



# Lisp Systems in the 1990s



**JAMES R. ALLARD**  
**LOWELL B. HAWKINSON**

**G**2 is a real-time expert system development and delivery environment for applications in manufacturing, process control, financial trading, telecommunications, telemetry, environmental monitoring, and wherever the monitoring of real-time data is required. It integrates a significant number of software technologies, such as object-oriented programming, rule-based reasoning, procedure execution, pseudo-parallel task execution, window systems, animated graphics, structured natural language, network services, and real-time data collection and management facilities.

G2 is written in a subset of Common Lisp, and runs in five different Common Lisp implementations on 14 different hardware platforms. Though Common Lisp would not be an obvious choice to most engineers as a real-time programming environment, we feel we have benefitted greatly from its use. Common Lisp is generally accepted as a powerful language for implementing artificial intelligence (AI) and symbolic processing programs. Other advantages for us were its powerful extensions, the strong development and debugging environments available in Common Lisp implementations, the ease with which compilers and interpreters can be written, and the quality of the Common Lisp standard itself. We have found that the different implementations of Common Lisp have very few variations, making it relatively easy for us to port our products to a broad range of platforms.

Within the AI software marketplace, several criticisms have been made in recent years against Common Lisp as a practical language for delivering commercial software products. Programs written in Common Lisp have been criticized as slow, unpredictable in response due to "garbage collection," overly large (due to the many built-in facilities), and difficult to integrate with other software. Common Lisp does have a large number of facilities, many of which are not available in other widely used languages. This presents users of Common Lisp with a large design space and many choices to make regarding which approach is best for a particular problem. We believe that undesirable program characteristics such as those mentioned are the consequences of particular programming styles made possible by the large design space, rather than inherent failings of Common Lisp itself. In fact, by restricting oneself to a subset of Common Lisp, it is possible to write code that is isomorphic to code written in more standard languages, such as C. Then, by using the power of Common Lisp to go beyond what can be expressed at the level of C, one can achieve higher quality, easier maintenance, and even more efficient software for demanding real-time applications.

This article discusses four techniques for programming real-time applications in Common Lisp. The first is memory management in real-time environments. We have found that garbage collection imposed an unacceptable level of overhead for real-time applications. Therefore, we have written our software to avoid doing garbage collection at run time. Instead, we explicitly manage all data structures, as would be done in more standard languages.

The second topic is macros in Common Lisp. Rather than a source text replacement facility, Common Lisp macros are true programs that execute at macro expansion time. This powerful capability is used throughout our code to implement abstractions of complex control structures, object-oriented programming facilities, and type-specific arithmetic. Achieving comparable power in most other languages requires preprocessors that perform transformations before source files are compiled by the standard compiler for that language.

The third topic is type declarations to support compile-time optimizations. Many of the primitive operations of Common Lisp (even the basic arithmetic

operations) can have very complex behaviors. By utilizing declarations about types, Common Lisp compilers can emit machine-instruction arithmetic operations. Optimizing operations based on declarations applies to other operations within Common Lisp as well.

The last topic concerns techniques for reducing the size of applications in Common Lisp. By default, all facilities of Common Lisp are carried into final program images, unless something special is done to eliminate them. All major commercial implementations of Common Lisp now provide some means of omitting unused facilities of the language from final program images.

## Memory Management

The issues of memory management for object-oriented programs like G2 are similar to those of any program that creates and recycles large numbers of data structures of varying sizes. In typical Common Lisp programs, memory management is handled automatically by a memory reclamation facility known as the garbage collector (see [1]). We have found the performance characteristics of garbage collection to be unsuitable for real-time applications, due to the interruptions of normal processing and/or the space/time overhead involved. Therefore, in using Common Lisp for such applications, we have chosen to manage memory explicitly—much as one would in conventional languages.

