The GCL ANSI Common Lisp Test Suite

Paul F. Dietz*

Abstract

I describe the conformance test suite for ANSI Common Lisp distributed as part of GNU Common Lisp (GCL). The test suite includes more than 20,000 individual tests, as well as random test generators for exercising specific parts of Common Lisp implementations, and has revealed many conformance bugs in all implementations on which it has been run.

1 Introduction

One of the strengths of Common Lisp is the existence of a large, detailed standard specifying the behavior of conforming implementations. The value of the standard to users is enhanced when they can be confident that implementations that purport to conform actually do.

In the 1990s I found substantial numbers of conformance bugs in many Lisp implementations. As a result, I decided to build a comprehensive functional test suite for Common Lisp. The goals of the effort were, in no particular order:

- To thoroughly familiarize myself with the standard.
- To provide a tool to locate conformance problems in CL implementations, both commercial and free.
- To enable implementors to improve CL implementations while maintaining conformance.
- To explore the standard itself for ambiguities, unintended consequences, and other problems.
- To explore different testing strategies.

I deliberately did not design the test suite to measure or rank conformance of Lisp implementations. For this reason, I will not here report the overall score of any implementation.

I decided to locate the test suite in the GCL development tree for two reasons. First, its development team had a goal of making GCL more ANSI compliant, and tests would assist there. Secondly, the GCL CVS tree is easily publicly accessible[1], so any developers or users of Common Lisp implementations would have easy access to it.

The test suite was constructed over the period from 1998 to 2005, with most of the work done in 2002 to 2004. As of 24 May 2005, the test suite contains over 20,000 tests.

The test suite is based on a version of the ANSI Common Lisp specification (ANSI/INCITS 226-1994, formerly ANSI X3.226-1994) that was made publicly available by Harlequin (now LispWorks) in hyperlinked form in 1996 [9].

Table 1 contains a list of Lisp implementations on which I am aware the test suite has been run.

*Motorola Global Software Group, 1303 E. Algonquin Road, Annex 2, Schaumburg, IL 60196. paul.f.dietz@motorola.com

[1] See `http://savannah.gnu.org/projects/gcl/`

Implementation	Hardware Platforms
GNU Common Lisp	All debian platforms
GNU CLISP	x86
CMUCL	x86, Sparc
SBCL	x86, x86-64, Sparc, MIPS, Alpha, PowerPC
Allegro CL (6.2, 7)	x86, Sparc, PowerPC
LispWorks (4.3)	x86
OpenMCL	PowerPC
ABCL	x86 (JVM)
ECL	x86

Table 1: Implementations Tested

```
(deftest let.17
  (let ((x :bad))
    (declare (special x))
    (let ((x :good)) ;; lexical binding
      (let ((y x))
        (declare (special x)) ;; free declaration
        y)))
  :good)
```

Figure 1: Example of a test

2 Infrastructure

The test suite uses Waters' RT package [11]. This package provides a simple interface for defining tests. In its original form, tests are defined with a name (typically a symbol or string), a form to be evaluated, and zero or more expected values. The test passes if the form evaluates to the specified number of values, and those values are as specified. See figure 1 for an example from the test suite:

As the test suite evolved RT was extended. Features added include:

- Error conditions raised by tests may be trapped.
- Tests may optionally be executed by wrapping the form to be evaluated in a lambda form, compiling it, and calling the compiled code. This makes sense for testing Lisp itself, but would not be useful for testing Lisp applications.
- A subset of the tests can be run repeatedly, in random order, a style of testing called *Repeated Random Regression* by Kaner, Bond and McGee [2][2]
- Notes may be attached to tests, and these notes used to turn off groups of tests.
- Tests can be marked as being expected to fail. Unexpected failures are reported separately.

[2]This was previously called 'Extended Random Regression'; McGee renamed it to avoid the confusing acronym.

Section of CLHS	Size (Bytes)	Number of Tests
Arrays	212623	1109
Characters	38655	256
Conditions	71250	658
Cons	264208	1816
Data & Control Flow	185973	1217
Environment	51110	206
Eval/Compile	41638	234
Files	26375	87
Hash Tables	38752	158
Iteration	98339	767
Numbers	290991	1382
Objects	283549	774
Packages	162203	493
Pathnames	47100	215
Printer	454314	2364
Reader	101662	663
Sequences	562210	3219
Streams	165956	796
Strings	83982	415
Structures	46271	1366
Symbols	106063	1141
System Construction	16909	77
Types	104804	599
Misc	291883	679
Infrastructure	115090	
Random Testers	190575	
Total	4052485	20702

Table 2: Sizes of Parts of the Test Suite

3 Functional Tests

The bulk of the test suite consists of functional tests derived from specific parts of the ANSI specification. Typically, for each standardized operator there is a file *operator*.lsp containing tests for that operator. This provides a crude form of traceability. There are exceptions to this naming convention, and many tests that test more than one operator are located somewhat arbitrarily. Table 2 shows the number and size of tests for each section of the ANSI specification.

Individual tests vary widely in power. Some are as simple as a test that `(CAR NIL)` is `NIL`. Others are more involved. For example, `TYPES.9` checks that `SUBTYPEP` is transitive on a large collection of built-in types.

The time required to run the test suite depends on the implementation, but it is not excessive on modern hardware. SBCL 0.9.0.41 on a machine with 2 GHz 64 bit AMD processor, for example, runs the test suite in under eight minutes.

Error tests have been written where the error behavior is specified by the standard. This includes spec-

ifications in the 'Exceptional Situations' sections for operator dictionary entries, as well as tests for calls to functions with too few or too many arguments, keyword parameter errors, and violations of the first paragraph of CLHS section 14.1.2.3. When type errors are specified or when the CLHS requires that some operator have a well-defined meaning on any Lisp value, the tests iterate over a set of precomputed Lisp objects called the 'universe' that contains representatives of all standardized Lisp classes. In some cases a subset of this universe is used, for efficiency reasons.

There are some rules that perform random input testing. This testing technique is described more fully in the next section. Other tests are themselves deterministic, but are the product of one of the suite's high volume random test harnesses. The 'Misc' entry in table 2 refers to these randomly generated tests. Each of these tests caused a failure in at least one implementation.

Inevitably, bugs have appeared in the test suite. Running the test suite on multiple implementations (see table 1) exposes most problems. If a test fails in most of them, it is likely (but not certain) that the test is flawed. Feedback from implementors has also been invaluable, and is deeply appreciated. In some cases, when it has not been possible to agree on the proper interpretation of the standard, I've added a note to the set of disputed tests so they can be disabled as a group. This is in keeping with the purpose of the test suite – to help implementors, not judge implementations.

4 Random Testing

Random testing (more properly, random-input testing) is a standard technique in the testing of hardware systems. However, it has been the subject of controversy in the software testing community for more than two decades. Myers [8] called it "Probably the poorest ... methodology of all". This assessment presumes that the cost of executing tests and checking their results for validity dominates the cost of constructing the tests. If test inputs can be constructed and results checked automatically, it may be very cost-effective to generate and execute many lower quality tests. Kaner et al. call this High Volume Automated Testing [2].

Duran and Ntafos [3] report favorably on the ability of random testing to find relatively subtle bugs without a great deal of effort. Random testing has been used to test Unix utilities (so-called 'fuzz testing') [7], database systems [10], and C compilers [6, 5, 4]. Bach and Schroeder [1] report that random input testing compares well with the ability of the popular All-Pairs testing technique at actually finding bugs.

Random input testing provides a powerful means of testing algebraic properties of systems. Common Lisp has many instances where such properties can be checked, and the test suite tests many of them. Random testing is used to test numeric operators, type operators, the compiler, some sequence operators, and the readability of objects printed in 'print readably' mode.

One criticism of random testing is its irreproducibility. With care, this needn't be a problem. If a random failure is sufficiently frequent, it can be reproduced with high probability by simply running a randomized test again. Tests can also be designed so that on failure, they print sufficient information so that a non-randomized test can be constructed exercising the bug. Most of the randomized tests in the test suite have this property.

4.1 Compiler Tests

Efficiency of compiled code has long been one of Common Lisp's strengths. Implementations have been touted as in some cases approaching the speed of statically typed languages. Achieving this efficiency places strong demands on Lisp compilers. A sufficiently smart compiler needs a sufficiently smart test suite.

Compilers (and Lisp compilers in particular) are an ideal target for random input testing. Inputs may have many parts that interact in the compiler in unpredictable ways. Because the language has a well-defined

semantics, it is easy to generate related, but different, forms that should yield the same result (thereby providing a test oracle.)

The Random Tester performs the following steps. For some input parameters n and s (each positive integers):

1. Produce a list of n symbols that will be the parameters of a lambda expression. These parameters will have integer values.

2. Produce a list of n finite integer subrange types. These will be the types of the lambda parameters. The endpoints of these types are not uniformly distributed, but instead follow an approximately exponential distribution, preferring small integers over larger ones. Integers close in absolute value to integer powers of 2 are also overrepresented.

