Experience With an Uncommon Lisp

Cyril N. Alberga, Chris Bosman-Clark, Martin Mikelsons, and Mary S. Van Deusen
IBM T. J. Watson Research Center
Computer Science Department
Yorktown Heights, New York 10598

Julian Padget
University of Bath
School of Mathematics
Bath, Avon BA2 7AY, England

Abstract

In 1974, a group at IBM Research began work on a new implementation of Lisp. Because the work was initially done for internal use only, many design decisions led the developers away from the more traditional Lisp paths. The most important design decision was to create a language which would have consistent semantics between compilation and interpretation. This paper takes a retrospective look at the decisions we made to see how they have stood up against the test of time and usage, especially when seen against the decisions made for Common Lisp.

The Lisp language issues discussed include scoping, operator evaluation and consistency, and state saving. Compiler issues include the integration of assembly code into Lisp programs, our underlying formal semantics, and optimization. Our programming environment is as sophisticated as the majority of Lisp systems. Aiming at hardware such as the IBM 3270 terminals has affected the design of the tools that make up this environment. These issues will also be discussed.

Introduction

In 1974, a group at IBM Research began work on a new implementation of Lisp. What they produced became widely used within the company for internal research applications, and their work was the base on which future Lisp systems were built. The current Program Offering, LISP/VM, was a snapshot of that system in 1984. Because our Lisp system was originally intended only for internal use, some of our goals differed from those common in the wider Lisp community. Along with differing goals have come differing solutions to recognized problems.

The value of our system is found in the success of the applications running on it. Many, such as Alan Cobham's bridge program, run directly. A description of this expert system can be found in Alan Truscott's column on bridge in the *New York Times* [Truscott 1984]. Other applications, such as YES/MVS [Griesmer *et al* 1984], Epistle [Heidorn *et al* 1982] [Richardson 1985], and Scratchpad [Davenport 1985], are written in application languages which run on top of our system.

Since we are a research group, we are interested in an appropriate research vehicle, and also in making that vehicle reflect current thinking in language theory. Before we begin any redesign, we are taking this opportunity to examine our current design in the light of its success in achieving our goals, and in the shadow of other efforts which have found solutions to many of the same problems. This paper reflects that retrospective investigation.

Our language design was driven mainly by the goal of creating semantics which would be consistent under both interpretation and compilation. This was a goal we were able, for the most part, to meet. Our system had to compile efficiently, but an almost more important goal was that the semantics should be consistent, elegant and predictable.

The environment was severely constrained by display hardware (24 by 80 character monochrome terminal) but, with that constraint, our goal was to produce a user-friendly system. The dynamic display of the structure editor and the integrated debugger were our solutions.

Overall, we took the point of view that the features that set Lisp apart from other languages must be protected from erosion by premature performance concerns. In several cases, if we could not design a consistent facility, we backed off to a simpler design. But in any important area, we would not compromise.

Language Issues

At the time we began our design, there were many exciting ideas being discussed in the Lisp community. As a research group, we felt relatively free to innovate, and did not feel that we had to tie ourselves to compatibility with any other dialects of Lisp. This gave us the opportunity to actually try out many of these new ideas, concentrating instead on internal consistency of language and implementation. We wanted the semantics to be consistent and predictable, so that they could

be easily described, and so that reasonable user expectations would be satisfied. We wanted the meaning of an expression to be clear without reference to subtle provisos or special cases.

This emphasis extended to many facets of the system. We wanted above all to eliminate the distinction between interpreted and compiled evaluation. The evaluation rules for operators were made to be the same as for arguments, and were applied in all cases. Most functions were extended to behave reasonably in the presence of substructure sharing and cycles. Variable scoping in compiled and interpreted code was made equivalent.

In some of these endeavors, we feel that we succeeded and that the system is stronger for it. In others, some compromise was found to be required in order that other equally desirable goals might be met.

Operator Evaluation

An early decision was to abandon the tradition of the function value cell. This decision was influenced by the early Susman and Steele papers defining SCHEME [Sussman and Steele 1975] [Steele 1976] [Steele 1977]. It was thought that this evaluation model would offer a cleaner and more easily understood meaning for expressions. It also was seen as allowing great flexibility in defining and redefining operators, particularly in light of the uniform scoping rules for compiled and interpreted code, and the facilities for creating multiple environments. Our final design went further than other SCHEME derived dialects in a number of ways.

Expressions are initially classified as special forms, macros, and anything else. The first two cases involve the processing of the unevaluated form, while the last is assumed to be a function application. In that final case, the remainder of the form is treated as a list of expressions, the arguments, which are sequentially evaluated in a fixed order of evaluation. Only after the arguments have been processed is the operator examined to determine its exact nature. If it is prima facie applicable (i.e. a compiled function, explicit LAMBDA expression, funarg, partial application or state descriptor), the requisite action is carried out. If it is a non-applicable constant (e.g. a number or string), an error is signaled. If it is any other expression (an identifier or a list), it is repeatedly reevaluated until an applicable object or a constant is obtained.

This allows such coding styles as the use of an array-accessing expression as an operator, where the array in turn contains identifiers (symbols) which will finally evaluate to applicable objects.

A second way in which we differ from other Lisp dialects is in always evaluating the CAR of a form before classifying it. This means that, unlike other dialects (whether using function value cells or not), the meaning of an operator is never inherent in its identity within a form. Suppose the interpreter encounters a form such as:

```
(COND ((NUMBERP X) (+ 2 X)) ... )
```

This is not a Lisp conditional statement unless the value of the identifier COND is (the special form) COND. Conversely, a form such as

```
((FOO BAR) ((NUMBERP X) (+ 2 X)) ... )
```

will be recognized as a Lisp conditional only if that expression evaluates to the special form COND. In particular, the form:

```
('COND ((NUMBERP X) (+ 2 X)) ... )
```

will be so recognized.

In a newly generated system, most (but not all) special forms are bound to themselves as their value. Unfortunately we failed to make the final step and separate special forms and builtin functions from ordinary identifiers. Where we should have defined an object, say :SETQ (to use the Common Lisp [Steele 1985] notation for keywords), which always evaluates to itself, we simply have SETQ, distinguished as a special form, but usable as a variable. Thus if one wishes to write a SETQ expression which is immune to redefinitions, one must use (QUOTE SETQ) as the operator.

We extended our SETQ operator to allow the same destructuring possible in our LAMBDA variable lists. In implementing our definition of SETQ, which subsumes the Franz LISP DESETQ operator [Wilensky 1984], we define the value of the identifier SETQ to be a compiled macro which produces de-structuring code in terms of the primitive special form SETQ. The operators in the resulting macro expansion are QUOTEd, to preclude further invocations of the macro.

One must be careful in the fully redefinable world not to lose the ability to quote expressions. If some value other than QUOTE is assigned to the identifier QUOTE, then unless the special form QUOTE has been saved as the value of some other identifier, the system will be irreparably damaged.

This redefinability at the most primitive level was quickly put to use. For example, in constructing a compatibility package to allow the execution of MACLISP programs, the differing ways of declaring dynamic variables was accommodated by defining LAMBDA as a macro. At the same time, functions are defined as the values of SPECIAL and UNSPECIAL. These receive unevaluated arguments and maintain a list of the currently declared special variables in a free variable accessible by the LAMBDA macro. When a LAMBDA expression is applied or compiled in the presence of this redefinition, the macro examines the bound variable list of its form, together with the list maintained by the functions SPECIAL and UNSPECIAL. Based on the occurrence or nonoccurrence in the global list, variables may be replaced by the (FLUID variable) form which we require.