In programs which manage memory explicitly, a block of memory called the *heap* is set aside for dynamically created data structures. When it is necessary to allocate a new instance of a data structure, memory is set aside from the heap and a pointer to this block of memory is returned to the program. When the instance of the data structure is no longer needed, the program returns the allocated memory back to the memory management facility for reuse. This is called *freeing* the memory. Blocks of freed memory of similar sizes are kept in separate resource pools. When an allocation

of a new data structure is made which requires a block of memory of the same size as a block which had previously been freed, then a pointer is returned to this block of memory instead of allocating a new block from the heap.

Explicit memory management is a burden to programmers. Garbage collection in Common Lisp frees programmers from this burden. Programmers can confidently build data structures that arbitrarily share subparts, contain circular references, and are generally interwoven into arbitrarily complex graph structures. When such data structures are no longer needed, a programmer need not decompose and individually reclaim each piece. Pointers to entire collections of data structures may be simply dropped, with confidence that garbage collection will eventually reclaim all memory associated with those structures.

However, this feature carries a cost. Garbage collection takes time, typically added as an overhead to the Common Lisp functions that create new heap-allocated data structures. When an allocation function is called that requires more memory than is available in the heap, garbage collection is triggered to attempt to reclaim memory that is no longer being used.

Garbage collection techniques have been developed that are real-time with respect to the total size of the heap. They achieve this by performing small, incremental amounts of the garbage collection work while allocating structures (see [2, 3, 4]). All of these techniques, however, even the so-called "ephemeral" ones, still have execution time dependencies on the size of scanned data structures.

All Common Lisp garbage collectors determine which heap-allocated data structures are still in use by scanning used data structures

for pointers to other data structures, starting from a root set of pointers including global variables and stack frames. The referenced data structures are scanned in turn. In this way, every data structure still in use will be scanned. (Ephemeral techniques optimize this process by limiting scanning to the most recently created data structures, but the process of scanning for these structures does not change.)

The smallest amount of incremental work that these techniques perform is scanning one Common Lisp object. The largest possible objects in Common Lisp environments are vectors. Common Lisp requires that implementations be able to allocate vectors at least 1,024 elements long, though typically implementations allow vectors that are much longer. Therefore, even though the defined techniques are real-time with respect to the size of the heap, they still have execution time dependencies on the sizes of the heap-allocated data structures, and these data structures can be quite large.

In any case, garbage collectors on non-Lisp machine hardware still do not, as far as we are aware, use incremental algorithms. Interruptions of normal program execution for garbage collection in the best implementations can still last for hundreds of milliseconds in typical situations, and tens of seconds in worst cases.

This is unacceptable for our real-time needs, so we have written G2 to avoid invoking the garbage collector. We accomplish this by always retaining pointers to allocated objects, by allocating recycled data structures instead of allocating new ones, by avoiding or reimplementing Common Lisp facilities that implicitly create garbage, and by using objects created with dynamic extent (defined later) where we

could not avoid new object instantiations. By managing memory in this way, we never exhaust the heap, and therefore do not trigger execution of the garbage collector.

There were some facilities in Common Lisp whose services we required, but whose use implied the creation of data structures which could only be reclaimed through garbage collection. The file input and output routines were examples of such facilities. We have reimplemented all such facilities—sometimes by writing the underlying primitives for the facility in C and then linking that code back into our Common Lisp programs. Lisp interfaces are then written to these primitives that mimic the interfaces to the Common Lisp facilities, thus allowing us to use familiar tools while achieving garbage-free execution.

When programmers first become acquainted with Common Lisp, they are often surprised to learn that it may represent floating-point numbers and large integers as heap-allocated structures. In Common Lisp the type of any datum can be determined at run time from the datum itself. With the exception of the *immediate* representation of some integers (typically those that fit within a 30-bit two's-complement representation) and characters, and optimizations based on type declarations, all objects in Common Lisp are represented as heap-allocated data structures which contain a field for a type descriptor as well as fields for the data contained within that object.

The only Common Lisp objects for which we could not always control or avoid allocation were floating-point numbers and large integers ("bignums"). For these we have used a tool similar to the dynamic extent declaration defined within the draft ANSI standard for Common Lisp described in [6].

