L -- A Common Lisp for Embedded Systems

Article · September 2004

Source: CiteSeer

CITATIONS
8

READS
3,352

2 authors, including:

Rodney A. Brooks
Massachusetts Institute of Technology
176 PUBLICATIONS **29,976** CITATIONS

SEE PROFILE

L – A Common Lisp for Embedded Systems

Rodney A. Brooks* and Charles Rosenberg
IS Robotics
Twin City Office Center, Suite 6
22 McGrath Highway
Somerville, MA 02143, USA

Abstract

A commercially available system has been developed which allows for the use of COMMON LISP in real time embedded control systems. The backbone of this system is a language called **L**. **L** is a subset of Common Lisp with multi-processing extensions. It is ideal for use in embedded systems with small computers. The system has a minimal memory footprint and can run on small processors. **L** contains both a runtime environment and an interpreter which runs on the target system and a cross compiler which runs on the host system. Making use of COMMON LISP's macro facilities, a special purpose language has been constructed on top of **L**, called **MARS** for MULTIPLE AGENCY REACTIVITY SYSTEM. This language is uniquely tailored for real time control applications. It allows for the spawning of many small processes which communicate with one another via a message passing approach. A graphical user interface debugging tool was also developed which allows real time monitoring and modification of values of variables on the target system by the host system. Underlying this system is an efficient real time operating system, called **VENUS**. The operating system is written in **C** for portability. The interface between **L** code and the hardware takes place via operating system calls. **L** accesses the **C** operating system calls via its foreign function interface. The first application for this system has been the programming and control of autonomous mobile robots. These robotic systems must process and react to their sensor systems in real time.

1 Introduction

This development effort was motivated by IS Robotics main business, the manufacture and programming of small autonomous robots for real world tasks. These robots have relatively small processors, a 16 MHz Motorola **68332** with 1 megabyte of RAM. Development for these systems is performed on a host computer and code is then installed on the target system. IS Robotics uses the behavior control approach (Brooks 1986) to program its robots. In this approach the control of the robot is broken up into many small parallel processes each of which operates on some sensor data and can control actuators on the robot. In earlier work we used what we called the BEHAVIOR LANGUAGE (Brooks 1990). It is written in COMMON LISP and produces a complete ROM image for a particular instance of a users program which is then installed (or downloaded to non-volatile RAM) on the robot. We decided we needed more flexibility and dynamic debugging however, so we decided to implement COMMON LISP (Steele Jr. 1990) directly on the robots. We wanted a system which allows us to dynamically redefine functions, test new code interactively, start and kill processes, and easily create extensions to the language, and tools which would reduce development time. To realize these goals we created a new system composed of a dialect of COMMON LISP (which we call **L**), a set of language extensions specific to real time control, and a special operating system.

2 VENUS

VENUS is a compact, efficient real time operating system. **VENUS** provides the low level interface to the target system hardware necessary to support **L**. **VENUS** is written primarily in **C** for portability. Only a small portion, the processor dependent part, of the operating system is written in assembly language. The low level target system support includes basic input/output and timing functions. The input/output functionality implements a protocol which utilizes a single bi-directional serial line to support five virtual bi-directional serial lines. A front end processor on the host decodes this protocol and exchanges packets with the embedded system. Currently these virtual serial lines are used to allow the embedded system to: access files on the host, provide a TTY interface for communication with the embedded system, and provide a means for data exchange with graphic debugging tools. The tim-

*Also a member of the faculty at the MIT Artificial Intelligence Laboratory.

ing functions implemented by **VENUS** provide access to the processor periodic interrupt timer for multithread scheduling.

VENUS is designed to be compact and modular; only those pieces of the operating system required for a specific application need be included in the run time kernel. For our typical Motorola **68332** implementation, the entire operating system occupies less than 32K bytes of memory. **VENUS** is also a dynamically linkable and modifiable system. The **C** functions which make up the most basic parts of the operating system can be changed dynamically, without rebuilding the entire system, as is the case with many other operating systems.