Once we had built our "always evaluate the operator" system, we discovered that full redefinability requires a more complex set of rules. In the following example, by assigning FOO the value MLAMBDA, we expect FOO to behave as the macro expression marker.

```
(SETQ FOO 'MLAMBDA)
(SETQ BAR '(FOO (() . BODY) (CONS 'PRINT BODY)))
(BAR X)
```

But this does not follow from a naive operator evaluation scheme. The sequence of events in evaluating (BAR X) is:

```
BAR evaluates to (FOO ... )
   -- this value is not apparent as a macro
X is evaluated
   -- this is the evaluation of arguments in
      the case where the operator is assumed
      to be a function
(FOO ... ) evaluates now to a macro funarg
   -- we now signal an error since we notice
      the macro after the arguments have been
      evaluated
```

To resolve this dilemma we had to refine the operator classification process. Simply put, if the value of an operator is a list, its CAR (the operator of the operator) is further examined. If it also is a list, we recursively continue this process. If it is an identifier, it is evaluated in turn and the original operator is replaced by the new value (by CONSing, not by RPLACA). Once this procedure has been completed, the final result is examined to determine if the original form is a macro, a special form, etc.

To return to the previous example, the sequence of operations is:

```
BAR evaluates to   (FOO (() . BODY) ... )
FOO evaluates to   MLAMBDA
   -- causing the operator to be rebuilt
operator becomes   (MLAMBDA (() . BODY) ... )
   -- and is seen to be a macro
form becomes       (PRINT X)
   -- and evaluation starts over
```

While we would prefer a simpler operator evaluator scheme, the advantages of total redefinability lead us to favor this approach.

Operator Consistency

In many Lisps, the implementation of operators such as COPY, READ and PRINT have to deal with tree-walking in the presence of circular structures. Because the identification of circularity can be expensive (sometimes quadratic), it is common to implement the operators to not terminate in the presence of circularity. If the structure might be circular, special operators are often provided which are understood to be expensive, although correct.

One of the design goals of the language has been universality of operators, that is, trying to define operators to work consistently across types. Because of that we chose to implement only a single set of operators, using tree-walking algorithms which were as efficient as we could make them. Our algorithms turned out to be linear. We were able to accomplish this because of our alternate heap which guarantees the presence of the working storage needed by linear algorithms.

SHAREDITEM is a function which makes the linear techniques available to the user. It does this by extracting the shared pointers from a circular structure.

It is important to decide when to stop walking a tree. We must define which nodes are terminal or atomic. We identify objects as atomic if we choose not to descend through their structure. Atomic objects include identifiers, state descriptors, binary program objects, hash tables and read tables. We saw a straightforward way to provide the user with useful information on funargs, and so did not make them atomic. We were not able to come up with ways to do this for hash tables or read tables, although we would have preferred to do so.

Equality

We used terminating, linear algorithms, similar to the operators in the previous section, to implement our equality predicates. One consequence of this approach is that our equality predicates terminate in all cases. Another consequence is that, since the tree-walking algorithms detect shared sub-structure, our equality predicates can be sensitive to the subtle effects of sharing. The operators which we distinguish are:

EQ	Traditional pointer identity
UEQUAL	Structural isomorphism
UGEQUAL	Relaxation of structural isomorphism to account for gensyms (in effect, unification of gensyms).
EQUAL	Access equivalence which finds two objects to be EQUAL if they are indistinguishable by any sequence of access operations

We can illustrate some of these distinctions more easily with a few examples. Consider the following LISP program.

```
(SETQ X (CONS 1 2))
(SETQ Y1 (CONS X X))
(SETQ Y (CONS 1 2))
(SETQ Y2 (CONS Y Y))
(SETQ Y3 (CONS (CONS 1 2) (CONS 1 2)))
```

What can we say about the equality of Y1, Y2 and Y3? Clearly none of them are EQ since they are the results of distinct CONSes. But there is one sense they are all equal: any sequence of CAR and CDR applied to Y1, Y2 or Y3 will yield the same result. This is the kind of equality tested for by our EQUAL predicate.

Let us now consider the following expressions and their results.

```
(RPLACA (CAR Y1) 3) = ((3 . 2) . (3 . 2))
(RPLACA (CAR Y2) 3) = ((3 . 2) . (3 . 2))
(RPLACA (CAR Y3) 3) = ((3 . 2) . (1 . 2))
```

We see that the three RPLACA expressions have identical effects on Y1 and Y2, but a different effect on Y3. How are Y1 and Y2 equivalent, but different from Y3? We can see that the representations of Y1 and Y2 consist of identical patterns of CONS cells and pointers. Both Y1 and Y2 consist of one CONS cell in which both the CAR and CDR components point to the same component. But the CONS cell in Y3 points to two distinct components. Our UEQUAL predicate tests for this kind of equality. This is also the kind of equality we use when hashing data structures.

Since our Lisp reader renames gensyms on input, the same expression read twice in a row will yield two expressions with distinct gensyms. When comparing such expressions, it is often useful to know if two expressions came from the same input form. The UGEQUAL predicate serves this purpose.

Although we paid great attention to the distinguishing of structure equality, we neglected the opportunity to differentiate numeric equality as Common Lisp and many other Lisps do.

Scoping

Variable Scoping

Compiler and interpreter semantics have traditionally diverged in the area of variable scoping. Lisp 1.5 and descendents such as MACLISP used pure dynamic scoping in the interpreter and a form of static scoping, modified by dynamic (or special) variables, in the compiler. Attention was drawn to this issue by Steele and Sussman who, having identified the adoption of dynamic binding as being a major contributor to errors, proposed a purely lexically scoped dialect of Lisp called SCHEME [Sussman and Steele 1975].

The initial design of LISP/370 [IBM 1978] recognized that both compiler and interpreter semantics should obey the same rules with respect to the scoping of variables. The rules we chose provided lexical scoping as a default, with dynamic (FLUID variables) as an option. Providing lexical scoping as a default allows users to choose to program without unexpected side-effects. Adding dynamic variables as an option allows users to choose power at the expense of safety. Common Lisp adopted a solution similar to ours.

We also discovered situations in which it was valuable to extend our scope rules. An application is commonly built using default interactive error handlers. When this application is sufficiently stable to use in a production environment, we no longer want those error handlers to take control because the users of this application will not understand what actions to take. We created a new error handler which could override all others, thereby providing automatically-taken, predefined actions. Since this type of an override should not be masked by any kind of environment switching during execution of the application (e.g., funargs or state saving), we extended scoping in the late 70's to allow the evaluation of dynamic variables using only the control chain.

Scoping Primitives

In early Lisps, the functional form identified by LAMBDA was the only way of introducing variables. Later, PROG was added to the language to allow local variables, labels, and transfer of control via GO.

We tried to refine these concepts in LISP/370 by introducing the special form SEQ to define the context for labels and the scope for GO expressions. We retained LAMBDA (and the macro variant MLAMBDA) as the form that defines the scope of variables and the scope for RETURN expressions.

The Common Lisp definition has taken this separation one step further by introducing the BLOCK special form as the scope for RETURN expressions and leaving LAMBDA to be a pure variable scope.

Our experience has clearly demonstrated to us the value of the three scoping primitives in Common Lisp. In our definition, any macro that emits variable bindings must announce that fact to the programmer, since any use of that macro becomes implicitly a scope for user RETURN expressions. This situation complicates the descriptions of many commonly used macros, and discourages the creation of new macros that need to introduce additional variables.

Formal Parameter Declarations

From the very beginning, we associated any declarative information directly with the scope in which the variable is bound. Thus, in the following expression

```
(LAMBDA (X (FLUID Y)) (FOO X))
```

the variable X is bound as a lexical variable and the variable Y is bound as a dynamic variable. With this notation, each variable is specified only once, and information about each variable is close to the variable name, even in long parameter lists. The result is an elegant, easy to write and easy to read notation.

Common Lisp achieves the same semantics with the more awkward syntax

```
(LAMBDA (X Y) (DECLARE Y SPECIAL) (FOO X))
```

We have two objections to this notation. First, the variable is mentioned and declared in two different places. Second, each declaration may be the result of a macro expansion. This may result in programs that are difficult to understand, not only because declarative information may be hidden in obscure macros, but also because of the more subtle semantic problems caused by exposing the interpreter to arbitrary side-effects while it is still discovering critical information about an expression. It is mainly for this reason that we restrict our declarations to a part of the LAMBDA expression which is not macro-expandable.