# We have chosen to manage

Objects in Common Lisp usually have indefinite extent, which means they persist until the garbage collector can prove they are inaccessible. Dynamic extent is a declaration that the lifetime of the object that is the initial value of a variable is limited to the lifetime of that variable. Common Lisp implementations typically optimize the dynamic extent declaration by allocating the initial value object on the stack. When the function containing the declaration returns, the memory for the object is reclaimed.

Our code actually uses an allocation technique whereby objects are created within temporary areas of memory. We have defined a special form that establishes a region of code within which objects can be allocated from the heap. On exit from that region of code, any objects that were created are immediately reclaimed. Within this region of code, we say objects are created within the temporary area, and the region itself is loosely referred to as a "temporary region." This style of reclamation allows us to use functions that create and return heap-allocated data structures—such as floating-point numbers—as their results. Many Common Lisp functions that return numeric results operate in this fashion. Structures that have been created within a temporary area can be examined anywhere within the temporary region, as well as anywhere within any functions called from the temporary region. If we want to retain anything created within the temporary region, we must copy it into a nontemporary data structure.

Temporary areas are typically implemented as a memory allocation heap whose allocation pointer is saved on entrance to the region and restored on exit. This is not a defined feature of Common Lisp, but we have obtained it as an extension to the implementations we use.

It is often assumed that software written in Common Lisp entails use of a garbage collector. It is our view that a garbage-creating programming style is what entails use of a garbage collector.

### Macros

The macro facility of Common Lisp is uniquely powerful, for two reasons. First, Common Lisp programs are represented, before compilation, as list structures that can be readily decomposed and transformed. Second, macros are defined as functions that can perform arbitrary computations to produce their expansions.

While macros in most languages are tools for performing textual replacements in source code, Common Lisp macros are allowed to perform more than mere textual substitutions. They are true programs. Their arguments are the parsed source code from the call sites. These arguments may be inspected, tested, decomposed, and in general manipulated in arbitrary ways by the macro-expanding function. The result of a macro execution is parsed code that replaces the macro call, and macro expansion is attempted again on this form. This can be repeated an arbitrary number of times, as long as the expansion eventually terminates. Since all of this execution occurs at compile time, use of macros imposes no run-time penalty.

Articles about Common Lisp often praise the representation of code as data, stating that the ability to create new functions at run time is its major benefit. However, we feel that a more important benefit comes from generating code at compile time—when the generated program code can still be compiled into efficient machine code.

Highly efficient implementations of data-abstracted operations can be achieved using Common Lisp

macros. In fact, Common Lisp macros make it practical to choose very complicated "multiple-case" implementations of data abstractions to handle special cases more efficiently.

A characteristic of some program facilities is that they perform different actions based on a control argument. In some cases, the actions called for by a control argument can be accomplished very quickly. Also, the actual parameters for these facilities are sometimes constants in the call sites. In our system, we have written many macros of this sort. Such macros examine their control argument, and if the argument is a constant, specific code for the selected operation is returned as the expansion of the macro. Often, just the data type of the control argument is needed to determine what the actual selected operation will be—again allowing compile time determination of the appropriate dispatching code. If a constant value or data type is not available for the control argument, then, as its expansion, the macro returns code that will dispatch on the control argument at run time.

If the dispatch can be optimized and the selected code is small, the returned macro expansion can contain operations that happen completely in line within the calling function. In languages that are not able to conditionalize macro expansions based on their arguments, dispatching is typically carried out at run time. For often-used facilities within inner-loop code, compile-time dispatching provides an important type of optimization.

Another important use of macros within our code is an implementation device for large-scale facilities. Macros are used as code generators that define elements of these large-scale facilities in terms of smaller components.

For example, the protocol hand-

# memory explicitly

lers for our networking tools are defined in this manner. Each kind of message handled within our protocol is defined in a separate call to a protocol definition macro. This macro expands into definitions of several functions and macros which implement the message handlers. It also expands into operations that register the names of these generated functions and macros in a data structure which is further used in our message handler dispatch function.