3. Generate a random conforming Lisp form of 'size' approximately s containing (mostly) integer-valued forms. The parameters from step 1 occur as free variables.

4. From this form, construct two lambda forms. In the first, the lambda parameters are declared to have their integer types, and random `OPTIMIZE` settings are included. In the second, a different set of `OPTIMIZE` settings is declared, and all the standardized Lisp functions that occur in the form are declared `NOTINLINE`. The goal here is to attempt to make optimizations work differently on the two forms.

5. For each lambda form, its value on each set of inputs is computed. This is done either by compiling the lambda form and calling it on the inputs, or by evaling forms in which the lambda form is the `CAR` and the argument list the `CDR`.

6. A failure occurs if any call to the compiler or evaluator signals an error, or if the two lambda forms yield different results on any of the inputs.

This procedure very quickly – within seconds – found failures in every Lisp implementation on which it was tried. Failures included assertion failures in the compiler, type errors, differing return values, code that caused segmentation faults, and in some cases code that crashed the Lisps entirely. Most of the 679 'Misc' tests in table 2 were produced by this tester; each represents a failure in one or more implementations.

Generating failing tests was easy, but minimizing them was tedious and time consuming. I therefore wrote a pruner that repeatedly tries to simplify a failing random form, replacing integer-valued subforms with simpler ones, until no substitution preserving failure exists. In most cases, this greatly reduced the size of the failing form. Others have previously observed that bug-exposing random inputs can often be automatically simplified [12, 6]. The desire to be able to automatically simplify the failing forms constrained the tester; I will discuss this problem later in section 6.

Table 3 contains a list of the fourteen compiler bugs detected by the random tester in GNU CLISP. Roughly 200 million iterations of the random tester were executed to find these bugs, using a single 1.2 GHz Athlon XP+ workstation running intermittently over a period of months. All these bugs have been fixed (in CVS) and CLISP now fails only when the random forms produce bignum values that exceed CLISP's internal limit.

The greatest obstacle to using the random tester is the presence of unfixed, high probability bugs. If an implementation has such a bug, it will generate many useless hits that will conceal lower probability bugs.

4.2 Types and Compilation

Type inference and type-based specialization of built-in operators is a vital part of any high performance Lisp compiler for stock hardware, so it makes sense to focus testing effort on it. The test suite contains a facility

Sourceforge Bug #	Type of Bug	Description
813119	C	Simplification of conditional forms
842910	C	Simplification of conditional forms
842912	R	Incorrect generated code
842913	R	Incorrect generated code
858011	C	Compiler didn't handle implicit block in `FLET`
858658	R	Incorrect code for `UNWIND-PROTECT` and multiple values
860052	C	Involving `RETURN-FROM` and `MULTIPLE-VALUE-PROG1`.
864220	C	Integer tags in tagbody forms.
864479	C	Compiler bug in stack analysis.
866282	V	Incorrect value computed due to erroneous side effect analysis in compiler on special variables
874859	R	Stack mixup causing catch tag to be returned.
889037	V	Bug involving nested `LABELS`, `UNWIND-PROTECT`, `DOTIMES` forms.
890138	R	Incorrect bytecodes for `CASE`, crashing the Lisp.
1167991	C	Simplification of conditional forms.

Legend: C Condition thrown by the compiler (assert or type check failure.)
R Condition thrown at runtime (incorrectly compiled code).
V Incorrect value returned by compiled code.

Table 3: Compiler bugs found in GNU CLISP by Random Tester

for generating random inputs for operators and compiling them with appropriate randomly generated type annotations, then checking if the result matches that from an unoptimized version of the operator.

As an example, the operator `ISQRT` had this bug in one commercial implementation:

```
(compile nil '(lambda (x) (declare (type (member 4 -1) x)
                                   (optimize speed (safety 1)))
                          (isqrt x)))
==> Error: -1 is illegal argument to isqrt
```

Amusingly, the bug occurs only when the negative integer is the second item in the `MEMBER` list. The test that found this bug is succinctly defined via a macro:

```
(def-type-prop-test isqrt 'isqrt '((integer 0)) 1)
```

The function to be compiled can be generated in such a way that it stores the result value into an array specialized to a type that contains the expected value. This is intended to allow the result value to remain unboxed.

The general random testing framework of section 4.1 is also useful for testing type-based compiler optimizations, with two drawbacks: it currently only handles integer operators, and it is less efficient than the more focused tests. Even so, it was used to improve unboxed arithmetic in several implementations (SBCL, CMUCL, GCL, ABCL).