Our notation may also be extended to include type information in the formal parameter list. For example,

```
(LAMBDA (X (THE PAIR Y)) (RPLACA Y X))
```

could be used to declare the type of the variable Y.

Pros and Cons of State Saving

When our development process began, there was ferment in the Lisp community. One of the more important events relating to the design of the new IBM Research Lisp system was the publication of the Bobrow and Wegbreit paper [Bobrow and Wegbreit 1973] proposing a new control structure, the spaghetti stack. We immediately seized on the elegance and flexibility of this idea.

A key component of this model of computation was the concept of the saved state, a data object which captured both the set of current variable bindings (the environment) and the current call chain and point of execution (the control). We have used saved states (state descriptors) extensively, both as a form of continuation and as a component of closures.

When State Saving Wins

The Yorktown Lisp compiler was designed to meet several goals. The overriding requirement was the maintenance of the equivalence of compiled and interpreted code, as discussed elsewhere in this paper. Other goals were:

- the protection of the compiler's inner state from the actions of arbitrary macro operators during their invocation, and
- provisions for collections of operator values distinct from those in the compiler's, or even the ultimate program's,

runtime environment, for the purposes of cross-compilation and optimization.

Saved states are used to provide the facilities that satisfy these latter goals during the compilation process. The strategy is to establish an isolation of the environment in which the compiler runs from those (separate) environments in which it resolves operators and in which it preforms macro expansions. By judicious construction and augmentation of these various environments, the user is given the ability to provide operator values to the compilation process which differ from those present in either the compiler's runtime environment or the macro expansion environment. This ability may be used for cross-compilation or for optimizations. In addition, it protects the compiler from interference by macros and the functions they invoke.

One of the major debugging tools we provide is an interactive interpreter, or stepper, described in a later section. We were able to write this interpreter entirely in Lisp, with no secret escapes into system internals, because saved states, evaluation in a saved state, and resumption of a saved state were all the primitives necessary to allow one Lisp program to intermix its evaluation with that of another Lisp program.

Another use of state saving is in exception handling in the top-level Lispedit read-loop. Rather than interrupting the user with an exception and requiring the user to give commands to continue execution, Lispedit saves the state in which the error occurs and returns immediately to the top-level, informing the user of the action which was taken. At this point, the user has the option of resuming the saved state and examining the error, or of simply continuing.

Spaghetti stacks have turned out to be an invaluable technique for the original development and debugging of back-tracking algorithms, using saved states. The section below discusses the problems with this approach for production work.

When State Saving Loses

State saving has not always proven to be beneficial. At times it has been a mixed blessing, or even the wrong way to do things.

In the compiler, the isolation of the saved states which embody the macro environments has made it impossible for macros to obtain the expanded versions of their forms. It has also made it difficult to pass information from one macro to another and to determine if they are being applied as part of the interpretive or the compile process.

Exception handling is a process with two main requirements: signals must be matched with handlers according to some scoping rule, and control must often bypass the normal call chain, passing directly back to some remote point in the computation. When control must bypass the call chain, as in ERRSET and UNWIND, a similar behavior is required. In the initial version of our system, exception handling and non-local returns (UNWIND) used saved states. That is, a program wishing to intercept a non-local return (e.g. ERRSET) would re-bind a specific dynamic variable to a saved state. The process wishing to return to such a point would evaluate that variable and apply its value to the datum to be returned.

This mechanism was found to be deficient in two ways. First, since the evaluations followed the environment chain, control could return to some remote point, unrelated to the current computation. Secondly, the number of stack frames immobilized by this constant state saving resulted in many more garbage collections than would have been needed otherwise.

The first problem was solved by the introduction of the control-chain evaluation primitives, the second by the addition of CATCH and THROW primitives and by their use for non-local returns.

Spaghetti stacks, while invaluable for prototyping, turn out to be too inefficient for production use. We have found the fastest way to program is to write backtracking algorithms using saved states, and then reimplement them using explicit routines such as CATCH and THROW. The inefficiency may be due to the dependence on the normal garbage collection process to free discarded state descriptors and their associated stack frames. If an explicit de-allocation operator were provided, another conclusion might have been reached. This would, of course, shift bookkeeping onto the shoulders of the programmer, which is contrary to the philosophy of Lisp.

Underlying Formal Semantics

The interpreter is unusual in that it is rule-based. It is a concrete implementation of an SECD machine [Landin 1964]. The basic semantics are specified by fewer than one hundred SECD rules. Before any part of the interpreter was written, numerous versions of the SECD rules were created, circulated and debugged on paper [Blair 1976]. Changes in the primitive semantics of the Yorktown Lisp system have always been worked out in the form of proposed changes to the SECD rules.

These rules are actually implemented by assembly language routines. The compiler implements the same semantics. Writing the compiler was made much easier by having the formal semantic description, as well as the interpreter source, as a guide. An appendix to this paper shows examples of SECD rules.

We found a gap in Landin's formalism. In order to fill this gap, we found we needed to add one more component to the SECD machine, the exit routine. This component records various actions, such as restoring shallow binding cells, which must be performed when control returns from a stack frame. The implementation of the SECD machine has not been extended to describe the actions of this component. For example, the semantics of CATCH and THROW must be described as special cases.

Machine Language Insertion

The code generated by our system is not assembly language, but rather an intermediate language known as LAP (the Lisp Assembler Program). LAP includes the complete System/370 instruction set plus extensions for testing types, performing Lisp function calls, accessing components of Lisp objects, manipulating the stack, etc. In addition, LAP is block-structured with bound and free variables declared at the start of a stack frame, or contour. LAP contains scoping rules for

both variable reference and transfers of control (GO and labels). Once defined, LAP became the target for the compiler and for macros which generate code. LAP is described further in the Implementation Issues Section.

We provide in our system a construct for including LAP code directly in a Lisp expression, the F*CODE construct. Having the convenience of this feature has enabled us to reduce the number of low-level functions needed, and to write most of the functions more readably in Lisp. In our system, only 150 functions have been written in LAP. All the rest are Lisp functions, with a small percentage of those taking advantage of the F*CODE.

The PNAME function is an example of a function which is a mix of Lisp and LAP. PNAME returns a copy of the print name of an identifier and, when we know we can safely manipulate the original print name, a single LAP instruction can be used to fetch the pointer to that name.

An F*CODE expression must be used as an operator. It contains two components, a Lisp expression and a list of LAP statements. When encountered by the interpreter, the Lisp expression is extracted and treated as if it had been the value of the operator. When encountered by the compiler, the argument evaluation is compiled, with the final argument value left in a specific register. Then the LAP statements are inserted into the code. The exclusive use of F*CODE expressions as operators requires that the Lisp expression be applicable, or that it evaluate to an applicable object.

An example of the use of F*CODE would be a macro definition for ASSQ. The form (ASSQ X Y) could be expanded to the following form:

```
( (F*CODE ASSQ ()
        "Load the first argument into a register"
        "*SCR2 points to what we are looking for"
    (L *SCR2 (TOP))
        "Move the second argument out of the "
        "     result register"
        "*S2 points to the current list tail"
    (LR *S2 *S1)
 LP     "If the tail is not a pair, its all over"
    (GOIFNOTR *S2 PAIR NG)
        "Pick up the CAR in *S1 and advance *S2"
    (LM *S1 (*S2 CAR) *S2)
        "If the CAR is not a pair, keep looking"
    (GOIFNOTR *S2 PAIR LP)
        "Look for the target in the CAAR"
    (CL *SCR2 (*S1 CAR))
    (TRA NE LP)
        "If found, exit with result in *S1"
    (TRA U OK)
 NG (LR *S1 *NIL)
 OK                ) X Y )
```

While it offers no guarantees, the syntactic proximity of the Lisp expression and the equivalent sequence of LAP statements does help to keep the programmer honest. A major deficiency in this construct is that there is no formal connection between the Lisp expression and the LAP code.