By organizing our implementation of message handlers into this abstraction, we have been able to implement them separately, but have provided an efficient, centralized dispatcher and automated registration of the handlers as a side effect of their definition. Most of the large facilities within our system have been built using similar constructs, all made possible by the strong macros of Common Lisp.

To achieve programmer-defined utilities of the same power in other languages usually requires the building of preprocessors that must parse source code files, perform their transformations, and then emit further text source files for compilation. Implementation of these preprocessors can in itself be a significant programming effort. In our view, the presence of a ready-made facility of this sort in Common Lisp is a distinguishing feature of the language.

### **Type Declarations for Compile-Time Optimizations**

Many Common Lisp operations are written to accept many different data types, dispatching to the specific code upon run-time determination of the actual type.

Numeric operations in Common Lisp are defined so that, for example, the “+” function takes an arbitrary number of arguments, where each argument can be of any type. The “+” function combines the arguments, using type contagion at run time to determine the type of the result. Common Lisp has often been criticized for this generality

because addition performed in this way is slow, compared to the type-specific machine instructions that implement addition.

The inefficiency of general operations has been addressed in Common Lisp by the inclusion of type declarations. Type-declared arithmetic can be compiled into direct machine instructions whose performance is equivalent to that achieved within other languages. Common Lisp provides two means of providing type information to the compiler. The first is through declarations associated with variables. The second technique is through the special form ‘THE’. By wrapping this form around another form, the type of the result of the wrapped form can be declared. Through the use of macros, we have built tools for using type-declared arithmetic throughout G2, using the ‘THE’ special form.

This approach to type declarations has given Common Lisp programmers an option that would not otherwise be available. In most languages, type declarations are required for the program to be compiled at all. If Common Lisp programmers want numeric operations to be executed as machine instructions, then they must provide type declarations, in one form or another. If numeric processing speed is not a primary concern within a piece of code, then the declarations may be left out, and the programmer is relieved of the burden of type declaration.

The technique of optimization through compile-time use of type declarations can be broadly applied in Common Lisp, though in our code we have used it mostly for numeric operations, array references, and dispatching to functions stored in arrays.

### **Program-Size Reduction for Application Delivery**

Common Lisp is a large language. It would be a remarkable program indeed which could justifiably use all of Common Lisp’s facilities. Most programs, even complex

ones, use only a fraction of what is available. In many other languages, commonly needed facilities are provided as libraries that may optionally be loaded into programs. In Common Lisp, all facilities are defined in the environment by default. Typical implementations require several megabytes of memory to hold the code associated with these facilities.

Early versions of Common Lisp systems required that all facilities remain available in final program images, even if they were not used by the application. This meant that even the simplest “Hello, world” programs would consume significant amounts of memory.

Utilities now exist within most Common Lisp implementations that can eliminate unused facilities from the environment. However, Common Lisp still differs from other languages, in that typically something special must be done to remove facilities from the final image, rather than to include them. These are often called *tree shakers*. The implication is that a program is the root of a tree and all utilities not used with (or somehow associated with) that tree are “shaken out” of the program before the final image is generated.

Another technique is to use C as an underlying implementation layer. Common Lisp source files are translated into C source files. The kernel facilities of Common Lisp are implemented as C libraries. When the translated C files are linked against the libraries of kernel functionality, only those utilities that are used are linked into the final program image.


### **Conclusions**

Historically, Lisp has been the language of choice for building sophisticated expert systems, and with good reason. Its broad set of built-in tools, its powerful macros, and its flexibility when writing numeric operations greatly improve programming productivity.

By using the memory management and other techniques de-



scribed, we have been able to take advantage of Common Lisp's powerful programming environment to realize efficient, real-time applications. (See [5] for a discussion of "soft real-time" applications for which G2 is particularly well suited and of a Space Shuttle monitoring application. Also see Rocky Stewart's sidebar in this issue describing an application at Biosphere II.)