4.3 `SUBTYPEP` Testing

The test suite uses the algebraic properties of the `SUBTYPEP` function in both deterministic and randomized tests. For example, if `T1` is known to be a subtype of `T2`, we can also check:

```
(subtypep '(not t2) '(not t1))
(subtypep '(and t1 (not t2)) nil)
(subtypep '(or (not t1) t2) t)
```

The generator/pruner approach of the compiler random tester was applied to testing SUBTYPEP. Random types were generated and, if one was a subtype of the other, the three alternative formulas were also tested. If any return the two values (false, true), a failure has been found.

Christophe Rhodes used feedback from this tester to fix logic and performance bugs in SBCL's SUBTYPEP implementation. The handling of CONS types is particularly interesting, since deciding the subtype relationship in the presence of cons types is NP-hard. At least one implementation's SUBTYPEP will run wild on moderately complicated cons types, consuming large amounts of memory before aborting.

4.4 Repeated Random Regression

As mentioned earlier, RRR is a technique for executing tests in an extended random sequence, in order to flush out interaction bugs and slow corruption problems. As an experiment, RT was extended to support RRR on subsets of the tests. The main result was to find many unwanted dependencies in the test suite, particularly among the package tests. These dependencies had not surfaced when the tests had been run in their normal order.

After fixing these problems, RRR did find one CLOS bug in CLISP, involving interaction between generic functions and class redefinitions. The bug was localized by bisecting the set of tests being run until a minimal core had been found, then minimizing the sequence of invocations of those tests. If more bugs of this kind are found it may be worthwhile to add a delta debugging [12] facility to perform automatic test minimization.

In Lisps that support preemptively scheduled threads, it would be interesting to use RRR with subsets of the tests that lack global side effects. The tests would be run in two or more threads at once in order to find thread safety problems.

5 Issues with the ANSI Common Lisp Specification

Building the test suite involved going over the standard in detail. Many points were unclear, ambiguous, or contradictory; some parts of the standard proved difficult to test in a portable way. This section describes some of these findings.

See 'Proposed ANSI Revisions and Clarifications' on http://www.cliki.net/ for a more complete list that includes issues arising from the test suite.

5.1 Testability

Some parts of the standard proved difficult to test in a completely conforming way. The specification of pathnames, for example, was difficult to test. The suite has assumed that UNIX-like filenames are legal as physical pathnames.

Floating point operators presented problems. The standard does not specify the accuracy of floating point computations, even if it does specify a minimum precision for each of the standardized float types. [3] Some implementations have accuracy that varies depending on the details of compilation; in particular, boxed values

[3] The standard does specify a feature indicating the implementation purports to conform to the IEEE Standard for Binary Floating Point Arithmetic (ANSI/IEEE Std 754-1985); this suite does not test this.

may be constrained to 64 bits while unboxed values in machine registers may have additional 'hidden' bits. These differences make differential testing challenging.

The Objects chapter contains interfaces that are intended to be used with the Metaobject Protocol (MOP). Since the MOP is not part of the standard, some of these cannot be tested. For example, there is apparently no conforming way to obtain an instance of class `METHOD-COMBINATION`, or to produce any subclass of `GENERIC-FUNCTION` except for `STANDARD-GENERIC-FUNCTION`.

5.2 Unintended Consequences

There seem to be many issues associated with Common Lisp's type system. One example is the `TYPE-OF` function. According to the standard, this function has the property that

> For any object that is an element of some built-in type: [...] the type returned is a recognizable subtype of that built-in type.

A *built-in* type is defined to be

> built-in type *n.* one of the types in Figure 4-2.

Figure 4-2 of the standard contains `UNSIGNED-BYTE`, the type of nonnegative integers. These constraints imply that `TYPE-OF` can never return `FIXNUM` or `BIGNUM` for any nonnegative integer, since neither of those types is a subtype of `UNSIGNED-BYTE`.