The **VENUS** system includes a low level monitor program which provides low level debugging support and the ability to modify the **VENUS** run time kernel. This monitor includes support for examining and modifying memory, linking in new OS kernel code, and debugging support for 'hard' crashes.

3 L

L is a completely new implementation of a subset of Common Lisp. It is carefully designed so that very small kernel subsets can be used to build Lisp images, e.g., ones without `eval`, `read`, etc., for small dedicated applications. We have successfully run **L** programs in load images as small as 10K bytes. More typically our full implementation of **L** now has a load image of almost 200K bytes, including robot specific libraries.

The main design goal in **L** has been simplicity. For this reason we have left out a number of aspects of Common Lisp, including its bulk.

Specifically we have excluded all the sequence functions, overloaded aritmetic, supplied-p arguments, extendable data structures (we do include automatically growing hash tables), the `EQ`/`EQL` distinction, `flet` and `labels`, parts of `format`, large parts of packages, preserving whitespace aspects of `read`, CLOS, the type system, the error condition system, and rationals, bignums, and trigonometry.

On the other hand **L** does include multiple values, strings, characters, arrays, a simplified but compatible package system, all the "ordinary" aspects of `format`, backquote and comma, `setf` etc., almost full Common Lisp lambda lists including optionals and keyword arguments, macros, an inspector, a debugger, `defstruct` (integrated with the inspector), `block`, `catch`, and `throw`, etc., full dynamic closures, a full lexical interpreter, floating point, fast garbage collection, and so on. The compiler runs in time linear in the size of an input expression, except in the presence of lexical closures. It nevertheless produces highly optimized code in most cases.

The **L** system is designed to be retargetable. So far we have run it on Macintoshes and on single chip Motorola **68332**'s. To retarget it is necessary to write a backend for the assembler, provide code templates for about 100 primitive operations, code sequences for about 20 relatively large but primtive operations (such as `cons`), code sequences for about 10 operations such as spreading multiple values on the stack and search the stack for a `throw` tag, and LAP macros for about 25 things such as procedure call, building `catch` frames, etc.

3.1 The Compiler

The compiler is an intellectual descendant of the S-1 compiler (Brooks, Gabriel & Steele Jr. 1982), and more recently the Lucid compiler (Brooks, Posner, McDonald, White, Benson & Gabriel 1986). It shares no code with either of these previous compilers however.

The design goals for the **L** compiler were

- it should produce reasonably efficient code
- compilation speed should be fast
- it should be target independent
- it should be easy to maintain

The compiler is written in Common Lisp, of course, and will probably run in **L** itself although that has not been tried at this point.

The compiler is rather simple and does not do register allocation for variables. It does open code many Common Lisp primitive operations however, and when compiling a tree of primitive operations will use registers to store multiple intermediate results.

The compiler has three phases: *alphatization*, *analysis*, and *generation*.

During alphatization everything is syntactically checked and a number of source to source transformations are made. These include quotation of self evaluating literals, macro expansions, and compiler macro expansions. Most of the Common Lisp control forms are implemented as compiler macros which take precendence over macros defined in the host Lisp—in this way the user can use the host Lisp macro system and the compiler can still use its own definitions of macro expansions for such things as `dotimes`. During alphatization the code tree is walked exactly once.

During the analysis phase a number of further source to source transformations are made that are target specific—for instance simple vector references and structure references might all get mapped to a common underlying reference mechanism at this point, or not, depending on the particular memory model being used for the target. Other transformations carried out here include constant folding in arithmetic expressions, and

even reordering of arithmetic operations. All these transforms take alphatized source and return alphatized source. At the same time as the code tree is being walked, in a single pass, for these transformations an analysis tree is formed. It notes side effects, read/write status of lexical variables, register usage requirements, stack height, pointers up the tree to lexical variable, block, and tag owners, etc. Also during this phase, analysis of closures is carried out, and source to source transforms are made so that shared lexical variables refer to the right data structures, with optimizations for read only lexical variables, etc. If closures are present it is sometimes necessary to re-invoke analysis on a subtree for a second time.

As analysis proceeds, the compiler looks at the requirements for open coding primitive procedures directly in machine code. The following are the descriptions of fixnum + and - for the Motorola **68332**.

```
(defun code-68k-adder (x y ops)
  (cond ((and (consp y)
              (eq (first y) 'quote))
         (if (<= 1 (second y) 2)
           `((,(third ops)
              ,(ash (second y) 2) ,x))
           `((,(second ops) ,y ,x))))
        (t `((,(first ops) ,y ,x)))))

(deflprim +
  :args ((x dreg) (y any))
  :code (code-68k-adder x y '(add addi addq))
  :loc x)

(deflprim -
  :args ((x dreg) (y any))
  :code (code-68k-adder x y '(sub subi subq))
  :loc x)
```

Note that these descriptions include requirements on where the arguments should be (the first argument should be in a data register), and the actual code generator does some optimizations when possible. If the compiler has a tree of primitive operations that require more than the available number of registers, it may introduce a lexical variable to save an intermediate value and rebuild the analysis tree and source appropriately. The descriptions above assume that earlier phases of the compiler have guaranteed that all calls to these two procedures are dyadic, and that constants have been pushed rightward wherever possible.

The following are additional primitive operations, out of a total of about 100 altogether.

```
(deflprim eq
  :args ((x areg) (y any))
  :code `((cmpa ,y ,x))
  :rep :flag
  :loc 'eq)

(deflprim car
  :args ((x areg))
  :loc `(ref-cons ,x 0))

(deflprim cdr
  :args ((x areg))
  :loc `(ref-cons ,x 4))

(deflprim set-car
  :args ((x areg) (y any))
  :code `((move ,y (ref-cons ,x 0)))
  :effects (make-rplac-effects)
  :loc y)

(deflprim second
  :args ((x areg))
  :code `((move (ref-cons ,x 4) ,x))
  :loc `(ref-cons ,x 0))
```

Note that some operations produce a value, while others, like `eq` produce a condition code. A coercion process in generation must convert such representations if they are not appropriate for the particular place in which the operation is used. Note also that `set-car` has its side effects noted—this means that generation can not optimize it away if its value is not used. Also `car` and `cdr` produce no code at all. They describe how given a cons cell in an address register, the `car` and `cdr` can be accessed by an addressing mode—expressed with the LAP (Lisp assembler program) macro `ref-cons` shown below:

```
(deflapoperand ref-cons (reg offset)
  `(areg-indirect-disp
    ,reg
    ,(+ offset
        (ltarget (- 8 *lptr-cons*)))))
```

The last phase of the compiler is code generation. For special forms, procedure calls, etc., it produces a series of calls to LAP macros which essentially describe an abstract machine. For primitive operations it uses the code templates as described above to generate specific machine instructions to implement them. Some primitive operations, such as floating point arithmetic, or storage allocation are too long to generate each time they are invoked. So there is a library of *fastops*, handcoded machine code subroutines to implement these, and the compiler generates simple subroutine calls to these where appropriate.

Except in the case of closure analysis, all three phases of the compiler are single pass, and thus rather efficient. The compiler has a simple form and is not weighed down

by special purpose optimizations. The code it produces for standard sorts of non-arithmetic things when run on a Macintosh is much more efficient than the code produced by the MACINTOSH COMMON LISP compiler (it would be unfair to compare arithmetic operations as **L** does not handle generic arithmetic).

3.2 Arithmetic

There are two types of arithmetic supported in **L**. Fixnum arithmetic and small floating point.

The arithmetic operators `+`, `*`, etc., only support fixnums. The Motorola **68332** implementation of fixnums is 30 bits signed. The fixnum operators act just like COMMON LISP arithmetic operators given fixnums, except that in **L** there can never be an overflow into bignums. Any overflows are ignored and arithmetic is essentially mod2^{30}. Thus fixnum arithmetic never causes any consing in **L**.

Besides normal arithmetic, the usual logical operators are also supported on fixnums.

To use floating point, other operators `+$`, `*$`, etc., must be used. This is similar to the situation in MACLISP except that **L** does not include any additional generic operators. The Motorola **68332** implementation of floating point numbers is 28 bits long, including 8 bits of signed exponent in excess 128 format, 19 bits of mantissa (plus a hidden bit), and 1 bit for the sign of the number. This floating point format is designed to fit in an immediate data pointer, so there is never any consing caused by floating point arithmetic. The drawback is that floating point arithmetic is implemented in software (as fastops), so they are slow, but there is no floating point hardware on the Motorola **68332** in any case.

Floating point underflow causes an error, but a macro `without-floating-underflow-traps` is provided to suppress these if desired[1].

3.3 Packages

The package system in **L** is simplicity itself, but nevertheless it provides the standard functionality of providing provate namespaces and it appears compatible with COMMON LISP packages as long as one does not poke too hard.

In **L** every symbol in every package is external and every symbol is in precisely one package. Packages can use other packages. When it comes time to resolve the name of a symbol in a package (e.g., in the package bound by `*package*` during `read`), that package is checked first. If the name is found there then that is the result. Otherwise all the packages used by the first package, and so on in a depth first search are checked until a symbol with the same name is found. If none are found, then a new symbol is interned in the original package. This system is very easy to implement. By restricting the total number of packages to be no more than 16,384 the total storage requirement per symbol is 14 bits, plus one 32 bit entry in a special package hash table which simply records the presence or absence of a symbol.

When **L** starts up there are five packages present: `"SYSTEM"`, `"KEYWORD"`, `"LIB"`, `"L"`, and `"USER"`. The `"SYSTEM"` package contains all the system code. The `"USER"` package is empty—both these packages use both the `"L"`, and `"LIB"` packages. The `"L"` package is where all the COMMON LISP symbols reside. The `"LIB"` package includes all entry points to robot specific libraries that have been built on top of **L** and **C** for the particular robot.

3.4 Garbage Collection and Storage Management

The garbage collection system used in **L** is a very simple and elegant stop and copy collector. This interferes with real time requirements placed on **L**, so the rest of **L** is designed so that for real time programming there is no need to invoke `cons`. This includes the design of the arithmetic system and the design of **MARS**. The idea is that the user loads everything (or has it autoloaded) during which time consing takes place, but thereafter, unless in debugging mode, there is no need for further consing.

The storage arrangement in **L** consists of six spaces:

pure code	ROM	
pure data	ROM	
static data	RAM	GC scans
user code	RAM	
from space	RAM	
to space	RAM	GC scans

The system code is all in the pure code space. String names of system symbols, keyword symbols themselves used by system code, the closure objects (including pointers to all literals) for all system procedures, and minor other constant data structures are kept in pure data. It has no pointers to from or to spaces, but may have pointers within itself and to the static space. The static space provides pre-allocated data structures which are guaranteed to remain in place. These include all other structures that are present when **L** starts up, such as symbols, packages, etc. The multithreading system allocates all stacks in this space so that the garbage collector does not have to deal with relocating an active stack. Users can also request that data be allocated here by using the macro `with-static-consing`. The trade off in using this space is that the garbage collector never has to copy things from here, but it does have to

[1] Note that this was proposed to X3J13 in June 1989 to be added to COMMON LISP but the proposal was rejected.

scan them even if they do become garbage, and they are never collected.

User loaded machine code goes into the user code space. This space is not garbage collected and can potentially fill up just from repeatedly reloading the same compiled file.

The from and to spaces are the two half spaces for user data. Their roles are swapped at each garbage collection.

3.5 Multithreading

The multithreading system in **L** is unique—it is not a subset or superset of anything in COMMON LISP. It is a polling pre-emptive scheduler with the abiltity for processes to put themselves to sleep for a given duration.

Since the pre-emptive scheduler uses polling, it only pre-empts if code was specifically compiled for multithreading—this is the default and only about 20 critical procedures in the whole **L** system are not compiled for polling.

A thread only polls, and thus is only ever suspended, on procedure entry right after all argument parsing has been done. This includes catching cases of self tail recursive calls. Many threads have their own stack, so on entry to a multithreading compiled procedure, a check is done to see if it looks like there is enough stack left, and if not an error is signalled.

When **L** starts up it is not in multithreaded mode. But it is possible to evaluate a procedure called `start-multi-processing`, which spawns a new `read-eval-print` loop, and schedules it to be run repeatedly. The scheduler runs in the normal Lisp stack. The scheduler is based on an internal clock, gotten with `get-internal-real-time`. It uses an efficient data structure to reschedule when to next run each thread as it requests as it is suspended.

A procedure called `spawn` can be used to start up additional threads. There are two sorts of threads.

3.5.1 Light weight threads

Lightweight threads do not have their own stack and are run from the normal Lisp stack. Whenever such a thread is invoked, some specific procedure is called and it runs until it returns—never polling. Its return value is how many internal time units before it wants to be run again.

3.5.2 Heavy weight threads

Heavy weight threads have their own stack. They are spawned as a procedure to be called, but it is expected that that procedure will never return—the thread is killed if it does. During every procedure entry within the thread the system polls a register maintained by **VENUS** saying how much time is left in the slice of the current thread. When it is zero or less the thread relinquuishes control to the scheduler which reschedules it some small time interval later. Threads can also voluntarily reschedule themselves by calling `sleep`, or variants thereon.

Context switching of heavy weight processes can be expensive if they are suspended with special variables rebound—their stack needs to be traversed to undo these binds before switching to another thread, and then be traversed again to put them back when restarting this thread.

3.6 Debugging

The standard debugging tools provided in **L** are a text based inspector and a text based stack walker.

The inspector lets the user examine the full internals of any data structure. It is integrated with `defstruct` so that slot names are displayed for structures whose `defstruct` has been loaded into the current environment.

The stackwalker lets the user move up and down any suspended stack (including those of other threads that have signalled errors), examing the stack frames and invoking the inspector on any slots. There are also facilities for looking at raw machine memory which is sometimes useful in debugging sensors and actuators.

Machine traps are caught in the **VENUS** operating system and handed back to **L** so that they can be displayed within the context of the Lisp computation which caused the problem.

Other useful tools are `apropos` and `who-refers?`. The first is standard COMMON LISP and uses `do-all-symbols` to find symbols whose name matches a partial string. The second is an **L** construct which searchs all the procedure objects in the system to find all literal references to an argument symbol.

3.7 The Motorola 68332 Implementation

For 68000 based architectures we have used a lowtag scheme to represent lisp objects. Here the bottom three bits determine the gross type of a pointer according to the following scheme:

Bits	Pointer
000	even fixnum
001	cons cell
010	gc forwarding pointer
011	object with header
100	odd fixnum
101	symbol
110	reserved
111	immediate data item

All of the even data types can also be pointers to objects created by **C**, and there only needs to be a little extra checking in the garbage collector to disambiguate forwarding pointers.

Besides fixnums, immediate data objects include characters, floating point numbers, unbound markers, various stack markers for special binds, etc., and headers for objects in the heap.

All objects in the heap are aligned at eight byte boundaries, and pointers to them point to within the previous eight bytes. Symbols and cons cells have no headers—all their elements are simply scanned by the garbage collector. Other objects such as strings, arrays, procedure objects, etc., have headers which say what their size is and how the garbage collector should treat them as it scans through the heap.

4 MARS

The Multiple Agency Reactivity System, or **MARS**, is a language for programming multiple concurrent agents. It is embeded in **L**, and as such, all of that subset of Common Lisp is available for use anywhere within programs.

MARS is a method for defining many asynchronous parallel processes which run in a single **L** heap. Groups of these processes are called *assemblages* which share a lexical environment and some number of *ports* which are message passing interfaces between assemblages. Ports are bi-directional, but a port from one assemblage can only be connected uni-directionally to a port of another assemblage. Connections of ports involve one deep buffers and message arrival flags. New instances of assemblages can be spawned dynamically at runtime. Assemblages also can be dynamically killed. Connections can be dynamically made and broken.

There is also a graphical user interface which allows for dynamic monitoring and altering of all port values. These monitors can be added dynamically at run time to any port. The graphics run on a host machine, such as a Macintosh—there is a standard interface envionment that runs in Macintosh Common Lisp.

MARS is somewhat like the earlier Behavior Language, a method for layering behavior producing programs, one after the other, in a system connecting sensors to actuators. In fact there is a fairly straightforward translation from Behavior Language programs to **MARS** programs. But **MARS** programs can be much richer as they have access to all of Common Lisp. In addition, all linking is completely dynamic, and all housekeeping is invisibly taken care of so that assemblages (roughly corresponding to *behaviors* in Behavior Language) can be redefined at run time, new assemblages can be created, and old ones killed.

The graphical user interface is supported via one of the extra virtual serial lines implemented in **L** and **VENUS**. A packet protocol is built on top of this serial line (which itself is built on top of a packet protocol which is built on top of an actual serial line). Besides handling graphics traffic this system is also used to implement other interactions between the front end (host machine), and **L**.

Great care has been taken in the implementation of both **MARS** itself and the graphical user interface to make debugging easy. Everything is dynamically linked so that new versions of subsystems can be recompiled and reloaded onto an existing working system, without any interruptions. All the graphic windows resize themselves should it be necessary when this happens, and things that have had monitors attached to them stay monitored even when they are completely redefined by overloading a new version of the code–even to the point of handling changes in underlying representations in the middle of display option choice back on the host computer. All this is a marked improvement over the earlier Behavior Language.

5 Conclusion

Our goal was to create a development system which would allow us to have all of the advantages of Common Lisp on a small embedded computer system. The **L** language was developed to serve this purpose. It is a highly efficient subset of Common Lisp with multithreading extensions and a cross compiler. It can run on the relatively small processors often used in embedded systems. To improve programming efficiency, a language, named **MARS**, was built on top of **L**. This package greatly simplifies the creation and debugging of large numbers of inter-communicating processes. IS Robotics is committed to the continuing development and improvement of the **L** system and its commercial viability. We firmly believe **L** to be a superior solution for creating complex real time embedded control systems.

References

Brooks, R. A. (1986), 'A Robust Layered Control System for a Mobile Robot', *IEEE Journal of Robotics and Automation* **RA-2**, 14–23.

Brooks, R. A. (1990), The Behavior Language User's Guide, Memo 1227, Massachusetts Institute of TechnologyArtificial Intelligence Lab, Cambridge, Massachusetts.

Brooks, R. A., Gabriel, R. P. & Steele Jr., G. L. (1982), An Optimizing Compiler for Lexically Scoped Lisp,

in 'Proceedings of the 1982 Symposium on Compiler Construction, ACM SIGPLAN', Boston, Massachusetts, pp. 261–275. Published as ACM SIGPLAN *Notices 17,* 6 (June 1982).

Brooks, R. A., Posner, D. B., McDonald, J. L., White, J. L., Benson, E. & Gabriel, R. P. (1986), Design of An Optimizing Dynamically Retargetable Compiler for Common Lisp, *in* 'Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming', Cambridge, Massachusetts, pp. 67–85.

Steele Jr., G. L. (1990), *Common Lisp, The Language*, second edn, Digital Press.