Another major problem has shown up in the use of register allocation. Originally, F*CODE was only used for machine code insertions and the use of actual register references was no problem. But once we realized that we could generate LAP through the expansion of macros, we found ourselves making heavy use of a facility that still required absolute registers. What we should have done is extended the notation of F*CODE to have allowed logical register references as well, but we didn't. The result of this decision is that we cannot get the advantages of global optimization from register reallocations.

Implementation Issues

If there's one myth heard more than any other about Lisp, it's that programs written in Lisp run so slowly that they can't be used for production work. We wanted a system which would give us the advantages of fast prototyping, but when the prototype is determined to be stable enough to be the production system, still be fast enough to let that program be used without change. It was to this aim that we determined that the semantics of interpretation and compilation had to be consistent. Not only that, compilation itself had to be fast.

As a research group, we were also intensely interested in consistent, elegant and predictable semantics for Lisp programs. In many compilers, when tradeoffs have to be made between fully defined semantics and compiler efficiency, the choice is in favor of compiler efficiency. We most often made that tradeoff the other way -- to preserve the intended semantics. Even so, we still find the final efficiency of our system extremely good.

The problems that we have found in our implementation stem mainly from two problems: locking ourselves into design decisions too quickly, and having to make many tradeoffs to achieve our goals of efficiency and consistency. The main tradeoffs came in the areas of flexibility and portability.

Execution Speed

The Yorktown Lisp system achieves the good execution speed of its compiled code by a number of techniques. The chief among these are a large number of special-casing code emitters, extensive analysis and consolidation of nested variable-binding constructs, and the use of the VM/SP Discontiguous Shared Segment. These, in turn, depend on various underlying features of the system.

Inline Code

The efficiency of our system is greatly enhanced by the use of built-in functions recognized directly by the compiler, and by the use of macros that emit the F*CODE forms described in the previous section.

Built-in functions (such as CAR, CONS, and PAIRP) have definitions which are hard-wired into the compiler and interpreter. When compiled, they expand to in-line code, avoiding all the overhead of function calls. The expansion of PAIRP, for example, results in the following LAP instructions,

```
(SECTION
   (GOIFR *S1 V INLABEL)
   (LR *S1 *NIL)
INLABEL)
```

where the value of the argument has already been left in register *S1.

A macro which generates F*CODE is expanded into very efficient inline code. Many of the F*CODE macros in the system are counterparts of function definitions, where the programmer chooses the macro or function version which meets the appropriate trade-off requirements. Some, like INTERLISP "fast" operators, don't check their arguments for validity. QCAR and QRPLACA, for example, assume that their first (or only) argument will be a pair. QSPLUS assumes its arguments will be small integers. Others, such QASSQ and QMEMQ, are safe but allow the programmer to choose between the bulk of in-line code or the overhead of a function call.

The compiler is able to take advantage of these built-in functions and F*CODE macros to do more extensive optimization in the context where they occur.

Grouping of Bindings

Because of our semantics, each LAMBDA expression must build at run-time a stack frame called a contour. The compiler wants to eliminate the overhead of individual stack frames and so, where possible, we attempt to optimize them out. The compiler analyzes nested LAMBDA expressions and, where it can be determined to be safe, raises the variables of a nested expression to the local variable list of an enclosing contour. At the point where a new stack frame should logically be created, the argument values are instead placed in the value cells of the current stack frame and the display is modified to make them visible to the binding search mechanism. The logical structure of nesting is retained, with the scoping rules for variables and labels fully enforced, but the overhead of stack frame construction elided.

We find that the vast majority of internal LAMBDA expressions, such as those generated by PROGs, DOs, MAPping operators, etc., are totally merged into the outermost function.

Shared Segment

The VM/SP operating system provides the ability to define a readonly block of virtual memory, the discontiguous shared segment (DCSS), outside a user's normal memory. The DCSS is shared among many users, and the chance of a page being in real storage when it's required is enhanced by multiple users being linked to the DCSS.

Our compiler and assembler have always produced read-only, address-free code. It was thus a relatively simple matter to place the bulk of the system code (the compiler, debugger, interpreter, editor, and the various utility routines) in a DCSS. This in turn has the paradoxical effect of making the probability of avoiding a page fault directly proportional to the number of users running Lisp at the moment.

Locking Ourselves In

An important tool in realizing the goal of compiler/interpreter equivalence was a stack frame which did not distinguish between the two situations. Early decisions, which in certain cases might have been made for efficiency reasons, have proven to be stumbling blocks to graceful extensions of the system.

Stack

The cost of maintaining the spaghetti stack as originally proposed by Bobrow and Wegbreit [Bobrow and Wegbreit 1973] was not fully appreciated. The allocation of components between the head-of-E (basic frame in their terminology) and the dump (frame extension) was taken directly from the original design. This results in a inextricable combination of the environment and the control components.

As part of our attempt to enforce compiler/interpreter equivalence, stack frames which are used by compiled code and by the interpreter have the same structure. Since frames used by compiled code must have a slot pointing to the compiled function object, a similar slot was added to all frames. Thus, interpreter frames have a pointer to a dummy compiled function. For the same reason, all stack frames contain a display component, mapping variable names to binding cells in the head-of-E. To compound the problem, we made the display a component of the compiled function, building it dynamically into the dummy compiled function for the interpreter frames.

The end result is that the control cannot be separated from the environment. All saved states must carry the weight of the dump portion of the stack. We could have moved the environment chain pointers from the dump, so that each head-of-E pointed directly to its predecessor. We could also have made the display an independent datum, also pointed to from the head-of-E. The combination of these two decisions would have allowed us to create "light" saved environments, usable only for evaluation.

It was known from the first that realization of the spaghetti stack would incur extra overhead at function call and return. To minimize this cost, the stack frames were carefully designed to allow their construction with the fewest possible machine instructions. As a result, no spare space is available in the stack frame structure. This, in turn, made certain changes to the system difficult or impractical.

We have often wished we could expand the display and include, for example, type information about variables. This has proven impractical, as every function and macro in the system would have to be recompiled in order to implement such a change. While we have made such drastic transitions in the past, they become more and more difficult in the presence of an ever-growing body of code.

Another case where an excessively economical design has made expansion of the stack frame difficult, was the retrofitting of CATCH and THROW as primitive operators. Because of the constraints of the stack structure, it was necessary to make rather unorthodox (and unanticipated) use of the exit field to record the presence of a CATCH-point and the alternative resume point in the code. The resume point is used to distinguish a THROW from a normal return.

Pointers and Types

Another area in which early design decisions led to a dead end is found in the structure of our pointers. Because of the nature of the target hardware, and in light of the thinking of the time,

a pointer containing twenty-four bits of address, with eight bits of type information, was used. While this often provided faster type tests, since memory references could often be avoided, it has prevented our moving easily to the new System/370 XA architecture.

The tagged pointer architecture of the system is very much like other Lisp systems, although there is an exceptionally wide range of types for a system running on non-specialized hardware. It is the tagged pointer architecture part of the system that reveals its age. It is not easy to introduce new types since they are hard-wired, whereas an object-oriented approach such as Flavors [Keene 1985] or CommonLOOPS [Bobrow *et al* 1985] would have provided for greater flexibility and extensibility. We intend to remedy this deficiency in the near future when it becomes clearer what the design fundamentals of OOPS should be.

Environments, Variable Evaluation, and Closures

The model of variable evaluation which we chose to implement was predicated on the structure of the spaghetti stack, and affected its detailed implementation. Rather than a single global environment, we support multiple quasi-global environments, referred to as non-LAMBDA environments. At any one instant, one of these is elevated to the status of current non-LAMBDA environment, and acts as a global name-value space for variables not found in the stack.

For further details, including the interaction between the environments and closures, the reader is directed to the appendix, "Appendix B. Environments and Closures".

LISP Assembler Program (LAP)

Although the compiler does a number of general optimizations, such as flattening and merging the contours which correspond to stack frames, the feature which has contributed most to efficiency is the ability of macro operators to specify machine instructions to be included in the generated code. This feature has both positive and negative aspects.

Effects of Designing Too Close to the Hardware

We recognized the desirability of an intermediate language in the creation of functions, such as long integer arithmetic, which must manipulate fragments of Lisp objects. It's also desirable for functions which require very high efficiency, such as COPY and EQUAL. A small measure of the success of this language, LAP, is found in the speed of the long integer arithmetic package.

While LAP makes the writing of operations (such as COPY, EQUAL and SHAREDITEMS) relatively easy, LAP is so close to the machine that it would significantly impede transfer of code to any other architecture.

The Artificially Limited Power of the Assembler

LAP exhibits weaknesses in two ways. On one hand, it is too close to the hardware. On the other hand, it is too close to Lisp. It is too close since the only programs LAP will assemble are those which correspond to a LAMBDA (or MLAMBDA) expression.

The effects of being too close to the hardware have been outlined previously. The effects of being too close to Lisp have caused a larger than desirable part of the system to be written in Assembler H, a non-Lisp System/370 assembler. This code includes the function-calling prologues and epilogues, the out-of-line routines for CATCH, THROW and other primitive operations, the binding search routine, the SECD machine interpreter and the garbage collector, to name only a few. These are pieces of code which must run in contexts different from that of a compiled function. Some are called (with linkages other than the normal Lisp to Lisp call) from compiled code, and must run without a stack frame and without disturbing the calling function. Others, such as the garbage collector, discard almost all the base registers that compiled code expects to find, throwing all conventions to the winds in an effort at maintaining as much data as possible in registers.

Programming Environment Issues

In our programming environment, we tried to combine many of the tools used by a programmer into an integrated package that is more useful than the components taken separately. We developed a few key ideas to their logical limits in order to provide a useful interface in spite of severe hardware and operating system limitations. We found that dynamic formatting and automatic condensation are necessary to make a structure editor tolerable. We also found that the same techniques were sufficient to make the structure editor an effective tool -- even in a command-oriented environment. The structure editor and the display manager interfaced naturally with a stepper/debugger in order to create a particularly useful debugging tool.

The Structure Editor

At first glance, the value of structure editors seems unquestionable when we must deal with complex but highly organized objects like programs. In the Lisp context, this observation applies equally well to data. But although structure editors exist in almost all Lisp systems, these editors are not used extensively.

We decided that this neglect was due to the fact that the user of most structure editors spends a significant fraction of the time figuring out where an editing operation will take place and what the effect of an editing operation was. Our solution to this problem was to present the user with a more dynamic view of the editing domain. By presenting a view that was automatically formatted and continuously updated, we freed the user to perform mostly editing operations.

Dynamic Display

The structure editor, Lispedit [Mikelsons 80], uses two techniques to generate useful and interesting displays: dynamic formatting and display condensation. A dynamically formatted display is one in which the screen always contains a prettyprinted form of the context in which the focus occurs. The focus is the current subexpression on which the editor is operating, that is, the user's finger in the program. Display condensation means the expression which is displayed can contain elided parts. What is displayed and what is elided are

chosen by a set of heuristics based on the position of the current focus. By the use of these two techniques, we find Lispedit to be as usable as a smart text editor.

Much of the work on the programming environment was done in parallel with research on a non-Lisp programming environment [Alberga *et al* 1984]. This research resulted in some very complex formatting and condensation techniques [Mikelsons 1981], which proved to be too expensive in CPU time and memory for the Lisp production environment. Instead, a collection of heuristics were developed based on trial and use.

The program display below illustrates many of these heuristics. Line numbers are for descriptive purposes only. The focus of attention on lines 6 through 16 is shown in almost full detail. The only condensation takes place on line 10, where a subexpression is abbreviated to the symbol '&'. Expressions preceding the focus in the body of the containing PROG are shown using one line for each. This forces the body of the DO on line 4 to be elided. All the expressions following the focus are elided since the focus has filled all the available space.

```
 1 (LAMBDA
 2   (INPUT)
 3   (PROG (WORDLIST)
 4     (DO ((I 0 (+ (FINDENDWORD INPUT I) 1))) ...)
 5     (SETQ WORDLIST (REVERSE WORDLIST))
 6     (NMAPCAR
 7       (LAMBDA
 8         (WORD)
 9         (COND
10           (((ONE-OF a e i o u) (ELT WORD 0)) &)
11           ( 'ELSE
12             (CONCAT
13               (SUBSTRING WORD 1 (- (SIZE WORD) 1))
14               (SUBSTRING WORD 0 1)
15               "ay "))))
16       WORDLIST) ...)
```

Figure 1. A Sample Program Display

Several numeric parameters can be set by the user to control the heuristics that format the screen. These numbers determine the maximum number of levels of nesting shown containing the focus, the maximum number of expressions shown preceding the focus, and the maximum number of lines in which to show the focus. These numbers are modified by special circumstances. For example, since the bound variables of the PROG expression are displayed on the same line as the PROG operator, that subexpression is not included in the count of expressions shown preceding the focus. Another special case is made for lists or vectors of one element. These are not included in the count of expressions shown containing the focus because they take very little screen space to show.

Our experience indicates that users may complain about the automatic choices made by the system, but are not willing, in general, to tailor interfaces unless very friendly tools are supplied to do the tailoring.

The key to speed in the display algorithm was to fill the screen, from top to bottom and from left to right, with no backing up and with only a small amount of looking ahead. One aspect of this was to always show a fixed depth of nesting around the focus. In many situations, this approach left a lot of blank screen which could have been filled effectively if a greater depth of nesting was shown. It turned out that in the cases where the screen is not filled, the display process is so fast that the solution was to repeat the display of that screen using a larger nesting parameter.

The use of a structure editor has a definite effect on the programming style that users develop. To many people, the form of a program can seem almost as important as the contents. A structure editor which continuously changes the indentation of what is seen, and changes it from the way in which it was originally typed, can be initially disconcerting. With experience, though, one stops relying on a particular indentation to yield context information (such as how deeply nested the current focus is within the enclosing expression) because the expression displayed by the structure editor will more likely be elided to show the context in which one is most interested.

A structure editor may also have an effect on the language definition and on the way the language is used. We have extended our definition of Lisp to allow more places where comments can exist as part of the Lisp structure. This means that when the program is read in by the structure editor, those comments will be retained, unlike comments which are only textually included.

To Copy or Not To Copy

The original design of Lispedit caused changes made to the expression on the screen to cause changes in the actual object (essentially, using RPLACA and RPLACD). This design is called editing-in-place. The design was chosen because it was felt to be an elegant approach to the problem. Unfortunately, it broke down at the top level expression. It was possible to lose contact at the top level with the object being edited, and from then on the effects of the editing may appear random and incoherent. For example, the object being edited may be a value of more than one identifier. When making changes below the top level, the values of both identifiers are changed. When making changes at the top level, it is necessary to make an assignment to the identifier in order to bring the identifier and the editor into synchronization again. The editor can automatically make the assignment to the identifier it knew it was editing, but it would be too difficult to find all the identifiers which also pointed to that value.

Our solution has been to back off from editing-in-place and to now edit copies as the default case. Editing in place must be requested by using the appropriate command. This means that we must provide operators to move the copy back to the environment. We have found this approach to be more comfortable because, although we have to do the assignments by hand, we are absolutely sure of the results and know that nothing is going on behind our backs.

Windows

Our user interface at the display terminal consists of a single command-oriented window. We realized at the outset that this put us outside the general Lisp community, but we felt that the advantages that we could offer our users would offset the incompatibilities.

Window management requires a certain minimum level of hardware support which the terminals commonly available within the IBM community did not have. The screen size was too small, there were no bit map displays, and it was not possible to get full-duplex communication between the terminal and the program. If we had chosen to implement on a non-standard terminal, we would have found ourselves isolated from this community. Because we were interested in developing a system for a large user base, we decided that we would have to stay within the available hardware. And that meant no windows.

Another related issue is that of allowing the user to type directly onto the displayed program text as well as in a predefined command area. In our more general structure editor [Alberga *et al* 1981], we devised the principle that if some new user text causes a well-formed program to become ill-formed, then the new program text should be treated as the cause of the error. Even though our half-duplex terminals allowed typing on any part of the screen, we decided against full-screen editing because in most cases it is not possible to determine which parts of the screen were written by the system and which were modified by the user. Unless this information is available to the editor on a character by character basis, it is not possible to apply the error principle consistently. As a result, we chose the command-oriented paradigm in order to provide a more consistent interface to the user.

Debugger: A Step Beyond Steppers

Debuggers are available with all Lisp systems but, like structure editors, they can be difficult to use. Before you can figure out what you want to do, you have to figure out how to get there. From its earliest implementation [Alberga *et al* 1981], our debugger, HEVAL, has offered the user the opportunity for true two-way communication with the program through the editor interface. In one direction, the debugger shows the next expression to be evaluated in the context of the containing expressions, that is, the expression is displayed as it would normally be through the structure editor. In the other direction, the user gives normal editor commands to HEVAL to guide the debugging. For example, we can point to an expression on the screen, make that expression the editor focus, and ask for evaluation to continue from that location.

Lieberman [Lieberman 1984] addressed the desirability of a stepper, or debugger, which could communicate back to the user through an editor, but didn't see the natural extension to two-way communication.

During development, we surprised ourselves with the general utility and ease of use of HEVAL. Contrary to our expectations and subjective evaluation, we found that twenty percent of all user commands invoke HEVAL. We also find that even devout anti-structure-editor users become enthusiastic toward this tool.

File System Integration

An attractive part of the system is the degree of integration with the host file system and the way in which that file system is accessed from the programming environment. The interface is provided through Lispedit by storing Lisp definitions in indexed files. Indexed files may be viewed conceptually as large (external) a-lists, keyed on the name of the definition. By using indexed files, our users need never interact with the external operating system.

The existence of the external file system is irrelevant to the user, and in this respect the environment approaches some of the ideals of persistent programming. The main criticism of indexed files stems from a more fundamental problem, the lack of flavors. If these "external a-lists" could be regarded as just another datatype by the system, and if there was a simple facility for providing a new method for those objects, then existing Lisp operators such as COPY, SETQ, and DELETE could be applied.

Conclusions

In closing, we must review our goals and measure our successes and failures.

In the language area, we wanted to define powerful and consistent semantics. Our operator evaluation model has proved to be precisely that. We have used the redefinability of fundamental operators on many occasions both to extend the language and to achieve compatibility with other dialects. In all cases, the extensions were done easily. We have used our general operators for equality and other utilities more extensively because of their uniform applicability and efficiency. We have come to trust interpreter-compiler equivalence so completely that we now take it for granted.

The power of state saving is irreplaceable when it is needed. But we definitely need to separate the concepts of saving an environment and the concept of saving a control state. The cost of saving both when only the environment is needed is too much of a deterrent to effective use.

In the implementation, the cost of several early decisions did not show until late in the life of the system. When extensions needed to be made, there were too many constraints. The availability of LAP and F*CODE was a big help, but at the cost of portability. Such a facility must be designed at a higher level in order to be most useful.

In the programming environment, the main lesson is that we need more. All the features have been well received by the user community. We also need to extend the facilities to modern full-duplex terminals and bit-mapped screens.

Over the years, Lisp has evolved in a community with very high expectations. It expects high expressive power in the language in order to address complex application areas. It expects high efficiency when these applications become successful and must cope with realistic quantities of data. It expects flexibility in the language and portability in the code. And last, but not least, it expects powerful and friendly user interface. Our experience has shown that these expectations are definitely worth living up to. Our main regret is that we compro-

mised some of these expectations in order to achieve some short-term implementation gains. In all the areas where we kept to a hard line, the effort and the cost were well worth it.

Acknowledgments

In twelve years, the IBM Research Lisp has been designed, built, influenced, and criticized by many. We would like to especially acknowledge the following people for their contributions: Mark Wegman (LISP/VM and YKTLISP); Alan Cobham, Barry Trager (Applications); Fred Blair, Alan Brown, James Davenport, Arthur Norman, Mark Pivovonsky, Richard Ryniker, Jon L. White (Lisp/370).

References

Alberga, C. N., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M., Wegman, M. N., "A Program Development Tool", *Eighths Annual ACM Symposium on POPL*, Williamsburg, Virginia, January 1981.

Alberga, C. N., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M., Wegman, M. N., "A Program Development Tool", *IBM Journal of Research and Development*, January 1984.

Alberga, Cyril N., *LISP Assembler Program: Reference Manual*, IBM Research Division report RA172, September 1985.

Blair, F.W., "The Definition of LISP1.8+0.3i", *IBM Internal Report*, 1976.

Bobrow, Daniel G., and Wegbreit, Ben, "A Model and Stack Implementation of Multiple Environments", *Communications ACM*, Vol. 16 Number 10, October 1973.

Bobrow, Daniel G., Kahn, K., Kiczales, G., Massinter, L., Stefik, M., and Zdybel, F., "COMMONLOOPS - Merging Common Lisp and Object-Oriented Programming", *Xerox Palo Alto Research Center Intelligent Systems Laboratory Series ISL-85-8*, August 1985.

Davenport, James H., "The LISP/VM Foundation of Scratchpad II", *The Scratchpad II Newsletter.*, Robert S. Sutor editor, IBM Corporation, Computer Algebra Group, Yorktown Heights, New York: Vol. 1, Num. 1 (September 1, 1985).

Griesmer, J.H., Hong, S.J., Karnaugh, M., Kastner, J.K., Schor, M.I., Ennis, R.L., Klein, D.A., Milliken, K.R., VanWoerkom, H.M., "YES/MVS: A Continuous Real Time Expert System", *Proceeding of the National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Los Altos, California.

Heidorn, G.E., Jensen, K., Miller, L.A., Byrd, R.J., and Chodorow, M.S., "The EPISTLE Text-Critiquing System", *IBM Systems Journal*, Vol. 21, Number 3, 1982.

Keene, Sonya E., and Moon, David A., "Flavors: Object-Oriented Programming on Symbolics Computers", Unpublished Paper, Common Lisp Conference, Boston, December 1985.

Landin, P. J., "The Mechanical Evaluation of Expressions", *Computer Journal*, Vol. 6, 1964.

Lieberman, Henry, "Steps Toward Better Debugging Tools for LISP", *1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984.

Mikelsons, Martin, *Lispedit Command Descriptions*, IBM Research Report RC8275, May 1980.

Mikelsons, Martin, "Prettyprinting in an Interactive Programming Environment", *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 1981.

Richardson, Stephen D., "Enhanced Text Critiquing using a Natural Language Parser", *Seventh International Conference on Computers and the Humanities*, Provo, Utah, June 1985.

Steele, Guy L., "LAMBDA: the Ultimate Imperative", *Memo No. 353, Artificial Intelligence Laboratory*, MIT, Cambridge, Massachusetts, March 1976.

Steele, Guy L., "Debunking the 'Expensive Procedure Call' Myth or, Procedure Call Implementations Considered Harmful or, Lambda: the Ultimate GOTO", *Memo No. 443, Artificial Intelligence Laboratory*, MIT, Cambridge, Massachusetts, October 1977.