A more serious set of problems involves `UPGRADED-ARRAY-ELEMENT-TYPE`. [4] This function (from types to types) is specified to satisfy these two axioms for all types T_1 and T_2:

$$T_1 \subseteq UAET(T_1)$$

and

$$T_1 \subseteq T_2 \Longrightarrow UAET(T_1) \subseteq UAET(T_2)$$

A type T_1 is a *specialized array element type* if $T_1 = UAET(T_1)$. These axioms imply:

Theorem 1 *If two types T_1 and T_2 are specialized array element types, then so is $T_1 \cap T_2$.*

This theorem has a number of unpleasant consequences. For example, if `(UNSIGNED-BYTE 16)` and `(SIGNED-BYTE 16)` are specialized array element types, then so must be `(UNSIGNED-BYTE 15)`. Even worse, since `BIT` and `CHARACTER` are required to be specialized array element types, and since they are disjoint, then `NIL`, the empty type, must also be a specialized array element type. Topping all this off, note that

> A string is a specialized vector whose elements are of type character or a subtype of type character. (CLHS page for `STRING`)

Since `NIL` is a subtype of `CHARACTER`, a vector with array element type `NIL` is a string. It is impossible for a conforming implementation to have only a single representation of strings.[5]

[4] I ignore the issue that, strictly speaking, `UPGRADED-ARRAY-ELEMENT-TYPE` is either an identity function or is not computable, since as defined it must work on `SATISFIES` types.

[5] But since 'nil strings' can never be accessed, it's acceptable in non-safe code to just assume string accesses are to some other string representation. The SBCL implementors took advantage of this when using nil strings as a stepping stone to Unicode support.

6 Directions For Future Work

The test suite still has a few areas that are not sufficiently tested. Setf expanders need more testing, as do logical pathnames and file compilation. Floating point functions are inadequately tested. As mentioned earlier, it isn't clear what precision is expected of these functions, but perhaps tests can be written that check if the error is too large (in some sufficiently useful sense.)

The random compiler tester, as implemented, is constrained to generate forms that remain conforming as they are simplified. This limits the use of certain operators that do not take the entire set of integers as their arguments. For example, `ISQRT` appears only in forms like `(ISQRT (ABS ...))`, and this pattern is preserved during pruning. The forms also make very limited use of non-numeric types.

More sophisticated random tester could avoid these limitations. One approach would be to randomly generate trees from which Lisp forms could be produced, but that also carry along information that would enable pruning to be done more intelligently. Another approach would be to check each pruned form for validity on the set of chosen random inputs by doing a trial run with all operators replaced by special versions that always check for illegal behaviors. I intend to explore both options.

The test suite has been written mostly as a 'black box' suite (aside from the randomly generated Misc tests). It would be interesting to add more implementation knowledge, with tests that, while conforming, will be more useful if the Lisp has been implemented in a particular way. The type propagation tester is an example of this kind of 'gray box' testing.

It would be interesting to determine the level of coverage achieved by the test suite in various implementations. The coverage is probably not very good, since the suite cannot contain tests of nonstandardized error situations, but this should be confirmed, and compared against the coverage obtained from running typical applications. Internal coverage could also provide feedback for nudging the random tester toward testing relatively untested parts of the compiler, say by using an evolutionary algorithm on the parameters governing the construction of random forms.

7 Acknowledgments

I would like to thank Camm Maguire, the head of the GCL development team, for allowing the GCL ANSI test suite to be a part of that project. I also would like to thank users of the test suite who have returned feedback, including Camm, Christophe Rhodes, Sam Steingold, Bruno Haible, Duane Rettig, Raymond Toy, Dan Barlow, Juan José García-Ripoll, Brian Mastenbrook and many others.

References

[1] James Bach and Patrick J. Schroeder. Pairwise testing: A best practice that isn't. In *Proc. 22nd Annual Pacific Northwest Software Quality Conference*, 2004. See http://www.pnsqc.org/proceedings/pnsqc2004.pdf.

[2] P. McGee C. Kaner, W. P. Bond. High volume test automation. At http://testingeducation.org/a/hvta.pdf, May 2004. Keynote address presented at the International Conference on Software Testing, Analysis, and Review (STAR East), Orlando, FL.

[3] Joe W. Duran and Simeon Ntafos. A report on random testing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[4] Ariel Faigon. Testing for zero bugs. At http://www.yendor.com/testing/, 2005.

[5] Christian Lindig. Random testing the translation of C function calls. At http://www.st.cs.uni-sb.de/ lindig/src/quest/quest.pdf, Feb. 2005.

[6] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[7] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[8] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[9] K. M. Pitman. Common Lisp HyperSpec. http://www.lispworks.com/reference/HyperSpec/Front/index.htm. A hyperlinked form of ANSI/INCITS document 226-1994. Translated in 1996 and updated in 2005.

[10] Don R. Slutz. Massive stochastic testing of SQL. In *Proc. 24th International Conference on Very Large Database Systems (VLDB'98)*, pages 618–622, Aug. 1998.

[11] Richard C. Waters. Supporting the regression testing of lisp programs. *SIGPLAN Lisp Pointers*, IV(2):47–53, 1991.

[12] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb 2002.