Use of these techniques on standard hardware platforms has only become practical in recent years, as sophisticated implementations of Common Lisp have become commercially available on those platforms. By strictly controlling the styles of programming used within the language, we have been able to use Common Lisp as a practical language for large real-time application development and delivery. 

## References

1. Abelson, H., and Sussman, G.J., with Sussman, J. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., 1985, 491–503.
2. Baker, H.G. List processing in real time on a serial computer. *Commun. ACM* 21, 4 (Apr. 1978), 280–294.
3. Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (June, 1983), 419–429.
4. Moon, D.A. Garbage collection in a large lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Tex., Aug. 6–8). ACM, N.Y., pp. 235–246.
5. Muratore, J.F., Heindel, T.A., Murphy, T.B., Rasmussen, A.N., and McFarland, R.Z. Acquisition as mission control. *Commun. ACM* 33, 4 (Dec. 1990), 19–31.
6. Steele, G.L. Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford, Mass., 1990, 232–236.

**Categories and Subject Descriptors:** D3.3 [Programming Languages]: Language Constructs and Features—Dynamic storage management; I.2.5 [Artificial Intelligence]: Programming Languages and Software—Expert system tools and techniques

**General Terms:** Design, Languages, Performance

**Additional Key Words and Phrases:** Garbage collection, Lisp, macros, real-time programming, type declaration

## About the Authors:

**JAMES R. ALLARD** is manager of languages, interpreters, and compilers of Gensym Corporation. His research interests include the design and implementation of languages for the representation, simulation, and monitoring of dynamic systems.

**LOWELL B. HAWKINSON** is chair and CEO of Gensym Corporation. His research interests include expert systems, distributed on-line system archi-

ture, and natural languages.

**Authors' Present Address:** Gensym Corporation, 125 CambridgePark Dr., Cambridge, MA 02140. Email: jra@gensym.com, lh@gensym.com.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/91/0900-064 \$1.50

# Biosphere 2 Nerve System

~~~~~ Rocky L. Stewart ~~~~~

**B**iosphere 2 (Earth being Biosphere 1) is an experiment in closed system ecology. It is a steel and glass structure about the size of three football fields and has a volume of more than three million cubic feet. The purpose of the project is to demonstrate the viability of materially closed ecosystems—a sort of bioregenerative life support system—where water, air, and food are recycled. Later this year, eight people, called *Biospherians*, will be sealed inside Biosphere 2 for two years with only power and information being exchanged with the outside. A major part of this project is an expert system-based environmental control and monitoring system called the "Nerve System."

The ecosystems of Biosphere 2 are varied and complex. There are seven distinct biomes including a 30-foot-deep ocean, complete with waves, tides, and a coral reef including a rain forest with a 50-ft. mountain, waterfalls, and clouds. There is also a desert, marshland, savannah, an intensive agricultural biome where most of the food will be grown, and a habitat where the Biospherians will live during the experiment. Several thousand species of plants, animals, and insects will live in these biomes.

Due to the complexity of these eco-

systems, a sophisticated and reliable control and monitoring system is required to ensure the success and safety of both the ecosystems and the Biospherians. To fulfill this requirement, a Nerve System consisting of a broad-band network, several HP9000 work stations, and a control and monitoring hierarchy of expert systems was developed using G2, a Lisp-based, real-time, object-oriented expert system development environment from Gensym Corporation.

After several available traditional control systems were reviewed, G2 was chosen for the project because of its ease of use and its high degree of integration between rules, objects, and graphical displays. Also, G2 rules are interruptible, thus allowing the developer to control data acquisition and response times. Controlling the heating, ventilation and air-conditioning (HVAC) systems of Biosphere 2 is a typical real-time control problem. In the past, Lisp would not have been considered a candidate to solve this problem due to response time limitations imposed by garbage collection. G2, however, is carefully designed to eliminate the garbage collection of Lisp, thus reducing the chance of a delayed reaction at a critical moment.

The architecture of the nerve system consists of five major levels: