ZIL: An Implementation of Lisp on MVS/XA at C.S.Draper Laboratory

Stephen E. Bacher

DESCRIPTION OF ZIL

"ZIL" is an implementation of Lisp on MVS/XA running on an IBM 3090-200 System/370 mainframe at the Charles Stark Draper Laboratory. The system includes a full Lisp interpreter and compiler, as well as a full-screen dialog user interface developed in IBM's ISPF (Interactive System Productivity Facility) product.

ZIL began life as an outgrowth of the Lisp 1.5 interpreter written by J. F. Bolce at the University of Waterloo, previously the only available MVS Lisp at the Laboratory. In order for the first ZIL compiler to be coded, the Waterloo interpreter was locally modified to permit features such as multiple input and output files, macro definitions and optional arguments to selected functions. Once the compiler was able to generate a complete top-level Lisp program, it was used to build the ZIL interpreter (which was itself coded in ZIL), and then to bootstrap the ZIL compiler. This process was then repeated using ZIL as its own bootstrapping environment.

As ZIL has been developed, features from other dialects (chiefly Maclisp) have been added and progress is being made toward the goal of Common Lisp compatibility. ZIL supports a wider variety of features than did its parent system, including comprehensive reader syntax that permits both Lisp 1.5-style code and Common Lisp-style code to be loaded.

Although ZIL is being made CL-compatible when possible, there are still areas where it differs. For example, ZIL's "special" variables do not behave exactly as they are supposed to in CL, but more like traditional Lisp "free" variables.

At this time ZIL does not have packages, hashtables, multiple values, or numerous other advanced facilities documented in Steele (1984). However, most of the basic features are present, certainly enough to load in the typical Lisp source program. In any case, incompatibilities with other dialects may be overcome by many means, notably the DEFLOAD special form (see below).

Data Typing

ZIL has the standard data types of conses, symbols, fixnums, and flonums; it also has bignums, strings, vectors, compiled code subroutines, compiled and interpreted lexical closures, and structures (used by DEFSTRUCT to implement user type extensions). Functions are available to build objects of one type from objects of another, as well as to coerce between types.

Type information is contained in the high-order 8 bits of each ZIL object; this technique works better on 370 than a typed-pointer architecture, since the hardware makes it easier to test bits in memory than to test bits in registers. 31-bit addressing support is not compromised, since the high-order byte of each object is considered to contain type information only if the high-order bit is one (signifying an "atom").[1] When the high-order bit is zero, the object is a cons, and its CAR may easily contain a 31-bit address, as none of the other bits in the high-order byte are meaningful to the type scheme in that case.

System-Specific Functions

ZIL functions are stored externally as MVS load modules, and are brought in by the ZIL loader, which uses MVS program fetch facilities to load the code and then does its own relocation of function pointers. Not only does this format permit the same functions to be linkedited into top-level ZIL programs, but it allows low-level ZIL functions to be coded in assembler and easily integrated into the system.

The DEFLOAD special form is used in ZIL to provide "shadowing" and compatibility with other Lisp dialects; its main

[1] The high-order bit is ignored during addressing access in both 24-bit and 31-bit address modes in the 370 Extended Architecture.

purpose, however, is to map Lisp function names to names of compiled code load modules that reside on direct access MVS libraries. Since IBM restricts load module names to 8 alphanumeric characters, it is necessary (for purposes of linking and automatic loading) to provide alternate IBM-compatible names for functions whose traditional name does not meet this standard. Thus, names of 8 characters or less are associated with function symbols via DEFLOAD. This works under both the interpreter and the compiler; in the latter case DEFLOAD is used to assign an external name in the generated code which may be resolved at link time or execution time. DEFLOAD is intimately related to the ZIL autoloading process described later.

ZIL has a "TSO" function, providing an interface to MVS's native interactive time-sharing system (similar to a "DOS" function on a microcomputer implementation). This permits the execution of TSO commands and command procedures (CLISTs) inside the ZIL environment.

> **Note:** MVS provides a service to invoke TSO commands from within a user program running under TSO, available only if the TSO/Extensions product is purchased. However, ZIL's implementation of this does not require TSO/Extensions to be present.

There is a group of functions which enable the user to write ISPF dialogs in ZIL; these invoke ISPF dialog services using the "ISPEXEC" syntax familiar to CLIST writers, and can retrieve and update values of ISPF dialog variables. This facility has been used to implement the ISPF dialog writing capability available in the MVS implementation of DOE-Macsyma. (IBM's ISPF version 2 is required for this facility.)

IBM's VS FORTRAN subroutine library is used to implement many numerical functions, insuring the highest possible degree of accuracy.

The arbitrary-precision integer (bignum) support is coded entirely in highly optimized assembler; intermediate consing is virtually eliminated.

A number of PL/1-like string operations are present in ZIL, making it useful for real-life applications like text processing.

A unique approach to handling attention interrupts (the MVS/3270 equivalent of the "break" key) insures recoverability to any desired level of computation without the implementation's having to resort to the standard MVS means of handling attentions (generally either subtasking or the generation of polling instructions in compiled and other run-time code).

Lambda list features of both Maclisp (FEXPR's, LEXPR's) and Lisp Machine Lisp (full nested destructuring of arguments for all lambda lists) are supported, including full support for &OPTIONAL, &REST, &AUX and &KEY variables, in both the interpreter and the compiler (well, actually FEXPR's are simulated in the compiler).

IMPLEMENTATION

The IBM 370 architecture has an undeserved reputation for being an impediment to the implementation of functional programming languages like Lisp. We have never found any difficulty with the architecture, especially since we have been programming in it so long, but there are some built-in difficulties associated with using the MVS operating system, as opposed to VM, for example.

The memory management schemes chosen are directly affected by the absence of any way under MVS (under the constraint of remaining in problem program state) to declare pages of memory read-only. Because of this, ZIL cannot rely on tricks like protection interrupts to control allocation of storage. However, through use of a handful of registers dedicated throughout the ZIL environment, we are able to maximize speed of most CONS operations by maintaining pointers to the beginning and end of the current "free list" in registers.

In-line compiled lists present another, more intractable problem. MVS does not provide a way to change the protection status of an area of memory at run time, and does not even allow the programmer to allocate read-only pages. Since in-line lists are physically part of loaded compiled code, and there is no convenient way to prevent such objects from being updated via RPLACA-type operations, an exposure exists. We deal with the problem at this time by advising users not to compile in-line lists unless they are sure that they will not be clobbered. The garbage collector will not be able to sweep such objects. However, a total object-copying scheme for loading compiled lists and atoms may be implemented at some point to resolve this; although it will not prevent destructive updating, it will at least prevent corruption of such objects following a GC.

ZIL Internals and Philosophy

ZIL was designed from the beginning to be as fast as possible without compromising maintainability. The function call interface, for example, does a minimal amount of register saving and restoring, and many low-level routines do not save register contents at all. Since the IBM 370 does not have a hardware stack, stack locations are pushed and popped lexically within a function, altering the stack frame offset at compile time rather than at execution time. A three-instruction sequence at the beginning of each routine checks for stack overflow. The coding of many functions in assembler reduces the need for consing and permits values to be kept in registers more frequently.

The emphasis on efficiency is at the expense of compile-time error checking and debugging. Calls from one compiled code subroutine to another are not traceable, and compiled-to-interpreted code transfers of control must be effected via calls to APPLY or FUNCALL. Error checking is minimal in compiled code; the interpreter will detect errors, and compiled code which calls out-of-line functions will detect errors if those functions contain such checks.

Bignums

The ZIL internal structure of bignums is as follows:

A bignum is an object in vector/string space which has a 1-byte type field and a 3-byte length field indicating the byte length of the bignum text, much like a string or vector. The body of the bignum is a series of contiguous 32-bit words, each of which contains a digit of the bignum in base 2**31 (31-bit radix). These words are stored in reverse order, least significant word first.

Negative bignums are stored using sign and magnitude, with the high-order bit of the leftmost word set to 1. All other high-order bits (in positive and negative bignums) are zero.

