thus, a program using QLISP teams can be extremely modular and flexible.

Teams have many uses: when a datum is being inserted in the data base, they can check its consistency with data already present, or they can also insert some simple consequences; or they can be used to deduce a fact not explicitly available. In fact, some programs do the main

work of their deductive inference by using team members: any team member will do a highly specialized analysis and then invoke some team, which often is the same as its own team.

For example, a QLISP statement whose goal is to prove some theorem could first check the data base to ascertain whether the theorem has already been proven. If not, the statement could invoke functions from a team of deduction functions in an attempt to prove it. Each of the functions could be highly specialized to prove a theorem of only a very special format; each would specify the format in its QLAMBDA pattern, and only the correct ones would be entered. If the theorem can indeed be proved, then it could be asserted with a QLISP statement specifying a different team of functions to enter additional subsidiary facts into the data base.

F. World Models and Contexts

The environment in which a program operates will typically include a model of some world. For example, a program modelling a process will maintain a model of that process's state, and a sentence understander will model its understanding of a sentence's content. It is often necessary to separately keep track of the various states of the world model or even to separately model alternative worlds for planning or hypothetical reasoning. Facilities to accomplish this are provided by the QLISP context mechanism.

A central notion is that of a "context"; it is profitable to think of a context as a viewpoint on the entire QLISP data base. (The QLISP data base stores QLISP property lists and the values of QLISP variables.) That is, any given expression can effectively have different property lists when viewed from different viewpoints, and QLISP variables can similarly have different values. Some understanding of how this is accomplished can be gleaned from a more detailed consideration of the interrelation of these viewpoints. There is a distinguished viewpoint representing the basal state of the data base; otherwise, any viewpoint is a refinement of some other (previously extant) viewpoint, with the refinement process ultimately leading back to the distinguished view on the world. These viewpoints (contexts), then, form a tree with this distinguished viewpoint as its root node.

With the idea of the context tree clearly in mind, one can easily understand how the context mechanism works. Whenever some property is assigned to some expression in some context, the assignment is recorded in the (unique) node of the context tree corresponding to the given context. Whenever some property of some expression must be retrieved with respect to some context, then a simple search takes

place. Starting with the given (corresponding) node, the context tree is searched back toward the root node until some record is found of an assignment to the given property of the given expression; if none is found, then the desired property is considered to have the value NOSUCHPROPERTY. Of course, some property of an expression can be deleted in some context by assigning it the value NOSUCHPROPERTY in that context; one should understand that such an act doesn't disturb that property's value in any context not derived from the given context. This kind of deletion works because the NOSUCHPROPERTY masks any other occurrences of its indicator in any context closer to the root node.

To give you an explicit handle on the context mechanism, QLISP provides contexts as a distinct data type, instances of which are tokens of the divers viewpoints on the data base. New contexts are generated implicitly by the structure of a program or explicitly by a QLISP construct called the "context designator". The context designator can produce the ancestor or a new descendant of any given context.

A context datum is slightly more complicated than thus far implied. Recall that when QLISP backtracks over a fruitless (or erroneous) branch of a computation, its side effects disappear. This isn't always desirable: a computation may produce valuable partial results but ultimately fail. To selectively permit persistence of side effects through failures, a context datum can also contain a flag to specify that changes made under its influence should be permanent.

C. Special Relations

QLISP incorporates extensions to efficiently handle the special cases of equivalence relations and transitive relations. In both these cases, information is clearly present which hasn't been directly asserted. For example, if A>B and B>C are known, then a practicing mathematician would think nothing of deducing A>C; similarly, A=B and B=C lead quite directly to A=C. Such deductions are readily available to a QLISP program. Not only are there special mechanisms to handle transitive relations and equivalence relations, but there are further special mechanisms to handle the important special case of strict ordering. For example, once you know A<B, then you can easily deduce that A=B is false.

This special case is the primary way QLISP handles special relations (i.e., transitive relations and equivalence relations). That is, you can declare and use sets of six connected ordering and equivalence relations (analogous to =, <, and > and their denials). Special relations require special bookkeeping (which is handled

automatically) when you insert statements into the data base. This extra information is used to make the deductive retrieval of data which can be deduced from the data base even though they're not stored directly.

You can get special relations in their full generality when you declare a set of six connected relations: you needn't use all six of them. For example, you get perfectly general irreflexive transitive relations if you only use the analogues of < and > and ignore the other four. By the same token, you don't even need to declare a full set of six relations; QLISP needs all six for its internal bookkeeping, but it'll make up names for any you don't supply.

III. The QLISP Language

A. Constructors

A QLISP program can generate tuples, bags, vectors, and classes by using the corresponding constructors supplied in the language. That is, a form such as (VECTOR A B C) will, upon instantiation, produce the corresponding datum, (VECTOR A B C). More precisely, a datum of one of the four types TUPLE, VECTOR, BAG, or CLASS can be produced by instantiating a form whose first element is one of those four names. The succeeding elements are all instantiated, and their instantiations are the elements of the datum being constructed.

These constructors have a few properties beyond those just mentioned. For starters, they are indifferent to the distinction between instantiation and evaluation in the sense that evaluating one of them will produce the same result as instantiating it. Also, bags and classes do not preserve the ordering of their elements, and so they will not necessarily come back with their elements in the order specified by the program. Moreover, classes are insensitive to the number of times (greater than zero) that any given element appears: a class constructed by QLISP might contain each element only once.

There is yet another twist: fragments. As the elements are instantiated to generate elements of the (possibly new) datum, some of the instantiations will be taken as producing several elements instead of just one. A variable is flagged to produce one element by giving it the $ prefix; it produces several elements (i.e., it is a fragment) when it has the $$ prefix. The distinction is crucial: if $X instantiates to (VECTOR B C), then (VECTOR A $X) will instantiate to (VECTOR A (VECTOR B C)), but (VECTOR A $$X) will instantiate to (VECTOR A B C). Please note that fragments make sense only in the context of constructors: a composite datum can contribute its elements to another composite, but LISP has no notion of several data items apart from the notion of their composite.

The constructors are quite prepared to handle fragments of types other than their own; after instantiating a fragment, they use its elements and ignore its type. In the above example, if $X were instead to instantiate to (CLASS B C), then (VECTOR A $X) would of course give (VECTOR A (CLASS B C)), but (VECTOR A $$X) would still yield (VECTOR A B C). There is a definite limit to this flexibility; a fragment should always be a TUPLE, VECTOR, BAG, or CLASS.

The $$ prefix is only defined for variables, but you

can make a fragment from any expression with the ! operator. It behaves like a unary constructor: the form (! $X) would instantiate (or evaluate) just the same as would $$X. Thus, the form (VECTOR (! (VECTOR A B)) (! (VECTOR B C))) will instantiate to the vector (VECTOR A B B C).

In summary, then, one of the forms
(TUPLE a1 a2... an),
(VECTOR a1 a2... an),
(CLASS a1 a2... an), or
(BAG a1 a2... an)
will build a tuple, vector, class, or bag, respectively, when it is instantiated or evaluated. The elements will be the instantiations of all the ai, with bags and classes possibly failing to preserve their order and with classes representing each element only once. Fragment variables are marked by the $$ prefix, and general fragment expressions are built with the ! operator; fragments contribute their elements but not their types.

B. Pattern Matching

The QLISP pattern matcher takes the task of unifying one datum with another. It is also the usual mechanism by which QLISP variables are assigned their values. In fact, those two activities are closely allied: usually, one of the data supplied the pattern matcher will contain _ variables somewhere within it; the implied unification will be possible (if at all) only if the _ variables take certain values dictated by what the other datum contains at the corresponding positions. When a unification is possible, the pattern matcher shows its result by assigning the _ variables those required values.

The pattern matcher can handle fragments in a manner closely resembling that of the constructors. That is, if a tuple (or vector, bag, or class) contains a variable with the __ prefix, then the __ variable will match a fragment of that tuple (or vector, bag, or class); i.e., it will match several elements instead of just one.

The unification of fragments does not always give a unique answer. For example, (BAG _X __Y) will unify with (BAG A B C) in several ways: _X could be matched with A, B, or C; and __Y would then match a bag containing whichever two elements were left over. Programs usually must reject some of the possible matches; Section II.C shows how backtracking and failure can be used to reject the undesired possibilities. Also, (BAG A __X) would unify with (BAG B __Y), but a new fragment must be generated to represent the common part of the fragments: $X and $Y would receive values like (BAG B __QLISP:GENVAR:17) and (BAG A __QLISP:GENVAR:17), respectively.

C. Context Operators

This section discusses two of the mechanisms by means of which a program can explicitly use the context mechanism: the context designator and the context datum it produces. Later sections treat QLAMBDA, QPROG, QDECLARE, and WRT clauses; the first two generate new contexts, and the other two provide the principal means of explicitly using them. Contexts have two uses: access to the values of variables (instantiation and the three MATCH statements); and access to the world model (via the property list statements). All of these are treated in later sections.

The context designator instantiates to a context datum. It takes an argument specifying some (already extant) context and an optional additional argument directing it to return the ancestor or a new descendant of that context. It can also be directed to return a non-backtrackable context, which can be used in a branch of a computation to ensure that its noteworthy discoveries persist even if the whole branch fails.

The form of a context designator is exemplified by (CONTEXT PUSH LOCAL). Its first element is (the INTERLISP literal atom) CONTEXT, and it instantiates to a context datum. It resembles the constructors (e.g., VECTOR) in two respects: it evaluates and instantiates identically; and it instantiates the succeeding elements, of which there are one or two. If there is only one additional element (e.g., (CONTEXT LOCAL)), then the context designator produces either the local context or the global context; it may flag the context as non-backtrackable. The additional element can instantiate to one of these "context names", each corresponding to one of the four cases:

LOCAL -- the current (local) context;
GLOBAL -- the top (global) context;
ETERNAL -- a non-backtrackable version of LOCAL;
or
UNIVERSAL -- a non-backtrackable version of GLOBAL.

On the other hand, if it has two additional elements, the context designator will produce a neighbor of some context already in the context tree. The first additional element tells it which neighbor to produce, and the second tells it where to start. The second (i.e., the starting point) can instantiate to

one of the four "context names" just mentioned, or
a context datum resulting from some previous use of a context designator.

The first (i.e., the direction) can instantiate to either

POP, which produces its ancestor, or
PUSH, which produces a new descendant.

A few examples will illustrate how these aspects of the

context designator can be used in conjunction with one another.

(CONTEXT GLOBAL) is the global context.
(CONTEXT PUSH GLOBAL) is a brand new descendant of the global context.
(CONTEXT POP LOCAL) is the context immediately surrounding the local context.
(CONTEXT PUSH $C) is a new descendant of the context which is the value of the QLISP variable $C.
(CONTEXT PUSH (CONTEXT POP $C)) is a new sibling of the context designated by $C.

D. Instantiation, Evaluation and Instantiation Operators

Instantiation takes place in the local context. That is, when the value of a variable must be retrieved, the current context is examined for a binding of the variable. If a binding is found, the value in it is used; otherwise the search proceeds to succeeding ancestors until a binding is found. If the search arrives at the global context and examines it without finding a binding, then an error is generated.

Whether an expression is instantiated or evaluated depends on the structure of the program surrounding it. For example, LISP functions usually evaluate their arguments, while the BAG constructor instantiates its arguments before building its bag. However, this is only the default, and you can override it with the unary operators ' and @, which let you explicitly specify the mode in which their operand will be handled. That is, the expression

(' exp) instantiates exp, while
(@ exp) evaluates exp,

regardless of the default dictated by the environment. These operators would let you use

(PRINT (' (EQUALS (PLUS 3 1) (@ (PLUS 3 1)))))

to print the expression

(EQUALS (PLUS 3 1) 4).

The @ and ' operators facilitate the construction of QLISP expressions containing QLISP variables. It can sometimes be useful to construct such an expression and later give values to its variables and then instantiate it. If this expression is the value of a QLISP variable, though, instantiating the variable will only produce the expression, and the ' operator merely replaces evaluation by instantiation. For repeated instantiation ("superinstantiation"), a new operator operator is needed. This new operator, the " operator simulates the effect of repeatedly instantiating its operand as many times as possible:

(" exp) superinstantiates exp:
constants and QLISP variables having no

value remain as they are;
any QLISP variable, regardless of its prefix, which has a value is replaced by that value;
the operators @ and ' force evaluation and ordinary instantiation, respectively; and finally,
if a new expression was produced, the whole expression is again superinstantiated.

Understand that fragments (e.g., variables with the prefix $$ or __) are correctly expanded at each iteration of superinstantiation.

E. Statements

1. Statement Parts and Statement Processing

The QLISP statements take various forms and are processed in various ways, according to their "statement types". However, since the forms (and processing thereof) have some overlap and even a little unity, some general remarks are in order before we proceed to the more specific discussions. Each QLISP statement takes the form of a list; evaluating one of them makes it perform its action. The list is headed by a literal atom specifying its statement type (e.g., ASSERT),in a manner resembling a LISP form. Next in the statement may be one or two QLISP data expressions; their meaning and number is fixed by the statement type. Finally, the tail of the statement may contain some optional additional information in a freer form.

Parsing of the freeform part of a statement is guided by keywords. Some keywords take exactly one parameter; the others take an indefinite number, delimited by the next keyword in the statement (or its end). Some statement types also admit the inclusion of property list indicators (and possible associated properties too). Parsing them adds a slight complication: it is essential that indicators and property expressions be different from the keywords. Indicators are recognized as such by their presence at the beginning of the freeform part of a statement or immediately after the parameter of a one-parameter keyword. The indicator's property follows it immediately, and then the situation is the same: the next item can be a keyword or an indicator (or the statement may end). Another complication is that some statement types can use an indicator with no associated property expression; this is shown either by placing an indicator at the very end of the statement or by immediately following it with a keyword.

Parsing is usually followed by a little more

preprocessing before control is turned over to the execution routine for the statement type. Often, the entire contents of the statement other than the statement type and the keywords will be instantiated; but on other occasions, parts of the statement might instead be evaluated or simply turned over to the execution routine for more highly specialized processing. The choice between the alternatives varies from one statement type to another but is always the same for any given type. The only appropriate generalization is that instantiation is the most usual case; the individual discussions of the different statement types in the next few sections will note the exceptions.

For example, one of the statement types is QGET; the result of evaluating a QGET statement with a QLISP datum and an unpaired indicator is the corresponding property stored on the datum's QLISP property list. Evaluation of the form (QGET $X COLOR) will instantiate $X to some datum and return its color as the value of the statement.

There are five different keywords; this is how they behave.

WRT (one parameter). Many of the statement types can be told to take their effect in some context other than the context that is current when the statement is evaluated. The parameter of a WRT clause in such a statement specifies the other context to be used; it can instantiate to a context datum or to one of the words LOCAL, GLOBAL, ETERNAL, or UNIVERSAL.

APPLY (one parameter). The APPLY parameter instantiates to the name of a QLAMBDA function or to a tuple, vector, bag, or class of such names; those functions are collectively called the statement's APPLY team. Various statements will, under various circumstances, execute the team members in turn. If a team member fails, then the backtracking mechanism will make its side effects disappear before the next team member is tried; the process continues until a team member returns without failing (or the team is exhausted). APPLY teams are usually used with the statements which store and retrieve QLISP properties, and the explanations of those statements will further explain how APPLY teams work.

NAME (one parameter). A FAIL statement (described below) in a member of an APPLY team can explicitly name the statement to which the failure should be directed. If a NAME clause is included in a statement, the instantiation of

its parameter is used as that statement's name. If a NAME clause is omitted, then the name defaults to the statement's type, e.g., ASSERT.

THEN (several parameters). Some statement types take a THEN-part; the parameters are forms which are passed (uninstantiated and unevaluated) to the corresponding execution routine. Under some circumstances, the execution routine will serially evaluate them. Details vary from one statement type to another; they will be specified in the descriptions of the applicable types.

ELSE (several parameters). Similarly, some statements admit an ELSE-part; the parameters are forms that are passed untouched to the execution routine for possible later execution; the ATTEMPT statement uses this option.

This is the stylized manner in which I describe the individual statement types. Each statement type will be described in one paragraph or several; heading each description will be a schematic representation of the statement with all the options shown. The schematic representations will be of the form

(type exp1... inds/props ind k1 k2... kn).

The actual statement type will be included where the word "type" is shown here. The expi (the positional and usually required components) will usually be shown by something like "iexpi" or "eexpi" to suggest that the i-th positional expression will be instantiated or evaluated as part of the preprocessing for the statement. If "inds/props" or "inds" is included, then statements of that type will handle pairs of indicators and properties or an unpaired indicator, respectively. Finally, any ki included will be among the words WRT-clause, APPLY-team, NAME-part, THEN-part, and ELSE-part and indicate that the such a statement can handle the corresponding keyword (and its parameters). Any optional statement parts will be enclosed in square brackets [for emphasis.]

2. Pattern Matching Statements

These statements provide for explicit invocation of the pattern matching operation. Aside from representing one of the principal ways that QLISP variables will normally be assigned values, they are also useful for testing possible matches before deciding which way a given computation should proceed.

(MATCH eexp1 eexp2 [WRT-clause])
(MATCHQ iexp1 eexp2 [WRT-clause])
(MATCHQQ iexp1 iexp2 [WRT-clause])

The only difference among these statement types is the choice between instantiation and evaluation for the positional expressions. The resulting data are unified, and any new values are assigned in the specified context (if present, or else in the current context). That is, for each _ variable in the positional expressions, the given (or current) context is searched for a binding, and the search proceeds to succeedingly distant ancestors until a binding is found. (If no binding is found, a new one is inserted into the global context.) Finally, the new value is inserted into that binding.

3. Statements that Store and Retrieve Properties

The heartwood of QLISP resides in the statements about to be described. They are the only means by which the QLISP property lists are accessed, aside from access to the values of variables. They are also the principal users of the APPLY teams and the NAME clauses.

Many of these explanations will allege that one statement type can be explained in terms of another. You should not take that too literally. The paraphrases are only intended as compact representations of the explanations of the statements they mention; those statements do not appear on the control stack and cannot be the referent of a FAIL statement, although the allegations are otherwise accurate.

This is the place to explain the missing detail from the above discussion of APPLY teams: the argument to which the team members are applied. Understand that the principal business of each of these statements is to store some datum in the QLISP data base or to retrieve a datum or several data from it. Given that, it should be easy to understand, for each datum entering or leaving the data base, the team members are successively applied to it. There will be further discussion of this later.

The last detail is the context under which these statements access property lists. There are four cases: a statement may or may not include a WRT clause, and in either case the statement may either store or retrieve. Storage by a statement including a WRT clause happens in the context recommended by the WRT clause; retrieval happens there or in the nearest ancestor context containing a property under the required indicator. (Retrieval gives the value NOSUCHPROPERTY if it fails to find or if the search is blocked by an occurrence of the property NOSUCHPROPERTY under the required indicator.) If the statement does not contain a WRT clause, then it is presumed to be somewhere within a member of an APPLY team of some statement that does have a WRT clause; that recommendation is used instead. (If

there are several such enclosing recommendations, the nearest one is used.) If there is no such enclosing recommendation, then the global context is used. Note that WRT GLOBAL may be superfluous, but WRT LOCAL seldom is.

These are all the property list statements and how they work.

(QPUT iexp inds/props [WRT-clause] [APPLY-team] [NAME-part])
e.g., (QPUT (PLUS X 0) SIMPLIFIESTO X)

This is the basic statement that inserts new properties onto a QLISP property list. Having instantiated iexp and all other parameters in the manner described above, it puts each of the properties onto the property list of (the datum generated by instantiating) iexp under the corresponding indicator with respect to the context supplied. (If no context is supplied, then the context defaults to the one recommended by the WRT-clause of some enclosing statement, if any, or else to GLOBAL.) Next, any team members present will be applied to the datum; if the WRT-clause is present, it will override any current context recommendation for the team members. Finally, the datum is returned as the value of the QPUT statement.

(QGET iexp [inds/props] [ind] [WRT-clause] [APPLY-team] [NAME-part])
e.g., (QGET (PLUS X $Y) SIMPLIFIESTO)

This statement is quite similar to QPUT, except that it retrieves information from the QLISP data base. The expression iexp and any of the properties may instantiate to any patterns. Some datum is retrieved which matches iexp, and its indicated properties are retrieved in the required context (or one of its ancestors, and NOSUCHPROPERTY is used for any properties which can not be found.) If a datum can be found that matches iexp and whose indicated properties match the patterns supplied, then the team members (if any) are applied to it; otherwise the QGET statement fails. Finally, there is the business of choosing a value to return as the value of the QGET statement: if an unmatched indicator is present, then the corresponding property (or NOSUCHPROPERTY) is returned; otherwise the datum matching iexp is returned.

The rest of this section treats statements concerned with truth or falsity of a proposition. They all follow a convention of using the proposition's MODELVALUE property to store its truth value.

(ASSERT iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])

performs (QPUT iexp [inds/props] MODELVALUE T [WRT-clause] [APPLY-team] [NAME-part])
e.g., (ASSERT (SIMPLIFIED (PLUS X 0)))
Directed failure can be used in conjunction with the APPLY team to effect any desired consistency checking.

(DENY iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
performs (QPUT iexp [inds/props] MODELVALUE NIL [WRT-clause] [APPLY-team] [NAME-part])
e.g., (DENY (SIMPLIFIED (PLUS X 1)))
Again, the APPLY team can be used to make sure the denial is consistent before permitting it to remain.

(IS iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part] [THEN-part])
performs (QGET iexp [inds/props] MODELVALUE T [WRT-clause] [APPLY-team] [NAME-part])
e.g., (IS (SIMPLIFIED (PLUS X $Y)))
In addition to performing the indicated QGET, the IS statement can use its THEN part to provide a primitive form of backtracking. After the QGET retrieval has been performed (and the team members applied), a backtrack point is established and the THEN forms evaluated in turn. (Presumably, they will test the values given QLISP variables by the pattern matching operation in the QGET retrieval.) If one of the THEN forms fail, another datum will be retrieved (and the team members applied to it), a backtrack point established, and the THEN forms tried again. This process continues until a datum is retrieved for which none of the THEN forms fail (or no more QGET-able data are to be found). The IS statement returns the satisfactory datum if there is one, or else it fails.

(ISNT iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
performs (QGET iexp [inds/props] MODELVALUE NIL [WRT-clause] [APPLY-team] [NAME-part])

(INSTANCES iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
e.g., (INSTANCES (SIMPLIFIED X))
A statement of this statement type returns a class of all data for which
(QGET iexp [inds/props] MODELVALUE T [WRT-clause] [APPLY-team] [NAME-part])
would not fail. (The MODELVALUE T check is not performed if MODELVALUE is among the indicators supplied.)

This next one also tests the truth of a proposition, but it succeeds when the proposition is known to be true or can be derived from other knowledge available to QLISP.

(GOAL iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
first performs (IS iexp [inds/props] [WRT-clause] [NAME-part])
(Notice that the APPLY team is omitted.) If a datum is retrieved, it is returned as the value of the GOAL statement. Otherwise, the team functions are applied successively to the instantiated iexp until some team member returns without failing. The value it returns is returned as the value of the GOAL statement. (If all team members fail, then so does the GOAL statement.)

For example,
(GOAL (SIMPLIFIED $X) APPLY $SIMPRULES)
will see whether (the instantiation of) $X is true. If it has not already been simplified, then the functions in (the instantiation of) $SIMPRULES will be used in an attempt to simplify it.

Finally, here is one to delete a proposition's MODELVALUE so that both IS and ISNT will fail on it.

(DELETE iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
resembles (INSTANCES iexp [inds/props] [WRT-clause] [APPLY-team] [NAME-part])
The difference is the treatment of the MODELVALUE property. Immediately upon retrieval, each datum is checked for the presence of a MODELVALUE property. If that property is absent, the datum is rejected; otherwise the property is removed before the team members are applied to the datum. The MODELVALUE is deleted by assigning it the property NOSUCHPROPERTY so that any more global MODELVALUE will neither be disturbed in its context nor appear in the given context.

4. More Elements of the QLISP Language

Some of the statement types do not fit neatly into the other categories by that the statement types are grouped. These first two are closely related to those which handle QLISP property lists, even though they do not touch property lists themselves.

(CASES iexp [WRT-clause] [APPLY-team] [NAME-part])
This one is just like a GOAL, except that it will not even try to retrieve anything from the data base before applying the team members to (the

instantiation of) iexp.

(FAIL [iexp])

Here lies an entry to the failure mechanism. The component iexp is optional and tells where the failure should occur. Failure handling is discussed in the section on backtracking and failure, under the ATTEMPT statement and also in the discussions of the APPLY teams of various of the statements. The iexp parameter is intended primarily for use in conjunction with APPLY teams and can take these forms (on instantiation).

If it is missing, the failure occurs at site of the FAIL statement.

If it is CALLER, then the failure happens to the QLISP statement which invoked the nearest enclosing APPLY team member.

If it is UTTERLY, then the failure goes back to the nearest enclosing execution of LISPX. In other words, (FAIL UTTERLY) is a programmed ^X; further details will be found in Section IV.J.

Otherwise, it is the NAME of some statement which invoked some enclosing APPLY team member; the nearest such statement fails.

The next statement is a conditional statement whose decision is dictated by failure or lack thereof in its predicate, rather than the more usual truth or falsehood.

(ATTEMPT val [THEN-part] [ELSE-part])

This statement first evaluates val. If the evaluation does not fail, ATTEMPT evaluates the forms in the THEN part ant returns the value of the last one (or the result of evaluating val if there is no THEN part). If the evaluation of val fails, ATTEMPT evaluates the forms in the ELSE part and returns the value of the last one (or NIL if there is no ELSE part). ATTEMPT fails only if its its attempted evaluation of the THEN or ELSE part fails.

Moving further afield, we can consider two statements that evaluate and instantiate some expression in a specified context.

(QDO iexp WRT-clause)

The WRT-clause is required; iexp has already been instantiated in the current context, and QDO evaluates it in the given context with the given context also recommended. QDO expects (the instantiation of) iexp to be a tuple so that the evaluation will do something interesting.

(VAL exp WRT-clause)

Again, the WRT-clause is required; exp is instantiated in the given context. (The statement type derives its name from its original use: to get the value of some QLISP variable in any given context.)

The next one is allied.

(UNBOUNDP exp)

It evaluates exp and returns a value of T or NIL, according to whether any unbound QLISP variables were encountered. UNBOUNDP avoids causing any errors by using ERRORSET protection when it evaluates exp.

The next three represent a QLISP adaptation of the PROG feature of LISP.

QPROG (format is like PROG)

This differs from PROG in only two ways: entering it pushes the current QLISP context and makes the new one current as well as entering a new scope for LISP variables; and it admits declaration of QLISP variables among the declarations of PROG variables. (Any QLISP variables among the declarations of local variables are shown to be QLISP variables by having the _ prefix.) Analogous to the practice of giving LISP local variables the value NIL by default, any QLISP local variables defaulting their initial values will be given the (non-) value NOSUCHPROPERTY. Otherwise, a QPROG is exactly like any other PROG in LISP.

(QRETURN iexp)

Other than instantiating its argument rather than evaluating it, QRETURN is just like the RETURN of LISP.

(QDECLARE var WRT-clause)

The WRT-clause is required and var must be a QLISP variable with an _ prefix. The variable is given a local binding in the specified context; it is given the value NOSUCHPROPERTY in that context. The effect on var is the same as using it as a QPROG variable, but QDECLARE is considerably more flexible (and less highly automated) than QPROG.

The next one is a LAMBDA function, not a statement. It is used for its side effect, which is to systematically instate an abbreviated notation.

deftype (fn; type)

It declares fn to take a single argument of the specified type, and it declares the spread-out tuple notation to be an abbreviation for the

single-argument notation. For example, if EQUAL were declared to take a class, then the tuple (EQUAL A B) would be regarded as an abbreviation for the tuple (EQUAL (CLASS A B)). The QLISP interfacing mechanism handles the abbreviations automatically: it expands them when building new tuples and abbreviates tuples when passing them back to LISP. Therein lies the difficulty: the abbreviation is handled partly at translation time and partly at execution time. This has the effect of requiring that all relevant deftypes be in effect whenever any affected function is compiled or interpreted; Section IV.E gives another detail on DEFTYPE.

While QGET will search for a property list, QPUT will not; the search is needed when some extant property must be changed where it is and cannot merely be hidden in some descendant context. This statement fills that need.

(WHERE iexp)

It performs the usual context search, starting from the local context and proceeding to successive ancestors, terminating in the first context containing a property list. Then it returns a context designator for that context.

F. QLAMBDA Functions

Functions may be defined by using the word QLAMBDA in a manner similar to the use of LAMBDA and NLAMBDA. In the place of the usual LISP-style argument list, QLAMBDA functions take a pattern, usually containing _ variables. Upon entering a QLAMBDA function, its pattern is matched to the datum supplied as an argument. Then the current context is pushed, the new one made current, and any _ variables in the QLAMBDA pattern take their new values in the new context. A QLAMBDA function can take advantage of any possible indeterminacies in the unification of the QLAMBDA pattern to the argument supplied: if the QLAMBDA pattern is immediately followed by the word BACKTRACK and a failure generated inside the function propagates out to the level of the QLAMBDA, then another match is generated and the function body is entered again. The failure is propagated out of the QLAMBDA only when there remain no more matches to be tried.

A QLAMBDA function will give special treatment to any _ variables in its argument: they will be distinct from any _ variables in the QLAMBDA pattern even though they might have the same names. Furthermore, the _ variables (if any) in the argument will receive their values in the "outside" context -- the one which was current just before the QLAMBDA function was invoked. For example, if a QLAMBDA function has (BAG A _X) for its pattern and (BAG B _X) for its

argument, then X receives the value A in the outside context and B in the new context generated for the QLAMBDA. In addition, _ variables can be given as values to other _ variables; so if you apply a QLAMBDA whose pattern is (_U V _W) to the tuple (X _Y _Z), then U, W, Y, and Z will get the values X, _Z, V, and _W, respectively. (In this example, the outside variables _Y and _Z will get their values in the outside context, and the inside variables _U and _W will get their values in the inside context.) One way this "renaming" can be used is to pass values back out of a QLAMBDA via variables in its argument: a (MATCHQQ $W FOO) would instantiate $W and FOO (to _Z and FOO) and then unify the instantiations, giving _Z the value FOO in this example.

G. Special Relations

This section covers the facilities by which QLISP internally handles equivalence relations and transitive relations. (Together, these two kinds of relations are called "special relations".) Internally, special relations are handled in groups of six relations which interact in the same way as those of linear ordering: equivalence, strong and weak ordering, and their classical negatives.

Special relations are handled by specialized manipulations of property lists. Perhaps these manipulations are best explained by means of an example whose generalization should be obvious. I will use the numeric ordering relations LT, LTQ, GT, GTQ, EQ, and NEQ, and assume they are deftyped so that:

LT, LTQ, GT and GTQ take vectors; and
EQ and NEQ take classes.

If I were to assert (LT A B) (i.e., A<B) in some context, then QLISP would store B on the property list of A in that context. Then a later query of (EQ A B) in the same context would find that it could not possibly be true because (LT A B) is already known. That is the underlying idea of QLISP special relations; the next two paragraphs refine this scheme, and the remainder of the section describes how to use it.

When a special relation is stored, its form may be changed to facilitate these special deductions. The six relations (and their negations) are all collapsed into four: EQ, NEQ, LT, and LTQ in our example. Equivalence relations work this way, even from an intuitionistic point of view, because QLISP distinguishes falsity of a proposition from its lack of truth. (Cf. ASSERT, DENY and DELETE in Section III.E.3.) However, the restriction of transitive relations to linear ordering (in the presence of a coupled equivalence relation) comes from the way it regards falsity of (LT A B) as equivalent to truth of (GTQ A B), for example.

Special relations gain greatly in power from the way they interact with the context mechanism. All QLISP property list manipulations refer to a specific context, and this applies as well to the manipulations occasioned by the special relations handler. The added power comes from the deductive retrieval available for special relations: a deduction in a context will make use of all pertinent facts visible from that context, regardless of the contexts in which the individual facts are stored. For example, if (LTQ A B) is known in some context and in some hypothetical subcontext (LTQ B A) becomes true, then equivalence of A and B can be immediately deduced in the subcontext without its leaking out to any enclosing context.

The QLISP facilities that handle special relations work only with elementary statements about whether a special relation holds between items. More specifically, all accesses to the data base are in terms of 2-tuples of a particular format. The first element of the tuple is the name of a special relation (e.g., GTQ), and the second element is a vector (or class) of elements in the domain of the relation. The choice between vector and class must be as in our example: the equivalence relation and its negation take a class; and the other four take a vector. When the vector (or class) has more than two elements, the notation is extended conjunctively. That is, asserting (LT A B C) is tantamount to asserting both (LT A B) and (LT B C). This conjunction is subject to de Morgan's law: denying (EQ A B C) merely states that A, B, and C are not all EQ. In the remainder of this section, the abbreviation "siexp" will stand for one of these special 2-tuples; the "i" in the abbreviation emphasizes that the tuple is instantiated preliminary to its use.

Special relations are declared as such to QLISP by means of this LAMBDA function.

specialrelations [rname; rellist]

This function declares the functions contained on rellist to be special and rname to be their collective name; rname must be supplied and rellist must be a list containing at least one relation name. The role of a relation is declared by its position on rellist; the relations of the current example could be declared by rellist having the value (LT LTQ GT GTQ EQ NEQ). The list can be truncated, or it can contain a NIL for any position you do not wish to specify; QLISP needs all six and will supply names you leave out.

This next function declares a useful special case.

number-relations []

It instates special relations just as in the

example of this section. That is, the relations LT, LTQ, GT, GTQ, EQ, and NEQ are declared to be special and deftyped just as the second paragraph outlines.

Special relations are automatically stored in the data base. That is, when the "iexp" argument of an ASSERT, DELETE or DENY statement happens to be of the special "siexp" form of this section, then QLISP updates the special deductive information in the data base. (Understand that iexp is also stored directly and can be retrieved by QGET and IS, etc. just as if special relations were not involved.)

This statement queries the data base without making any changes to it.

(ISREL? siexp [WRT-clause])

If siexp is stored directly in the data base or can be deduced from information stored there, then the statement returns T; otherwise it fails. The ISREL? statement works only with special relations; that is, siexp must be a 2-tuple of the special type outlined above.

IV. The QLISP System

Since Part IV is concerned with QLISP as a programming system, it necessarily refers to INTERLISP as a programming system. This is unavoidable. Moreover, a concern of this part is to explain to the cognoscente some of the details of how QLISP was integrated into the INTERLISP system. If you merely want to find out the things you need to know to use QLISP, you need only read Sections IV.B and IV.E; the rest of this Part will become more interesting to you when (and if) you make fuller use of the QLISP system. The INTERLISP reference manual contains a detailed explanation of INTERLISP as a programming system.

A. DWIM Corrections and CLISP Transformations

INTERLISP errors provide the principal means by which control passes into the QLISP system. Most elements of the QLISP language are invalid INTERLISP constructs, which are interpreted by means of the machinery DWIM provides to correct errors. In fact, QLISP constructs are treated as CLISP: they are transformed into internal calls on the QLISP system, and the translations are stored (and later retrieved) by the same facilities CLISP uses to handle its own transformations.

If you are not familiar with INTERLISP, you may be interested in this brief explanation of DWIM and CLISP as they relate to QLISP. Whenever the interpreter or the compiler encounters a construct that is not valid LISP (e.g., a call on an undefined function), the DWIM machinery attempts to correct the suspected error. If DWIM succeeds, it alters the program so that it will never again cause that particular error. CLISP is a package which uses the DWIM facilities to interpret various invalid LISP forms iterative statements or other extended constructs. When CLISP translates one of its statements into valid LISP, it modifies the program so that the valid LISP will be interpreted or compiled, but the original CLISP will be used by the editor and symbolic file package. QLISP uses those CLISP facilities to transform QLISP statements to valid LISP while still retaining the original QLISP in the symbolic function definition.

Speed motivated this approach. It takes a while to warm up the DWIM machinery, but then the situation is analyzed once and the result saved for fast retrieval in the future. A statement body is parsed once, and later executions of the same statement will directly retrieve the results of that syntactic analysis considerably faster than the statement could be reparsed. Then, the speed paid for analysis of a statement the first time INTERLISP encounters

it is regained when it must again be understood.

There are several important consequences of this approach. The words that correspond to the statement types (e.g., QPU1) are not defined as functions and will not appear on the stack for you (or your program) to find. Also, QLISP variables do not have values stored where LISP can find them, nor will its methods help you find them. Both those facts must remain true for QLISP to function properly; if you give a LISP value to a variable whose first character is a $, then it will stop working as a QLISP variable.

B. Tracing Functions in QLISP

It is not always possible to trace (or otherwise break) functions in a QLISP program because of an unfortunate interaction between the break package and the implementation of failure and backtracking. The fix is simple: there is a pair of functions QTRACE and UNQTRACE, which circumvent the problem. They are both NLAMBDA nospread functions taking a list of function names and operating on the given functions; UNQTRACE will also accept NIL as an argument, in which case it will untrace everything that is traced. QA4:TRACEDFNS always contains a list of the functions that are currently traced. UNQTRACE is undoable; QTRACE is not because of some more interactions between failure and undoing. I will postpone discussing those interactions until the last paragraph of this section; first come the various ways tracing is a richer facility than that outlined just above.

The most conspicuous exception to this simple picture is that it will not ignore you even if you ignore it. Since QLISP can often invoke functions in a somewhat arbitrary order, it attempts to dispel some of the resultant mystery by automatically tracing all QLAMBDA functions (unless you say otherwise.) You have two means by which to control this behavior: QTRACEALL and UNQTRACE. As long as QTRACEALL has the value NIL (initially T), no function will be automatically traced. (Do note that QTRACEALL only applies to automatic tracing, and that explicit use of the function QTRACE just ignores the QTRACEALL flag.) A specific function can be flagged for exclusion from automatic tracing by untracing it at a moment when it is not traced; it need not be defined for this flagging to take effect. This flag will be removed if such a function is ever traced explicitly; you need to untrace it TWICE to restore the flag.

A fairly subtle question is raised in the previous paragraph: what is the instant at which the automatic tracing machinery looks at a QLAMBDA function (and QTRACEALL and the "don't trace" flag) and decides whether to proceed? There are different answers for different

functions. For compiled functions, the decision is made when the compiled definition is stored; the following sections on the file package and compiled QLISP detail further considerations. For interpreted functions, the decision is made when the function is translated from a QLAMBDA to the corresponding LISP function the first time it is entered. When the compiler translates a QLAMBDA in order to compile it, however, it will not be automatically qtraced. The translation is a CLISP transformation (see the next section) and will be discarded if you change the function by editing it. Then the function will be translated again the next time you enter it, and automatic qtracing will be reconsidered then.

The printout generated by qtraced functions is indented to show the depth of recursion. After a certain depth is reached, though, the indentation wraps around to the left margin. If your program goes very deep in recursion, it can be confusing to keep track of the number of wraparounds. The depth of recursion is counted and printed out after the function name, both on entry and on exit. If you find the numbers objectionable, you can turn them off by setting QTRACECOUNTFLG to NIL; its initial value is T.

Since qtracing and ungtracing a function are done by advising and unadvising it, you will have a problem if you put your own advice on a QLAMBDA function (or any other function) that is or will be qtraced. The problem will show up when you ungtrace or unadvise it. Ungtracing will make your advice disappear too, and unadvising it will make the qtrace advice vanish without cleaning up the other bookkeeping information maintained by the qtracing package. Beware.

Qtracing uses the ADVISE package instead of the BREAK package just because of the unfortunate interactions mentioned in the first paragraph. When a function is qtraced, it is advised to print out its argument on entry and its value on exit. Recall that backtracking is currently implemented via the INTERLISP undo mechanism and that failure is currently registered by causing an INTERLISP error to trigger that undoing. Since tracing is implemented via a special kind of break and since BREAK1 enters the traced function under ERRORSET protection, a failure's error cannot propagate back through a call to a traced function. It is even worse than that: backtracking is implemented via UNDONLSETQ, whose effects are confined to one LISPX event; even were the error to be propagated back from the break, the event number might well be changed, and only part of the desired backtracking would actually get done. Finally, values of QLISP data are often passed in their internal forms (more about that in Section IV.F) and carefully translated to their public representations before they leave QLISP; by supplying the routines which print the trace

information we can ensure that the correct version will be printed in all cases.

QLISP statements can also be qtraced. Use the statement name (e.g., MATCH) to QTRACE and UNQTRACE just as if it were a function. The qtracing machinery will automatically use the corresponding execution routine (e.g., MATCH:ER) instead.

C. Variables and Symbolic Functions

QLAMBDA functions and QLISP variables certainly must be treated somewhat differently from LISP functions and variables; that is their motivation. The implementation does its level best to conceal any inessential distinctions from the casual observer, but there are some that still show through. The rest of this section discusses some of the grossest ways those distinctions are visible.

QLAMBDA functions are automatically translated to LISP functions whenever necessary; this act is treated as a CLISP transformation. There is an added operation: any QLAMBDA function will be automatically qtraced (see the previous section) if it is translated while the flag QTRACEALL has the value T (its initial value). (Actually, automatic qtracing is slightly more complicated than that: it only happens to atoms whose definitions are QLAMBDA expressions; other QLAMBDA expressions simply can not be qtraced.)

EDITF can edit QLAMBDA functions just the same as it can edit an ordinary LISP function. If EDITF is given a qtraced function, it will automatically unqtrace it before editing it. Remember that the translation will be thrown away when you invalidate it by changing the function and that automatic qtracing is considered any time a QLAMBDA function is translated upon entry.

QLISP variables can be edited by using EDITV in almost the conventional manner. Such a variable should be given the $ prefix when supplied to EDITV. The value edited will be the one visible in the current context; it will be retrieved before the editor is entered and stored back on exit from the editor. If the editor is exited abnormally (e.g., the STOP command or ^D), the variable's value will not be changed.

QLISP variables are saved on files by giving them the $ prefix and naming them in the QVARS prettycommand. QVARS behaves in the same way as the conventional VARS command, except that QVARS will save both QLISP variables and LISP variables. In the default case, the file package will save and load the values of QLISP variables with respect to the global context. The way to override the default is to change the value of the QLISP variable QLISP:FILECTX. When

the time comes to save the value of a QLISP variable on a file (or restore the value), $QLISP:FILECTX is used in a WRT clause of a VAL or MATCHQQ statement. Thus, $QLISP:FILECTX is instantiated with respect to the local context at the time a QLISP variable's value is saved (or restored); initially it has the value GLOBAL in the global context and no other values.

D. Compiling QLISP

There are certain things you have to do to get your code to compile correctly. They are as highly automated as possible, but you need to cooperate to some degree, as the next paragraph discusses. The rest of the section concerns dwimification. It is performed automatically to the degree necessary, but if you know (and care) about some of its inner workings, you can tune the procedure to save yourself some compilation time. The only way you need to cooperate with the dwimification process, though, is to compile your code in a QLISP of the same flavor (see Section IV.J) as you will use to run the compiled code.

The compiler needs to know a lot about QLISP to compile a function containing QLISP. As long as you compile from a QLISP system, the compiler will already know about all the standard features. However, you still need to inform it of any additional options or extensions you use, such as DEFTYPEs (Section III.E.4) or any statements you have added to the language (Section IV.H). The way to inform the compiler is to load all your files containing any of your own extensions before you compile anything (or at least to LOADFROM them first). READ THOSE PREVIOUS TWO SENTENCES AGAIN and take heed. It is an endless source of fascination to watch people take a file that uses their own special DEFTYPEs, compile it without informing the compiler about them, and then wonder why the compiled code does not work. I'm all for telepathic compilers, but this one is only human - I just haven't had the time to raise its consciousness to those planes.

Any function containing QLISP must be dwimified before it is compiled because all translations of QLISP to LISP are performed as CLISP transformations. The standard ways to automate this are augmented by a few extensions which are specifically oriented toward QLISP. In fact the preferred way to force dwimification for the sake of QLISP is to use the controls described in this section (rather than, say, setting DWIMIFYCOMPFLG to T) because these controls make sure, for example, that NOSPELLFLG and CLISPRETRANFLG are both set to T during the dwimification.

The default assumption is that any function might contain QLISP, and so dwimification (with a QLISP orientation) is forced for any function compiled (in a QLISP

system). The reasoning, of course, is that you would not be using a QLISP system in the first place if the assumption were not fairly good. Remember that the assumption is only a default and that, while this kind of dwimification is sometimes necessary, it is always harmless because NOSPELLFLG is T while it is going on.

The default assumption is controlled by the variable QLISPIFYCOMPFLG. If DWIMIFYCOMPFLG is NIL, then QLISPIFYCOMPFLG is checked and dwimification (with the QLISP flavor) is done or not, according to the value of QLISPIFYCOMPFLG, (initially T, of course). All this can be modified for a specific file if DWIMIFYCOMPFLG is null. If compilation is from a file and that file (i.e., the atom whose print name is the name-part of the file) has the property QLISP:FILE, then the property's value will dictate the QLISP dwimification, regardless of the state of QLISPIFYCOMPFLG. If the property is T or NIL, then the dwimification will happen or not, respectively, to functions compiled from the file. (Note, though, that absence of the property is different from its having the value NIL).

There are two more possibilities for the property, however. If it is a list of function names, then, when the file is compiled, the dwimification will happen to those functions and no others. (This option is intended principally for large files of which only a few functions contain QLISP. The user gains the time not spent in dwimifying much of the file at the cost of putting more care into maintaining the property's value.) Finally, the property can be NOQLISP if there is truly no QLISP contained on the file. This last option is discussed in the next section, but its bearing here is that the file will be compiled just as if the property were NIL.

Compiled functions are peculiar in another way: automatic qtracing. Recall that interpreted QLAMBDA functions are automatically qtraced (if QTRACEALL is T) when they are first entered. Compiled functions, howver, are translated when they are compiled, and it is meaningless to decide whether to qtrace them until they are loaded into the system in which they will be used. For this reason, the decision on automatically qtracing a compiled function is applied to the compiled definition at the time it is stored. Thus, for example, if a file is compiled and the compiled definitions also stored, automatic qtracing will be decided independently on two occasions: when the newly compiled definition is stored by the compiler; and later when it is loaded from the compiled file.

E. The File Package

The file package will (usually) modify your fileCOMS in various ways to help lubricate the process of saving QLISP

programs and variables on symbolic files; this section details the ways in which the file package behaves differently from that of standard unmodified INTERLISP. These modifications are highly automated and you need not read past this paragraph if you are willing to endure a certain amount of stumbling, to trust QLISP to do the right things in all cases, and otherwise to trust your intuitions. All modifications necessary will be performed automatically, including those necessary to rectify the action of outmoded versions of QLISP. The only cooperation needed from you is that you make your symbolic files via PRETTYDEF (or MAKEFILE, CLEANUP, and their friends) and that you generate your compiled files by using BCOMPL (or TCOMPL, CLEANUP, etc.) rather than by giving an output file name to COMPSET.

Broadly speaking, there are two kinds of things QLISP may do to your fileCOMS: it might modify your commands, and it could add commands of its own. Any of your commands that are modified will stay where they are, and you should feel free to take whatever liberties you wish with them. Its own commands, however, will be added to the end of the list of commands; you could confuse it by modifying them, but you can freely reorder them (or add your own commands after them), and it will make sure they are at the end. (The point of its commands following all of yours is, of course, that their effects will take place after the file is loaded as you wish it to be.) All these modifications are made by PRETTYDEF just before it writes out the file; they all happen without comment and without user interaction.

This paragraph discusses the two things that might happen to your own commands. Since all $variables must be in QVARS commands, QLISP will make sure they are. It will change a VARS command to a QVARS command if it saves the value of a $variable, and it will wrap a QVARS command around any atom starting with a $ which appears at the top level of your fileCOMS. Also, since any DEFTYPE on the file must be obeyed when the file is compiled as well as when it's loaded, any P commands containing a DEFTYPE expression at the top level (i.e., any command matching the edit pattern (P --(DEFTYPE --)--)) will be embedded in a DECLARE: command of the appropriate form. Change the DECLARE: flags at your hazard: this is the only way in which you should not feel free to change its changes to your commands.

The remainder of this section describes the commands QLISP will add on its own initiative and the ways you can guide it. If you take the default handling, it will add just one command (of the form (QLISP:SAVEFILEINFO)) to your fileCOMS. That, in its turn, will write out forms identifying all the QLAMBDA functions in the file and their patterns in order to connect them to the compiler, automatic atracing (discussed above), and the FILTERPATS machinery (Section IV.J). Since different flavors of QLISP (Section

IV.J) may compile a given construct differently, that command will also write something to save a record of the QLISP from which you compile it; that will be checked against the QLISP you load the compiled file into and any incompatibility reported at that time.

So that your commands will be read back in at all the right times, we have also modified LOADFNS to take account of any functions you load individually. It will automatically acquire the QLAMBDA pattern for any function you read. Also, any compiled QLAMBDA function will be automatically qtraced (if appropriate) and linked in to the FILTERPATS mechanisms. Finally, if LOADFNS loads the fileCOMS, then it will load the QLAMBDA patterns for all functions on the file. This last is of interest to any one who might try loading enough bits and pieces of the file to write it out again without ever doing a LOADFROM of the file.

The file's QLISP:FILE property affects its compilation, as discussed in the previous section. If the property is present, it will also be saved automatically. If the value is T, NIL, or a list of function names then it will also be saved by the (QLISP:SAVEFILEINFO) command. If the value is NOQLISP (signifying a complete absence of QLISP from the file), then it will be saved in a slightly different manner. In that case, your fileCOMS will contain only one command generated by QLISP, and that command will restore the property to the file when it is loaded or compiled.

The file package, then, uses two prettycommands not included in a standard INTERLISP; they can be a real nuisance if you make a file from QLISP, load it into a standard INTERLISP, and then try to do almost anything involving the standard file package. This paragraph and the next are concerned with the means available to alleviate this problem. If the file contains QLISP variables, then there are many problems, but there could easily be QVARS commands saving only LISP variables. The QLISP:SAVEFILEINFO command is even worse though, because QLISP will replace it even if you remove it; there is no way to get rid of it without giving your file the NOQLISP property under the QLISP:FILE indicator. (There is a bifurcated reason the defaults are that way: if you are using a QLISP system, there is likely to be some QLISP buried somewhere in your functions; and PRETTYDEF would be hard put to find it, especially if the file is not completely loaded.)

It can be reasonable to load a QLISP file into standard INTERLISP if the file contains no QLISP variables. For example, you may wish to edit a few functions or even enter those free of QLISP and then re-make the file. You can give various degrees of self-containment to a file by changing the value of the flag QLISP:SELFCONTAINEDFILESFLG before you

make the file from QLISP. This flag is examined by PRETTYDEF just before the file is made; four different values are distinguished.

DONTBOTHER (the initial value) evokes just the behavior described above. That is, the commands generated by QLISP will be on your filecoms but your file will contain nothing to support them. This is intended for files which will only be loaded into QLISP. It is the default because files generated by QLISP usually will be loaded only into a QLISP system.

NOQLISP causes the commands generated by QLISP on its own initiative to be absent from the fileCOMS. (See the final sentences of this paragraph.)

NIL will cause the file to load a primitive bootstrap from a file in the QLISP directory.

T will cause an even more primitive bootstrap to be written on the file itself, so that the file is as self-contained as could be expected as long as it still contains QLISP.

So then, there are five degrees of self-containment which a file may have, corresponding to the four values of QLISP:SELFCONTAINEDFILESFLG and to the file's QLISP:FILE property being NOQLISP. The first four are global in the sense that they hold for all files made when the flag has the appropriate setting. The last one, however, is specific to a particular file both in the sense that it overrides the first four and in the sense that it is saved with the file itself and persists until it is removed (i.e., from the file's property list).

F. Data Storage

QLISP data are represented to you in the forms mentioned previously. However, their internal representation differs considerably. This section concerns the interrelations between those internal and external forms. The internal form contains pointers to the datum's property lists, to its external forms, and to the internal forms of its constituents (if any). Whenever a QLISP datum is passed from the internal parts of QLISP to ordinary LISP functions, its external form is retrieved and passed instead. Conversely, when an external form is presented to QLISP, its internal form is retrieved and used internally. Thus, the QLISP system ensures that its internal manipulations are performed on its internal representations of the data, while your program will only have the more intuitive external forms with which to contend.

The whole procedure is analogous to the mapping between atoms and their print names. The ordinary LISP user need not worry about the internal connection between the two; the print name is used for printing and reading, and the internal form is used for property list manipulations. The

translation is automatic. It also resembles like CLISP transformations, where the translation is interpreted or compiled, while the original CLISP is edited or prettyprinted.

One of the forms will of course be generated before the other; in fact, the alternate may never be required, so its generation is postponed until it is called for. While the external form is stored on the internal form, the former has no place for a similar pointer to the latter. The inward mapping rather uses INTERLISP hash links. A translation is generated only when it is required and not found. This is not quite true for an external form that does not have a hash link to the corresponding internal form; the hash link can be absent for an external form that is EQUAL but not EQ to an external form generated by QLISP. When the hash link is absent, the QLISP data base is examined to find the corresponding internal form; if none is found, a new internal form is constructed and inserted into the data base. QLISP maintains an index into its data base to make the search quite fast; the index is currently implemented in a discrimination net.

While computation time is saved by storing the external form on the internal form, this practice has a consequence that might not be obvious at first: if you destructively modify the external form, you will permanently change it. Make sure your destructive modifications are done to a copy instead.

You have some control over the use and generation of the hash links used for the inward mapping. Whenever an external form is generated from an internal form, the hash link back will be stored unless QLISP:HASHTRANSFLG is NIL (initially T) and QLISP:HASHTRANSARRAY is also NIL. Otherwise QLISP:HASHTRANSARRAY will be used as a hash array argument to PUTHASH. (If it is a list whose first element is a number, then the number will first be replaced by a new hash array of that size.) When the reverse translation is required, the value of the hash array is examined. If it is NIL, then the hash check is bypassed; otherwise it is used as a hash array argument to GETHASH. The initial value of QLISP:HASHTRANSARRAY is the dotted pair (200 . 200).

The qtracing mechanism prints out the external forms of the arguments and values of any functions it is monitoring. Occasionally people are interested in seeing the actual values that are being passed around, usually because they suspect the presence of a bug in the translation mechanism. The translation is controlled by QLISP:TRACERESPONDFLG. Initially it has the value T, but if you set its value to NIL then the values will be printed exactly as they are passed.

G. Restrictions and Caveats

QLISP has been integrated into CLISP so that the two work very smoothly together. There are, however, a few areas where the interface is not as smooth as might be desired, and this section tells you how to avoid the difficulties. Most of the awkwardness is caused by the historical fact that QLISP and CLISP developed concurrently and with imperfect communication; the rest comes from our having used INTERLISP features in ways of which you must take account to use them yourself.

The most obvious restriction of the second type centers around the UNDO features of INTERLISP. Backtracking is implemented by undoing unwanted side effects. Clearly, then, whenever a computation is backtracked, all traces of its undoable actions will vanish, such as loading a file. This can sometimes come as an unpleasant surprise.

INTERLISP provides many places where the user can affect its behavior: DWIMUSERFN is an example. In three important cases, the user is the QLISP system itself; this is crucial to the smooth coupling of QLISP into INTERLISP. However, our having used these handles (viz., COMPILEUSERFN, DWIMUSERFN and LISPXUSERFN) does not preclude your using them also, but you must ADVISE them rather than just defining them. Each of them tests its appropriateness and may perform some action. Your advice should check the value returned by our function and then proceed if ours is inappropriate (and signifies this by returning NIL.)

The interactions with CLISP are worse: they generally require that you forego some features of CLISP. The inset paragraphs detail them.

> Since the leftward arrow (_) has special significance to QLISP, it necessarily loses part of its CLISP meaning. That is, if it occurs as the first character of an atom, then the atom is a QLISP variable and cannot be part of a CLISP assignment statement. Moreover, you can not build an assignment statement using _ as an isolated atom.
>
> Another problem is presented by the QLISP MATCH statement and CLISP using MATCH internally to implement the CLISP pattern matching feature. The upshot of that conflict is that you can't use the latter facility. It also means that if you clispify a MATCH statement, you will generate erroneous code for the CLISP feature.
>
> Since QLISP statements must be dwimified to work properly, and since dwimification is actuated by an error, the statement types (e.g., MATCH or ISNT) cannot have function definitions.

The same applies to QLISP variables; no atom whose first character is a $ should ever have a LISP value.

All of these are restrictions. Some can hardly be noticed, and some can cause great difficulty; some are fundamental to the notion of QLISP, and some could yield to small amounts of implementational cleverness. Nonetheless, they definitely remain as restrictions.

H. Adding New Statements to the Language

The statement types included in the QLISP language are those that have tended to be useful. If another variant would be more useful, then this section is for you. Defining a new statement is a good alternative to writing a function to do the same work: statements are parsed when they are dwimified, normally when they are compiled or evaluated.

You declare a new statement type to QLISP with this undoable LAMBDA function.

newstatement [name; er; recipe]

It declares name to be the name of a statement type, er to be the associated execution routine and recipe to be a function used in parsing statements of the given type. It makes sure name has no function definition, in order to insure dwimification of any form whose CAR is name. The first thing done in translating such a statement is to apply the recipe to the CDR of the statement. The expected value of that application is a list of the form (STATEMENT name . newtail), with newtail a list. Each item of newtail is then translated to code that will, upon evaluation, produce the item's instantiation. The translation ultimately produced is a form whose CAR is er and whose CDR is the list of transformed newtail items. Thus, it is assumed that er will evaluate its arguments, and the main work of the recipe is to produce the items which will be instantiated to produce its arguments.

The statements that store and retrieve properties all take nearly the same format. Their recipes all use this LAMBDA function to help parse them.

parsestatement [tail]

(The argument list is somewhat different for internal reasons.) It produces a list suitable for use as the "newtail" just mentioned. It assumes the execution routine evaluates its arguments and takes an argument list like

er [iexp; apply; wrt; name; indsprops],

where (the normal evaluation of) the iexp, wrt, and

name arguments will produce the instantiations of the corresponding parts of the statement. The apply argument will produce a list of the members of the apply team. Similarly, the indsprops argument will produce an alternating list of all the indicators and properties in the original statement in the order given, except that any unmatched indicator will be moved to the end of the list. Instantiation will actually take place only when necessary; QUOTE will be used freely where only constants are involved.

An execution routine will normally be a LAMBDA function; if it is an NLAMBDA (spread or nospread), then dwimification might require some extra care. The normal default handling will be correct for an execution routine which is an NLAMBDA but still evaluates all its arguments; the rest of this paragraph tells how you can cause other behavior. Correct dwimification is especially important when a function is dwimified preliminary to compilation; if compiled code is not dwimified completely as it is compiled, then its subsequent execution will at least require the time-consuming intervention of the error correcting machinery. It may well cause an error. If a statement has an NLAMBDA execution routine, you can cause its translation to be selectively dwimified by using an extension of the notion of recipes. The "newstatement" function adds a RECIPE property to the property list of its "name" argument; the property's value is a list whose only item is the "recipe" argument. If the list contains a second function, it is used as a recipe for dwimification. This other recipe is applied to two arguments: the CDR of the translation and the translation itself; the resulting value is a list of forms that are dwimified.

I. Things That Came (Almost) for Free

A number of functions included in QLISP are general enough in their utility for me to publicize them and encourage you to use them. This section is devoted to describing them. They are all LAMBDA functions unless the contrary is specifically noted.

addtovarquietly [var; new]

It undoably adds NEW to the top-level value of VAR unless it is already there. If the value of VAR is not a list, it becomes a list whose sole element is NEW. It never types out a message, even if it changes the value of VAR.

advizaround [fn; old; new]

This one undoably moves function definitions around to give the effect of compiled advice. FN is the name of the function being advised; NEW is the name of a function you supply; OLD is the name under which

your function calls the original definition. When ADVIZAROUND has finished, calls to FN will come to your NEW function, and its calls to OLD will go to the original definition. If you supply NIL for the NEW (or OLD) argument, it will add the word NEW (or OLD) to the front of the FN argument. It also changes names as necessary so you can stack several ADVIZAROUNDs on top of one another without their getting confused.

filenamepart [name; suffix]

If SUFFIX is NIL, it returns the namepart of the file NAME; if NAME contains any occurrences of control-F or ESC, then TENEX is asked to expand the name. If SUFFIX is a string, it is packed onto the end of the name being returned. If SUFFIX is T, then the file's extension stays attached to the name when it is returned.