Steele, Guy L., *Common LISP: The Language*, Digital Press, 1984.

Sussman, Gerald J. and Steele, Guy L., "SCHEME: an Interpreter for Extended Lambda Calculus", *Memo No. 349, Artificial Intelligence Laboratory*, MIT, Cambridge, Massachusetts, December 1975.

Truscott, Alan, *New York Times*, October 7, 1984.

Wilensky, Robert, *LISPcraft*, W. W. Norton & Company, New York, 1984, p.329.

White, Jon L., "LISP/370: A Short Technical Description of the Implementation", *ACM-SIGSAM Bulletin*, November 1978.

LISP/370 Program Description/Operations Manual, *SH20-2076*, IBM Corporation (March 1978); available through IBM branch offices.

LISP/VM User's Guide., *SH20-6477*, IBM Corporation (July 1984): available through IBM branch offices.

Appendix A. SECD Rules

The SECD machine is an abstract machine containing four components, the stack (*S*), the environment (*E*), the control (*C*) and the dump (*D*). The state of the machine at a given instant is shown by a diagram of the form:

{*S*; *E*; *C*; *D*}

The *S* and *C* are push-down stacks which can contain any data objects (not to be confused with the stack consisting of linked frames, which records both the binding environment and the control structure of the overall system). The *C* can, additionally, contain any of a number of internal objects, referred to as meta-objects, which act as markers. Three of the meta-objects, **PRED**, **STMT2** and **OP2**, will appear in the examples extracted from our full set of rules.

The *E* is conceptually an a-list of variable/value pairs, augmented by additional structure indicating the boundaries of lexical scope. In practice, the *E* is implemented in stack frames, containing value cells and displays, which allow the cell corresponding to a variable to be located. This is done in order that the binding search mechanism be uniform for compiled and interpreted code, simplifying the intermixing of the two evaluation regimens. The *D* contains the state of the machine which is to be restored upon returning from the current state, e.g. via a RETURN. As with the *E*, the *D* is implemented in stack-frames of the same form as those used by compiled code, again upholding the uniformity of stack frames.

The "program" for the SECD machine consists of a collection of state transition rules. Each rule specifies a pattern of values for the *C* and *S* components of the current state, and specifies a new state in terms of the current components. The applicability of the rules is tested in a prescribed order to avoid ambiguities. The order of the rules below is implied by the state/rule numbers.

In the actual implementation, we allow ourselves shortcuts. Both the search for the applicable rule and for the transformation are hand-coded in assembler language, rather than as rules for a general SECD interpreter. This allows us to arrange the rules in a search tree, rather than in linear order, and to manipulate the *S* and *C* stacks directly, examining their elements without popping them, and replacing elements in situ.

In order to give a flavor of the programming of the SECD machine, we will present a few of the rules defining our system. These include the rule which recognizes a COND statement, the rules which process the body of the COND, and the rules which process the sequence of expressions in a PROGN, either explicit or implicit.

In these fragmentary examples, bold-face items represent meta-objects, italic items represent place holders, and other objects represent themselves. Thus in the first rule of the fragments of the SECD rules displayed, the interpretation of the initial state:

{COND . *s*; *e*; **OP2** (*x* . *body*) . *c*; *d*}

is:

COND	the special-form COND
OP2	the meta-object which indicates that the object at the head of *S* is the value of the operator for the form being evaluated
(*x* . *body*)	the form
x and *body*	place-holders for the CAR and CDR of the form

Within the *S* and *C*, the sub-components are written as if surrounded by parentheses. In the example, the *S*

COND . *s*

represents a stack of one or more items, with the top item being "COND", and the remaining items (if any) represented by the sub-component *s*.

In the resultant state:

➡ {() . *s*; *e*; **PRED** () *body* . *c*; *d*}

we find the place-holders, such as *s* and *body*, marking the arbitrary components of the initial state which are used in the construction of the subsequent state.

Rule 8.2.1.2.

{COND . *s*; *e*; **OP2** (*x* . *body*) . *c*; *d*}
➡ {() . *s*; *e*; **PRED** () *body* . *c*; *d*}

This rule is invoked when the value of the operator of a form has been resolved, and is at the head of the *S*. The original form follows the meta-object **OP2**. *x*, in this form, is the operator expression which evaluated to COND. When this state is found, we proceed to a state with the meta-object **PRED** at the head of the *C*, followed by an arbitrary object (in this case NIL) and the body of the COND. This mimics the state in which the predicate of a clause in a COND statement has been evaluated, has returned NIL, and thus forces the machine to try the next clause. This is the state of the machine which matches the following rule in our example, but in this case the "next" clause is in actuality the first.

State 13.1.

{() . *s*; *e*; **PRED** . *c*; *d*}

This state is distinguished by **PRED** at the head of *C* and () at the head of *S*. This condition occurs when a predicate has been evaluated and returned a value of NIL. There are three sub-rules under this state.

Rule 13.1.1.

{() . *s*; *e*; **PRED** *e-seq1* ((*p2* . *e-seq2*) . *x*) . *c*; *d*}
➡ {*s*; *e*; *p2* **PRED** *e-seq2 x* . *c*; *d*}

In this rule, the remaining portion of the body of the COND starts with a pair, (*p2* . *e-seq2*), which must be tried next. The sequence of expressions which had followed the previously evaluated predicate, *e-seq1*, is discarded and replaced by the sequence of expressions following the subsequent predicate, *e-seq2*. At the same, time the new predicate, *p2*, is pushed on

the head of the *C*, where it will be evaluated, and the entire clause is popped off the body of the COND.

Rule 13.1.2.

{() . *s*; *e*; **PRED** *e-seq1* (*atom* . *x*) . *c*; *d*}
➡ {() . *s*; *e*; **PRED** *e-seq1 x* . *c*; *d*}

This rule discards any non-pairs found in the body of the COND. This allows the use of character strings as comments between the clauses of a COND. Note that "atom" is synonymous with non-pair.

Rule 13.1.3.

{*s*; *e*; **PRED** *e-seq1 atom* . *c*; *d*}
➡ {() . *s*; *e*; *c*; *d*}

This rule causes the value of the COND to be NIL if no predicate is satisfied.

State 13.2.

{*x* . *s*; *e*; **PRED** *atom y* . *c*; *d*}

This state is distinguished by an non-NIL value at the head of *S*. Since the state number is higher than 13.1, it must have this property.

Rule 13.2.1.

{*x* . *s*; e; **PRED** *e-seq y* . *c*; *d*}
(Where *e-seq* is a pair.)
➡ {() . *s*; *e*; **STMT2** *e-seq* . *c*; *d*}

If there is an expression sequence following the predicate here, it is treated as an implied PROGN. Again, the *C* components associated with the COND body are popped, and a new meta-object, **STMT2**, is pushed onto the *C*, together with the list of expressions to be evaluated. At the same, time the *S* is augmented with an arbitrary value, in this case, NIL.

Rule 13.2.2.

{*x* . *s*; *e*; **PRED** *atom y* . *c*; *d*}
➡ {*x* . *s*; *e*; *c*; *d*}

This is the "predicate-only" case. The expression sequence associated with the predicate, which evaluated to *x*, is not a pair. In this case, the value of the COND is simply the value of the predicate which is left at the head of the *S*, while the meta-object **PRED** and the remainder of the COND body (if any) are popped off the *C*.

We will now display the rules controlling the body of an explicit or implied PROGN. These rules are distinguished by the presence of the meta-object **STMT2** at the head of the *C*.

State 15.

{*x* . *s*; *e*; **STMT2** . *c*; *d*}

The value at the head of the *S*, *x*, is the value of the last evaluated expression in the sequence.

Rule 15.1.

{*x* . *s*; *e*; **STMT2** (*exp* . *e-seq*) . *c*; *d*}
➡ {*s*; *e*; *exp* **STMT2** *e-seq* . *c*; *d*}

The item in the *C* following the meta-object **STMT2** is the portion of the the PROGN body which has yet to be evaluated. If there remains one or more items in the expression sequence, the first is popped off and placed at the head of the *C* to be evaluated. At the same time, the previously computed value is popped from the *S*.

Rule 15.2.

{*x* . *s*; *e*; **STMT2** *atom* . *c*; *d*}
➡ {*x* . *s*; *e*; *c*; *d*}

If, on the other hand, the item following **STMT2** is an atom, then all the expressions in the body have been processed, and the value at the head of the *S* is the value of the last such expression. In that case, **STMT2** and the objects associated with it are popped from *C* and the value is left on the *S*.

Appendix B. Environments and Closures

There exist three distinct environments in which variables may be bound. Two of these are strongly interrelated. The third was designed as an afterthought, and is only weakly integrated into the system.

The first environment (from the view point of a process searching for the currently operative extant binding of a variable) consists of the chain of stack frames. Each stack frame contains a (possibly zero size) set of value cells, together with a datum, referred to as the display. The display provides a mapping from variables to value cells, and indicates which are lexical and which fluid (i.e. dynamically accessible). The stack frames are linked by a pointer which may differ from the link used during a RETURN operation. These environment links are marked to indicate the boundaries of lexical scope.

Starting from a particular stack frame, usually the current head of the stack or a frame pointed to by a state descriptor, the binding search paradigm is to search for any (lexical or fluid) binding of the desired variable. This search continues, following the environment chain, until a lexical scope boundary is crossed. The search then changes, looking only at fluid bound variables. If no binding is found when the root of the stack is reached, the current non-LAMBDA environment is searched.

A non-LAMBDA environment consists of one or more association list, linked by means of state descriptors. Each name-value pair in the association list contains an identifier (variable) and a value. If the final CDR of the list is a state descriptor, the non-LAMBDA environment captured by that state descriptor is searched in turn.

There is no unbound variable error. If the binding search reaches a non-state descriptor terminator of a non-LAMBDA environment without locating a binding cell for the variable in question, a new binding cell is created and initialized with the variable itself as the value. This newly created binding cell will be added to the first association list in the chain constituting the current non-LAMBDA environment.

The current non-LAMBDA environment acts much like a global environment, but it must always be borne in mind that it is only one of many. Every use of EVAL with a state descriptor, every application of a funarg, every switch of environment to a previously saved state, may install an entirely different non-LAMBDA environment.

In fact, this facility is used heavily during compilation to provide the compiler with a set of operator definitions which differ from the operator definitions bound in the environment in which the compiler itself executes. This allows both cross compilation (as in the MACLISP compatibility package) and optimization (by the provision of compile-time macros).

The third environment is a true global environment, containing one value cell for each identifier. This set of global value cells is not used by any part of the system itself, nor is it included in any binding search. There exist functions, EVAL-GVALUE and SET-GVALUE, which access and update global value cells, together with a special form, GVALUE, also used to access global value cells.

The general multi-environment philosophy precludes the use of such a global environment.

In a restricted sense, the system may be thought of as actually having four environments. There exist functions which allow the control chain, rather than the environment chain, to be searched for bindings. Such a search is never done by the standard evaluation mechanisms but, during certain operations involved in exception handling, it is explicitly requested.

This allows exception handlers to be established in the flow of control, rather than in the evaluation environment. Thus, an error occurring during the application of a funarg will be handled by the applier, rather than by the context in which the funarg was created.

The set of environments affects the collection of bindings captured by a state descriptor. The closure is a general purpose object, holding both a set of bindings and a control. It can be used for evaluation of variables (and by extension, expressions), and it can be used to resume computation at a remembered point in the past.

The evaluation of a variable with respect to a state descriptor elicits a search of the stack starting with the frame pointed to by the state descriptor (the frame from which the primitive operator STATE was called), and continuing, if need be, in the non-LAMBDA environment which was current at the time of the state descriptor's creation.

Note that at no time is the environment from which the evaluation is requested involved, unless it and the environment captured by the state descriptor shares a common ancestor. In this, evaluation with respect to a state descriptor differs from Common Lisp's evaluation with respect to a (lexical) closure.

It differs also in the fact that the LISP/VM state descriptor captures the complete environment, both lexical and dynamic. Thus any variable bound in the captured state is accessible, within the scoping rules, not simply the closest lexical variable.

Appendix C. The Structure of a Stack Frame

The stack in the Yorktown Lisp system is fully framed, as required by the Bobrow-Wegbreit stack model [Bobrow and Wegbreit 1973]. The frames are divided into two components, the head-of-environment (headE) and the dump. In the presence of state saving, a single headE may be referred to by two or more dumps.

The headE contains the value cells for the variables bound in its frame, both the lexicals and the fluids. In addition, it contains a pointer to the end of the spatially preceding frame, and a count of the excess dumps referencing it (over the single dump in the case of simple LIFO control).

The dump contains all the information required to resume execution of the function associated with the frame. This information may not be completely valid in the currently active frame, i.e. the frame of the function currently in control. All other frames contain the saved registers required to run the associated function, together with non-pointer data, the current state of the execution stack, a list of actions to be executed on returning from this frame, etc.

A more flexible structure would result if various components of the stack frame were reallocated. In particular, the tail-E, display, and abstack should be moved to the headE. This would provide two improvements in the system, one semantic, the other pragmatic.

On the semantic front, the move of the abstack area to the headE would promote the non-pointer data to the same status as the ordinary variables. While there is no means of using this data in pure Lisp, it is accessible via LAP or F*CODE. It can be used to implement a form of unboxed numbers, as well as for efficiently allocated and released (not involving the garbage collector) storage for numbers or character strings. While one may argue against the sharing of value cells among multiple saved environments, it is clear that all value cells should be treated equivalently.

The shift of the display and tail-E pointer to the headE would allow the creation of environment-only closures. As stated previously, the only form of closure supported in the Yorktown Lisp system captures the full environment and control. As such, they affect the amount of stack space which can be released upon function return or by the garbage collector.

The current stack design was the result of careful thought. The short comings were not at all obvious until the system had been in use for a number of years. We might hope to have more foresight, should we attempt a redesign, but we should pay particular attention to providing for further modifications in a less painful manner than having to recompile all existing source code.

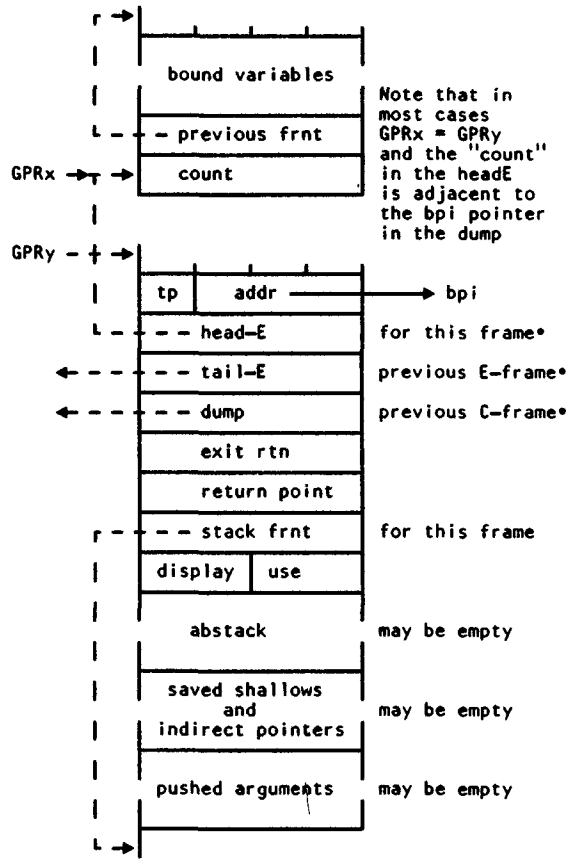

Figure 2. Stack Frames