Bignums are uniquely represented and always normalized: this means that (1) a bignum can never have a value between -2147483648 and 2147483647 inclusive, as this is always forced to be held as a fixnum; (2) the most significant (rightmost) word is never zero, since leading zero "bigits" are always removed; and (3) the minimum number of 31-bit "bigits" in a bignum is 2. The combination (-2147483647 1) can never occur, since this would represent -2147483648, which must be a fixnum.

Owing to the above architecture, tests like ZEROP and MINUSP are extremely fast (bignums are never ZEROP, and testing the high-order sign bit works for fixnums, flonums and bignums equally). The ZIL functions BIGLIST, BIGNUM and MAKE-BIGNUM exist to convert between bignums and lists of fixnums that make them up, mostly for our own testing. Automatic conversion of bignum to floating-point is done by the arithmetic functions when necessary.

Garbage Collection

ZIL's garbage collector combines two approaches: a variant of the Schorr/Waite (Winston, 1984) non-recursive pointer-altering algorithm for collecting cons space (which contains lists, symbols, fixnums and other basic objects), and a copying scheme for vector/string space (which holds strings,

vectors, bignums and structures). The latter requires allocating the space in two sections, one of which remains unused while the other is active; this method, suggested by Carrette (personal communication, 1984), is feasible on an MVS system with its vast memory resources. The two techniques are closely intertwined, as the forwarding and relocation of objects in vector/string space is performed whenever the tree-scanning portion of the GC encounters an object that does not point back into cons space.

cons space is maintained as a single area of memory divided up into cells of two 32-bit words each; these are treated as contiguous space until a garbage collection is done, at which point the area is reordered into a linked list. The pointer to the CONS routine is located in a common area and altered after the first GC to reference the new data structure. In this way first-time initialization of the cons space is avoided, but the free list can be maintained without compaction being necessary (since ZIL requires symbol pointers to remain unaltered, it is necessary for them to reside in a space that does not get relocated; thus the use of two schemes).

The GC is written entirely in assembler language and is highly efficient; it can collect a typical 5-megabyte space in about 0.2 CPU seconds. The sweep phase, generally the longest portion of the GC, uses the System/370 Vector Facility for added speed.

Variable binding

ZIL is primarily dynamically scoped, although lexical scoping can be effected through use of lexical closures. Variables are deep-bound, with their values pushed onto an alist-type environment. Compiled functions that reference "free" variables perform a single scan of the environment for values at the beginning of the function; this scan sets stack pointers to the variable/value cells on the environment for quick access from the code so that the binding environment need not be searched on each reference. This has the disadvantage that non-locally-bound variables must have been bound at some previous level to be referenced; global variable bindings are generally established via a PROG or LET just under top level in order to enable SETQ's of free variables. For this reason, an alternate method of retrieving the values of "free" variables in compiled code exists, used primarily when compiling "modules" (files). This generates inline code to locate an existing binding (which may be global) and cache it on the stack so that multiple references in the same function need not go through the overhead of searching for the value more than once.

ZIL also possesses a "symbol-autoload" facility. If a symbol is not currently bound, and it has an AUTOVALUE property, the value of the AUTOVALUE property is used to "autoload" the value of the symbol, similarly to the way function definitions are autoloaded. This feature permits the loading in of a file which contains a DEFVAR for the symbol, for example, or delaying the execution of some arbitrary form that initializes the symbol until it is actually asked for.

The Compiler

The ZIL compiler is a separate top-level program, rather than a function that can be invoked from within ZIL. While this may appear to be an inconvenience, it has the advantage of avoiding the environment-bashing problems associated with LISPs that invoke the compiler from within a LISP session. The compiler generates assembler source code that is then assembled into an object module, which is in turn processed by the linkage editor.

ZIL's compiler is a three-pass operation. The first pass reads all the forms in the source file, collecting function and macro definitions and processing declarations, compile-time directives and DEFLOADs. The second pass transforms the function code into intermediate lists of pseudocode; during this time all macros are expanded, instances of tail recursion are flagged for conversion to iteration, and unbound variables are detected and added to internal tables (using this scheme ZIL is able to determine which variables are "free" without the user having to provide declarations). The third pass converts the pseudocode into assembler language source output; this is subsequently processed by the IBM assembler to form an object

module, which is in turn input to the IBM linkage editor, producing the load module which is the compiled code in its final, loadable form.

There are three kinds of entities that can be compiled: functions, modules and programs.

In program mode, the input to the compiler is a sequence of forms to be executed as a program; a module is created that appears to the operating system as a separate program and can be invoked in the foreground (under TSO) or in batch (via JCL). The ZIL interpreter and compiler are instances of this mode, as are the OPS5 and Macsyma interpreters.

In function mode, the input consists of a function definition (DEFUN) along with accompanying auxiliary function definitions and macro definitions if needed. Other top-level forms are not permitted. The output of this compilation is a module which cannot be executed by itself, but may be loaded by the interpreter's autoloader or linked in with a program that was compiled in program mode. This is equivalent to the Common Lisp COMPILE function, although it is used externally rather than from within the interpreter. This technique is used primarily to compile ZIL's built-in defined functions that are too complex to code in assembler.

In module mode, the input is a file of Lisp code that includes function definitions, random forms and declarations. Because loadable entities in ZIL must be executable functions, the file is treated as one huge function that envelopes all the code to be loaded while establishing compiled definitions for the DEFUN's contained therein. All non-defining forms in the file are converted to compiled format, to be executed at load time. Loading the file consists of bringing in the huge function (via the autoloading process) by executing it one time only. This is the equivalent of Common Lisp's COMPILE-FILE.

Compile-time Operations

Compile-time operations are generally specified via EVAL-WHEN, although certain forms (like DEFMACRO) are handled specially to side-effect the compiler's environment. A DEFMACRO (in fact, any form that defines a macro) creates a "compiler macro" definition (called a CMACRO) that the compiler sees when it processes the source forms. Facilities like SETF that need to see true macro definitions at macroexpansion time require conventional macro definitions to be made available at compile time as well; this must be done by enclosing the relevant DEFMACRO's in an (eval-when (compile ...) ...). The effect of a DEFMACRO inside (eval-when (compile)) is to establish a MACRO property known at compile time, as opposed to the CMACRO property established by DEFMACRO outside of the EVAL-WHEN. In addition, there is an XDEFMACRO form, which compiles into code that establishes a macro definition at load time. Other forms (like DEFSTRUCT) which need to establish compile-time properties usually do so by being macros which expand into DEFMACRO's enclosed in the appropriate EVAL-WHEN wrappings.

Since source transforms have not been implemented yet, CMACRO's are the way the compiler transforms, say, (+ A B C) into (ZILADD (ZILADD A B) C), where ZILADD is the built-in two-operand addition function, without affecting the definition of "+" in the compiler's own LISP evaluation environment. In previous releases of ZIL, the compiler would not even recognize conventional macro definitions when expanding source forms to be compiled; part of compiler initialization consisted of copying MACRO properties to CMACRO properties for all built-in symbols, so that a user could define a macro that side-effected the compiler's run-time LISP environment without affecting compilation. This proved to be unworkable, particularly for the way that Macsyma macros are loaded at compile time. It was discovered that such a feature was relatively useless anyhow, so the compiler was modified to see both MACRO and CMACRO definitions.

Autoloading

The evaluator will automatically search for a function definition when it FUNCALL's, or when it attempts to evaluate a list whose CAR is, a symbol for which no function

definition currently exists. It does this by inspecting the symbol's AUTOLOAD property. If the value of this property is a string, it is assumed to be the name of a file ("data set") which is then loaded interpretively into the ZIL environment; this file is assumed to cause the symbol to receive a function definition. If the AUTOLOAD property value is a list, it is evaluated as a form; this form is assumed to cause the symbol to become defined. If the AUTOLOAD property value is a symbol, it is assumed to be the value assigned by the DEFLOAD special form; if so, or if there is no AUTOLOAD property, the ZIL loader is invoked to locate the module of that name (or the symbol name itself, if there is no AUTOLOAD property) on the ZIL load library (i.e. the librari(es) from which the main ZIL module was loaded, which constitute the task library setup). If it is found, it is brought into main storage by the ZIL loader, which searches for and resolves all references to external compiled functions in the code (this results in recursive loading and resolution of other compiled code at load time).[2] The code is then made the SUBR property of the function; and the FUNCALL proceeds or the evaluation is retried. The AUTOLOAD property is removed when necessary to prevent infinite iterative attempts at evaluation; an "undefined function" error is signalled when all attempts to resolve the function definition fail.

Compiled programs generally have their subfunctions linked in with them by the linkage editor in one large load module; thus there is no need to have ZIL load these functions at run time. To prevent duplication of compiled code in storage, the ZIL loader keeps a lookaside table of module names that are likely to be hard-linked with the interpreter, which it searches before it goes to the operating system to locate the code. In cases where functions must be loaded from the library, ZIL uses the operating system's own control blocks to keep track of which modules are currently in storage.

I/O

The term "file" has a different meaning to IBM than to the rest of the world; what everyone else calls a "file" IBM calls a "data set", and a "file" (also known as a "ddname") refers to a symbolic 8-character name associating a particular data set allocation with a program OPEN request. This fits into Lisp rather well, as a "file" is what is returned by the OPEN function (and passed to READ, PRINT, CLOSE, and other I/O functions), whereas what gets passed to OPEN is a system-dependent name (a "dsname" in MVS).

When a "data set" is opened, it is dynamically allocated using MVS dynamic allocation services, and then opened; ZIL assigns a file name ("ddname") which is used to construct a file object that ZIL returns as the value of the OPEN function. This object, which is actually a symbol whose print name is the final 4 characters of the "ddname", is used in READ and PRINT calls, and is also used to close the file.

If a file name (4-character symbol) is requested in an I/O operation for a file that has never been opened, ZIL searches for an existing file name as follows:

If the function needs an input file, ZIL concatenates "ZILI" to the file name and looks for an MVS ddname of that form.

If the function needs an output file, ZIL concatenates "ZILO" to the file name and looks for an MVS ddname of that form.

ZIL then automatically opens the file (except that if the ddname is allocated to the terminal, ZIL makes it a terminal file). The constraint on the ddname is a throwback to the modified Lisp 1.5 interpreter, which was frugal with memory, and therefore required all

---

[2] This applies only to "built-in" ZIL functions, or those that have been declared "built-in" by the DEFARGS compiler directive. Other functions are invoked dynamically by a FUNCALL-type interface.

files to be known at startup time. With ZIL's use of memory above the 16-megabyte line, such parsimony is no longer necessary, but overhauling the file system is too radical a change to make at this time.

Terminal input and output in an IBM 3270 environment is radically different in its demands from typical ASCII terminal processing. TSO provides a line-mode transparency to the application, but because of the nature of 3270's some operations are not possible. For example, carriage returns are meaningless in the EBCDIC environment. For this reason, new-line conditions are simulated in ZIL by detecting probable end-of-line situations during terminal input (although lines longer than the terminal line size can be typed in if desired). This also presents problems with file I/O. Under MVS, a "data set" consists of "records", which may be fixed-length or variable-length; in either case they do not terminate with carriage returns. The new-line simulation applies here as well. ZIL has a function NEWLINEP which returns true if the most recent character-retrieving operation on a file "ran off the end" of the record; the next such operation will return the first character of the next record. The READCH function will return NIL in this circumstance, while the TYI function returns the equivalent of hex 0D ("carriage return"). Similarly, strings that extend across record boundaries have EBCDIC "carriage return" characters (hexadecimal 0D) inserted in them; the printer recognizes these characters when writing to the terminal and generates the proper control sequence to cause the data to be printed out on a new line on the 3270.

When ZIL writes to the terminal, it uses assembler-language-level TSO terminal communication interfaces to maintain control over the presence or absence of carriage returns, and to pad the output line with null characters to the end of the terminal row. This enables optimal overtyping of output lines without having to use the ERASE EOF key to clear blanks, or having to worry about stray attribute bytes. Terminal input is similarly managed with an eye toward keeping an accurate count of characters actually typed and retention of alphabetic case.

EBCDIC VS. ASCII

Since IBM/370 software uses the EBCDIC encoding scheme rather than ASCII, some differences necessarily exist between ZIL code and code in other Lisps. Instead of an ASCII function, ZIL has an EBCDIC function to convert a fixnum to a character, and an UNEBCDIC function to do the reverse. Most Lisp code that uses the *#/n* or *#\n* syntax works correctly under ZIL; only code that has hard-coded ASCII numbers in it loses.

Fortunately, Common Lisp does not mandate reader support for square brackets, which are missing from EBCDIC. However, FORMAT does make use of them; we have not yet determined the best way to deal with this. (The ZIL implementation of FORMAT does not include the options that use either braces or brackets at this time.)

Mainly for Macsyma, ZIL attempts to support square brackets in every way possible. Square brackets input from files as hex AD and hex BD are supported; for output to a file, they may be written as the same hex characters or translated to another character configuration (via use of the ZIL SETBRACK function). Terminal output is more problematic. ZIL will attempt to send the proper character sequences to the terminal to print brackets, if it can determine that such is possible. In Macsyma, for the appearance of consistency to the user, brackets display as curly braces by default, although the user may change this by setting the Macsyma variable BRACKETS. We have also extended the Macsyma parser to accept curly braces as being syntactically equivalent to square brackets.

EXTENDED ARCHITECTURE (XA) SUPPORT

ZIL's largest memory areas are cons space, the control stack, and the "oblist" hash table for interning symbols. Storage for all of

these is acquired above the 16MB line (i.e. in memory accessible in 31-bit addressing mode only) to allow for sufficient size to perform huge computations efficiently. Due to architectural limitations which have not been overcome in the current release, vector/string space and all ZIL executable code must reside below 16MB. ZIL itself, however, runs in 31-bit addressing mode (leaving that mode only to invoke I/O operations, which require 24-bit addressing).

The sizes of cons space, the control stack, and vector/string space are all settable at run time by passing initialization specifications to the top-level ZIL program in the OS PARM field (or in the command operand field if the top-level is invoked as a TSO command processor). Initialization specs, if present, are enclosed in backslashes at the beginning of the parameter field. The remainder of the parameter field is passed to ZIL and is accessible via the ZIL GETPARM function.

THE SYSTEM/370 VECTOR FACILITY

The vector facility on the IBM 3090 is an integral part of the CPU, and has a few instructions which are particularly interesting in Lisp applications. For example, a single machine instruction can load or store a vector-register pair consisting of 128 64-bit cons cells in locations pointed to by a third vector-register containing 128 pointers. Speedup is typically a factor of 2-4, and up to 10 with floating-point.

To some degree, the vector facility has influenced the design of ZIL. Separation of the heap into a space of 64-bit objects, consisting mainly of cons cells, and a vector-string space of unequal-length objects has permitted fast vector operations on 64-bit object-space.

BENCHMARKS

Of course, running on one of the fastest hardware processors around means that ZIL would run fast even if it were not designed well. But we ran some standard benchmarks, including some from Gabriel (1985), and results were favorable in terms of processor time used, despite the lack of high compiler sophistication. The performance of ZIL using variables that are not locally bound (based on Gabriel's STAK benchmark) compares favorably with other implementations (including PSL on an IBM 370), attesting to the efficacy of ZIL's deep-binding scheme.

USABILITY FEATURES

Although ZIL has tended to stress internal development at the expense of user-friendliness at times[3], many features helpful to the interactive ZIL user have been added. A "dribble" capability allows the user to copy all terminal input and output to a file or to a stream that can be printed off by the operating system.[4] The ED function interfaces to the ISPF editor; this works in conjunction with the PP (pretty-print) function to allow the user to create and modify function definitions on the fly.[5]

---

[3] For example, there is no interactive debugging other than tracing, and that works on interpreted code only. The compiler cannot be accessed within the LISP environment, although the assembler-code-outputting part of the compilation process has been successfully invoked inside ZIL. There is no fancy full-screen windowing or graphics, which says more about the IBM environment than anything else.

[4] Unfortunately, dribbling cannot be turned on and off within ZIL; it must be specified on entry to ZIL, and remains active until exit from ZIL.

[5] An ISPF edit macro (coded in PL/1) enables parenthesis matching under the ISPF editor; the user can place the cursor under a parenthesis or brace and press a PF key,

SOFTWARE PORTING

The first successful installation of a Lisp-coded software product on MVS via ZIL was Carnegie-Mellon's expert system builder, OPS5. An MVS interface (also using ISPF) was developed to make the system easy to use for IBM mainframe users. A number of experimental expert systems have been developed under ZIL and OPS5, including:

an interface between OPS5 and IPCS (MVS's Interactive Problem Control System, a full-screen dump-analyzer);

an expert system which generates and submits JCL to unload the contents of a tape file onto MVS disk files;

a DCF-to-GML text processing conversion program.

A complete implementation of DOE-Macsyma (the symbolic algebra program) is available to MVS users under ZIL, including full support for translating and compiling Macsyma and Lisp code, a facility for coding ISPF dialogs in Macsyma, and interfaces to TSO and the local Draper electronic mail system. DOE-Macsyma has been the driving force behind many of the improvements in Zil since its inception.

In addition, an implementation of FLAVORS (apparently the most popular object-oriented programming facility in Lisp) is available in ZIL which follows Weinreb and Moon's original Lisp Machine FLAVORS paper very closely, as well as containing some features present in the NIL FLAVORS implementation.

FUTURE OF ZIL

We are working on a new architecture that will be 100% Common Lisp compatible, while retaining the features that enable software like DOE-Macsyma to run successfully. This will probably make heavy use of extended data types (through DEFSTRUCT) and more varied binding techniques. A "fluid" variable binding mechanism will be used to effect "pseudolexical" scoping, while "special" binding will be available simultaneously. Multiple control stacks for interactive debugging are also a possibility.

Native support for 3270 features such as extended attributes is a possibility; full-screen and graphics interfaces might be useful as well. This is not urgent, however, since we have access to both ISPF and the TSO Session Manager.

An interface to FORTRAN is in the planning stages, allowing FORTRAN programs to be used as ZIL functions.

The compiler will probably be rewritten, providing support for additional features of Common Lisp. For example, the compiler currently utilizes compile-time-only macro definitions, which in conjunction with DEFLOAD provide a simple source transformation capability. However, true source transforms would be desirable, as well as a better declaration scheme which would allow more control over the type of code generated for various functions.

The System/370 Vector Facility might be used more extensively by the GC as well as by library functions and compiled code.

REFERENCES

Gabriel, Richard (1985), *Performance and Evaluation of Lisp Systems*, MIT Press.

Winston, Patrick, and Horn, B.K.P. (1984), *Lisp*, Addison-Wesley.

Steele, Guy (1984), *Common Lisp: The Language*, Digital Press.

---

and the edit macro will move the cursor to the matching parenthesis or brace.

Weinreb, Daniel, and Moon, David (1980), *Flavors: Message Passing in the Lisp Machine*, A. I. Memo No. 602, Massachusetts Institute of Technology.

---

[1] **Editor's note:** The IBM 3090 is a very fast machine; its basic cycle time is 18 nanoseconds, and most instructions take only one or two cycles; also note that the particular machine at Draper Labs has a large amount of real memory (32megabytes or more). For comparison purposes, the authors have verbally mentioned that a ZIL-based Macsyma runs many Macsyma benchmarks at about 5 times the speed of Vaxima running on a VAX/8650, at about 10 times the speed of the original MacLisp-based Macsyma running on a PDP-10(KL10), and at about 20-30 times the speed of a NIL-based Macsyma running on a VAX/780.