NOTES

This prettydefmacro ignores its arguments; since it will be printed as part of your fileCOMS, its effect is to let you include comments in your file which do not pertain to any particular function.

qlambdap [fn; okpropflg]

If FN is a function or a definition, it tells you whether FN is a QLAMBDA function. If FN is atomic and undefined, then QLAMBDAP will look on its property list for a saved definition (if OKPROPFLG is true) or for a QLAMBDA pattern saved there when you LOADFROM its file.

qlambdapattern [fn; checkedflg]

It returns the QLAMBDA pattern of FN; if QLAMBDAP is not true of FN, it returns NIL. (It will bypass the QLAMBDAP check to save some time if CHECKEDFLG is true.) This one also works if FN is a definition or the name of a function on a file from which you have done a LOADFROM.

qlisp:error [mess1; mess2; nobreak; brkexp; brkfn]

This one behaves much like the INTERLISP function ERROR but is somewhat more flexible. It simulates the action of ERROR on MESS1, MESS2, and NOBREAK; BRKEXP and BRKFN are given to BREAK1 if an interactive break occurs.

/setq [x; y]

This NLAMBDA function is conspicuous by its absence from the standard INTERLISP system.

smartmovd [from; to; copyflg; props]

QLAMBDA functions have auxiliary information on their property lists; if PROPS is NIL, then SMARTMOVD will do the corresponding MOVD and also move those

properties as well as informing the qtrace and FILTERPATS mechanisms. If PROPS is a list, then properties on PROPS will be moved.

J. Other Goodies in the QLISP System

Together, the preceding sections tell you almost everything you would want to know about QLISP, at least at this manual's level of detail. Each section concentrates on a particular theme and covers various topics unified by that theme. This section discusses subjects that are of some interest to some QLISP users but that are too miscellaneous for inclusion elsewhere.

You can speed up the processing of your APPLY teams (at some cost of storage space) by using the FILTERPATS mechanism. Its central idea capitalizes on the very regular way APPLY teams are used: the team members are successively applied to a given datum. In the most useful applications of APPLY teams, the datum will fail to match the QLAMBDA patterns of most team members. In most cases, the unsuccessful attempt to enter a team member can be eliminated by this technique. QLAMBDA functions are stored in such a way that, given a datum, the FILTERPATS machinery can retrieve (the names of) all the functions to which it is applicable. On one side of the tradeoff, then, is the time required to unsuccessfully attempt to enter many team members; on the other side are both the space required to store the FILTERPATS information and also the time required to retrieve (the names of) the applicable functions and discover which team members are among them. The choice is left open since it depends so strongly on the particular problem at hand; you exercise it by using this LAMBDA function.

filterpats [sw]

You turn the FILTERPATS mechanism on and off by applying this function to T or NIL, respectively. Turning it on has these effects:

when an APPLY team is used, it is filtered in the manner just outlined; and

when a QLAMBDA function is translated, its name and QLAMBDA pattern are stored for that filtering.

The time and space required to store the FILTERPATS information is nominal but noticeable. However, the filtered retrieval can cost as much as several unsuccessful QLAMBDA applications.

A special QLISP feature allows interactive recovery from one of the most frequent user errors. If a required value is missing from a $ variable, then QLISP generates a special error (viz. UNBOUND VARIABLE) and enters the normal BREAKCHECK routines. If LISP goes into an interactive break

and you give the variable a value, then you can give the OK command to continue the operation of your program.

The interrupt character control-D is especially contrived in QLISP to behave correctly while giving every appearance of being completely standard. (The special extensions clean up after any tentative computations and ensure that their side effects vanish.) The implementation of control-D gives it the net effect of successive failures back to the top level of INTERLISP, though at a considerable saving of computation time. The main observable difference between the QLISP control-D and the standard one of INTERLISP is a slightly different typeout. QLISP also offers another interrupt character to largely supplant control-E, whose usefulness is largely vitiated by the fact that failure uses the error mechanisms. You can cause a local failure with control-E, and you can fail clear back to the top with control-D, but another case is often useful. If you are editing a program and try it out (e.g., using USEREXEC), you will likely want to abort the trial without aborting the edit session. Control-X will do precisely that. More particularly, control-X will abort the nearest enclosing computation you typed in for evaluation at the top level, in a break, in USEREXEC, or with the editor's atomic E command.

QLISP includes a variable you can use to distinguish between its variants. The variable you use is QLISPSYSTEMTYPE; its value is VANILLA in the standard QLISP and otherwise is some adjective appropriate to the variant. Its only intent is to help you gracefully handle transitions between important changes in the behavior of QLISP. With any new version, the new value of QLISPSYSTEMTYPE will be announced; after the transition, the value will become VANILLA in the new version. For example, you might wish to code a program to use one strategy when one version of the pattern matcher is present but to use a different one with different version.

Any QLAMBDA function (that is, any atom whose function definition cell contains a QLAMBDA function) has its QLAMBDA pattern stored under the property QLISP:QLAMBDAPATTERN. This is true whether the function is compiled or interpreted; qtracing uses this fact.

Appendices

A. An Example

With the goal of creating a better world through computer science, we present a small system for making people happy. It was written not with elegance or efficiency in mind but to give examples of the features of QLISP and their interactions with INTERLISP. This example has been criticized on a more human plane for using values that are not universally shared; I persist in their use because I expect them to be readily understood even by those not holding them. The example is perhaps most appropriately taken in the same vein as the deductions typically presented in the exposition of a novel mathematical system: well-understood results are deduced, but the purpose is not to establish the results but to establish the system by showing that (and how) it leads to them.

1. The GENESIS system

This symbolic file was generated by using MAKEFILE in the ordinary manner.

```
(FILECREATED " 2-DEC-74 20:47:40" QGENESIS.QLISP;2 6134

    previous date: "26-NOV-74 20:43:09" QGENESIS.QLISP;1)


  (LISPXPRINT (QUOTE QGENESISCOMS)
              T T)
  (RPAQQ QGENESISCOMS ((FNS * QGENESISFNS)
          (QVARS $MARRIAGEDEMONS)
          (QVARS $COMPUTERELATIONS)
          (DECLARE: DOEVAL@LOAD DOEVAL@COMPILE DOCOPY
                    (P (DEFTYPE (QUOTE MARRIED)
                                (QUOTE CLASS))))
          (QLISP:SAVEFILEINFO)))
  (RPAQQ QGENESISFNS (SETUP MAKEHAPPY HITCH CHECKAGE CHECKHOBBY
                            PARTNERSEX RICH MAKESPOUSE))
(DEFINEQ
```

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

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

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

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

(CHECKHOBBY
  (QLAMBDA (MARRIED _X
                    _Y)                                (* FIND AT LEAST ONE
                                                       HOBBY IN COMMON,
                                                       OTHERWISE FAIL.)
           (ATTEMPT (MATCHQQ (TUPLE (CLASS _H
                                           __OTHERS)
                                    (CLASS _H
                                           __OTHEROTHERS))
                             (TUPLE (@(QGET (PERSON $X)
                                            HOBBIES))
                                    (@(QGET (PERSON $Y)
                                            HOBBIES))))
             ELSE (FAIL CALLER))))

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

```
(RICH
  (QLAMBDA (HAPPY _HUMAN)                                    (* TRY TO ACHIEVE A NET
                                                             WORTH GREATER THAN ONE
                                                             MILLION.)
                                                             (* IF ACHIEVABLE, THEN
                                                             THE HUMAN IS HAPPY.)
                                                             (* THIS ROUTINE NOW ONLY
                                                             MAKES A SIMPLE CHECK
                                                             AGAINST THE DATA BASE.)
           (PROG (NETWORTH)
                 (SETQ NETWORTH (QGET (PERSON $HUMAN)
                                      NETWORTH))
                 (IF (GREATERP (IF (EQ NETWORTH (QUOTE NOSUCHPROPERTY))
                                   THEN 0
                                 ELSE NETWORTH)
                               100000)
                     THEN ('(HAPPY $HUMAN))
                   ELSE (FAIL)))))

(MAKESPOUSE
  (QLAMBDA (PERSON _PERSON)                                  (* TEAM MEMBER OF
                                                             $COMPUTERELATIONS.)
                                                             (* ENSURES THAT THE
                                                             SPOUSE IS NOT ALREADY
                                                             MARRIED.)
                                                             (* ASSERTS THAT THE
                                                             SPOUSE IS MARRIED.)
           (QPROG ((_SPOUSE
                     (QGET (PERSON $PERSON)
                           MARRIEDTO))
                   SPOUSESPOUSE)
                  (SETQ SPOUSESPOUSE (QGET (PERSON $SPOUSE)
                                           MARRIEDTO))
                  (IF (NOT (OR (EQ SPOUSESPOUSE (QUOTE NOSUCHPROPERTY))
                               (EQ SPOUSESPOUSE $PERSON)))
                      THEN (FAIL CALLER)
                    ELSE (QPUT (PERSON $SPOUSE)
                               MARRIEDTO $PERSON)))))
)
  (MATCHQQ _MARRIAGEDEMONS (CHECKAGE CHECKHOBBY)
           WRT $QLISP:FILECTX)
  (MATCHQQ _COMPUTERELATIONS (MAKESPOUSE)
           WRT $QLISP:FILECTX)
[DECLARE: DOEVAL@LOAD DOEVAL@COMPILE DOCOPY
  (DEFTYPE (QUOTE MARRIED)
           (QUOTE CLASS))
]
  (QLISP:LOADCOMS (QUOTE ((HITCH (HAPPY _HUMAN))
                          (CHECKAGE (MARRIED __COUPLE))
                          (CHECKHOBBY (MARRIED _X _Y))
                          (PARTNERSEX _X)
                          (RICH (HAPPY _HUMAN))
                          (MAKESPOUSE (PERSON _PERSON)))))
```

```
[DECLARE: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVARS
  (ADDTOVAR LAMS HITCH CHECKAGE CHECKHOBBY PARTNERSEX RICH MAKESPOUSE)
]
(DECLARE: DONTCOPY
  (FILEMAP (NIL (512 5522 (SETUP 524 . 1162) (MAKEHAPPY 1166 . 1602)
(HITCH 1606 . 2421) (CHECKAGE 2425 . 3027) (CHECKHOBBY 3031 . 3495)
(PARTNERSEX 3499 . 3830) (RICH 3834 . 4672) (MAKESPOUSE 4676 . 5519))))
)
STOP
```

2. A Sample GENESIS Run

Here is a session at the teletype illustrating the use and operation of the GENESIS system.

```
@qlisp


INTERLISP-10  18-NOV-74 ...
QLISP-10  14-JAN-75 09:03:26
(<SUBSYS>QLISP.SAV;9 . <SUBSYS>LISP.SAV;21)


Hello, Mike.
T
2_load(qgenesis.com)
COMPILED ON  4-DEC-74 13:26:16
FILE CREATED  2-DEC-74 20:47:40
QGENESISCOMS
QGENESIS.COM;1
3_$marriagedemons
(CHECKAGE CHECKHOBBY)
4_pp hitch
loading from QGENESIS.QLISP;2

(HITCH
  (QLAMBDA (HAPPY _HUMAN)  **COMMENT**     **COMMENT**     **COMMENT**
           (IS (PERSON _Y)
               SEX
               (@(PARTNERSEX (' (PERSON $HUMAN))))
               THEN (ASSERT (MARRIED $HUMAN $Y)
                            APPLY $MARRIAGEDEMONS)
                    (QPUT (PERSON $HUMAN)
                          MARRIEDTO $Y WRT GLOBAL APPLY
                          $COMPUTERELATIONS))
           (' (HAPPY $HUMAN))))
HITCH
5_setup)
(PERSON SARA)
```

```
6_makehappy((adam sara))
HITCH + 1:
#0= (HAPPY ADAM)
  PARTNERSEX + 2:
  #0= (PERSON ADAM)
  (PARTNERSEX) - 2 = FEMALE
  CHECKAGE + 2:
  #0= (MARRIED ADAM MARY)
  (CHECKAGE) - 2 = OK
  CHECKHOBBY + 2:
  #0= (MARRIED ADAM MARY)
  CHECKAGE + 2:
  #0= (MARRIED ADAM ALICE)
  CHECKAGE + 2:
  #0= (MARRIED ADAM EVE)
  (CHECKAGE) - 2 = OK
  CHECKHOBBY + 2:
  #0= (MARRIED ADAM EVE)

GC: 8
5221, 10331 FREE WORDS
  (CHECKHOBBY) - 2 = ((CLASS SNAKE-CHARMING GARDENING VOLLEYBALL)
(CLASS GARDENING HUNTING FISHING))
  MAKESPOUSE + 2:
  #0= (PERSON ADAM)
  (MAKESPOUSE) - 2 = (PERSON EVE)
(HITCH) - 1 = (HAPPY ADAM)
(HAPPY ADAM)
HITCH + 1:
#0= (HAPPY SARA)
  PARTNERSEX + 2:
  #0= (PERSON SARA)
  (PARTNERSEX) - 2 = MALE
  CHECKAGE + 2:
  #0= (MARRIED ADAM SARA)
RICH + 1:
#0= (HAPPY SARA)
(RICH) - 1 = (HAPPY SARA)
(HAPPY SARA)
FINISHED
7_
```

B. Summaries

1. Prefixes

A QLISP variable is a literal atom whose print name starts with a special prefix. This prefix may be a dollar sign ($) or a left arrow (_), and it may be doubled (giving $$ or __). The dollar prefixes cause a variable to produce its value upon instantiation. On the other hand, an arrow variable instantiates to itself but will accept a new value during the unification operation. The double prefixes cause a variable to produce (or accept) a fragment instead of a whole compound datum, such as a bag. Here, then, are the recognized forms of a QLISP variable:

$X, which produces an element;
_X, which accepts an element;
$$X, which produces a fragment; and
__X, which accepts a fragment.

2. Operators

The two unary operators used to force instantiation or evaluation of an expression when it occurs in a construct that would normally cause the other are:

@, which forces evaluation; and
', which forces instantiation.

A unary operator that forces repeated instantiation of its operand until no further changes will take place is

", the superinstantiation operator.

However, a QLISP variable will be replaced by its value regardless of its prefix, and expressions governed by @ or ' will be evaluated or instantiated, respectively.

Fragments can be produced from QLISP variables by giving them the $$ prefix; a unary operator producing a fragment from an arbitrary expression is

!, the strip operator.

The four n-ary operators used to produce aggregates of data are:

TUPLE, which produces a tuple;
VECTOR, which produces a VECTOR;
BAG, which produces a BAG (or multiset); and
CLASS, which produces a CLASS (or set).

Each aggregate datum is represented to LISP as a list of a particular format: a tuple as a list of its elements; and each of the others as a list headed by its type name (e.g., BAG) and then containing its elements. A tuple evaluates like a LISP list, and each of the others evaluates to its instantiation.

A context datum is produced by the context designator, which takes one of the forms:

(CONTEXT ctxn), which produces the indicated context; or

(CONTEXT dir ctxd), which produces a neighboring context.

Both forms instantiate their operands. In the first form, the ctxn operand tells which context to produce and can take one of the forms:

LOCAL -- the current (local) context;
GLOBAL -- the top (global) context;
ETERNAL -- LOCAL, but non-backtrackable; or
UNIVERSAL -- GLOBAL, but non-backtrackable.

In the second form, the ctxd operand can be:

one of the four "context names" just mentioned; or
a context datum resulting from some previous use of a context designator.

The dir operand in the second form tells which neighbor of the indicated context to produce. It can be either:

POP -- its ancestor; or
PUSH -- a new descendant.

3. Default Contexts

When QLISP needs to default a context, this is how it behaves:

MATCHQ's and instantiation of a variable refer to the variable's nearest enclosing binding, as generated by QPROG, QLAMBDA or QDECLARE; while
property list access happens in the context of the nearest enclosing recommendation.

Moreover, the global context is treated as if it enclosed all else.

4. Statements

This list schematically shows the formats of the various QLISP statements. The notation used is intended to be suggestive without being unnecessarily rigorous; these abbreviations are used in the schematic statements:

(The first five are positional.)

eexp is an expression to be evaluated;
iexp is an expression to be instantiated;
exp is an expression handled in some way idiosyncratic to the statement in question;
val is an expression to be evaluated;
var is an arrow variable;

(The other seven are parsed by context.)

ind instantiates to an indicator;
inds/props is a sequence of expressions, paired by alternation, of which the first, third, etc. instantiate to indicators and the others to properties;

WRT-clause takes one parameter;
APPLY-team takes several parameters;
NAME-part takes one parameter;
THEN-part takes several parameters; and
ELSE-part takes several parameters.

Some parts of the statements are required, and some [enclosed in square brackets] are optional.

```
(ASSERT     iexp     [inds/props]     [WRT-clause]
     [APPLY-team] [NAME-part])
(ATTEMPT val [THEN-part] [ELSE-part])
(CASES iexp [WRT-clause] [APPLY-team] [NAME-part])
(DELETE     iexp     [inds/props]     [WRT-clause]
     [APPLY-team] [NAME-part])
(DENY  iexp [inds/props] [WRT-clause] [APPLY-team]
     [NAME-part])
(FAIL [iexp])
(GOAL iexp [inds/props] [WRT-clause]  [APPLY-team]
     [NAME-part])
(INSTANCES    iexp    [inds/props]    [WRT-clause]
     [APPLY-team] [NAME-part])
(IS iexp  [inds/props]  [WRT-clause]  [APPLY-team]
     [NAME-part] [THEN-part])
(ISNT iexp [inds/props] [WRT-clause]  [APPLY-team]
     [NAME-part])
(MATCH eexp1 eexp2 [WRT-clause])
(MATCHQ iexp1 eexp2 [WRT-clause])
(MATCHQQ iexp1 iexp2 [WRT-clause])
(QDECLARE var WRT-clause)
(QDO iexp WRT-clause)
(QGET   iexp   [inds/props]   [ind]   [WRT-clause]
     [APPLY-team] [NAME-part])
QLAMBDA {format is like LAMBDA}
QPROG {format is like PROG}
(QPUT  iexp  inds/props  [WRT-clause] [APPLY-team]
     [NAME-part])
(QRETURN iexp)
(UNBOUNDP exp)
(VAL exp WRT-clause)
```

5. Flags and Their Friends

Here is a list of the various flags and other variables you can use to communicate with the QLISP system. Some let you tell it how to behave. It uses others to record its own internal data in a format that is easy to interpret. Most of these variables are ordinary LISP variables; the only QLISP variable has a dollar prefix to remind you.

QA4:TRACEDFNS is a list of all the functions that are currently qtraced.

$QLISP:FILECTX is used in a WRT-clause when the value of a QLISP variable is read from a file or written on one.

QLISP:HASHTRANSARRAY maps objects of the QLISP extended data

types back to their internal forms.

QLISP:HASHTRANSFLG controls the storing of links in QLISP:HASHTRANSARRAY.

QLISP:SELFCONTAINEDFILESFLG tells PRETTYDEF how strongly files should be connected to QLISP. It can be overridden for an individual file by giving the QLISP:FILE property to the file.

QLISP:TRACERESPONDFLG controls whether the qtrace printout shows internal forms of QLISP data.

QLISPIFYCOMPFLG tells the compiler whether to expect QLISP in functions it compiles. Individual files can be treated differently if they have the QLISP:FILE property.

QTRACEALL controls the automatic qtracing of QLAMBDA functions.

QTRACECOUNTFLG controls the inclusion of depth counts in the qtrace printout.

Bibliographical References

This list cites those works to which this manual alludes, by title or otherwise.

Johns F. Rulifson, Jan A. Derksen, and Richard J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning", Stanford Research Institute, Menlo Park, Calif., SRI AIC Technical Note 73 (1972).

Warren Teitelman, "INTERLISP Reference Manual", Xerox Corporation, Palo Alto (1975).