Systematic Unit Testing in a Read-eval-print Loop

Kurt Nørmark
(Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk)

Abstract: Lisp programmers constantly carry out experiments in a read-eval-print loop. The experimental activities convince the Lisp programmers that new or modified pieces of programs work as expected. But the experiments typically do not represent systematic and comprehensive unit testing efforts. Rather, the experiments are quick and dirty *one shot validations* which do not add lasting value to the software, which is being developed. In this paper we propose a tool that is able to collect, organize, and re-validate test cases, which are entered as expressions in a read-eval-print loop. The process of collecting the expressions and their results imposes only little extra work on the programmer. The use of the tool provides for creation of test repositories, and it is intended to catalyze a much more systematic approach to unit testing in a read-eval-print loop. In the paper we also discuss how to use a test repository for other purposes than testing. As a concrete contribution we show how to use test cases as examples in library interface documentation. It is hypothesized—but not yet validated—that the tool will motivate the Lisp programmer to take the transition from casual testing to systematic testing.
Key Words: Interactive unit testing, program examples, Scheme programming, Emacs.
Category: D.2.5, D.2.6, D.1.1.

1 Introduction

This paper is about systematic program testing done in a read-eval-print loop. A read-eval-print loop is also known as an interactive shell or a command interpreter. The paper deals with unit testing of Scheme functions [Kelsey 98], but the results of the paper are valid for any Lisp language, and beyond. Using a read-eval-print loop it is natural to *try out* a function immediately after it has been programmed or modified. It is easy to do so via a few interactions. This *trying out* process can be characterized as casual testing, and it stands as a contrast to systematic unit testing [Beck 94, Beck 98]. In this paper we describe how to manage and organize the testing activities which are done in a read-eval-print loop. It is hypothesized that better means for management and utilization of test cases will encourage programmers to shift from a casual testing mode to a more systematic testing mode.

In order to understand the context and the overall problem, which is independent of Lisp programming, we start with a brief discussion of different ways to execute programs. There are, in general and in essence, two different ways to execute a part of a program:

1. **Single entry point execution.**
   Program execution always starts at a designated place called the main program, typically a function or a method called `main`. The main program is the only entry point to the program. The only way to execute a given part P of a program is to arrange that P is called directly or indirectly from the main program.
2. **Multiple entry points execution.**
   Any top-level abstraction in the program serves as an entry point. Thus, if a program part is implemented as a top-level abstraction, the abstraction may be executed (called/instantiated) from a read-eval-print loop.

The work described in this paper relies on environments that support multiple entry points execution. Almost all Lisp environments belong to this category, and many environments for languages such as ML, Haskell, F#, Smalltalk, Ruby, and Python also support multiple entry points execution. The initial observation behind this work is the following:

Programmers who work in multiple entry points IDEs are privileged because once a top-level abstraction has been programmed it can be executed right away in a read-eval-print loop. It is hypothesized that programmers in such IDEs test non-trivial program parts incrementally and interactively in this manner. A problem is, however, that these test executions are sporadic and casual, not systematic. Another problem is that the test-executions are not collected and preserved. Hereby it becomes difficult to repeat the test executions if/when an underlying program part has been modified (regression testing). The latter problem probably amplifies the first problem: What should motivate a programmer to perform systematic and more comprehensive "one shot testing"—tests which cannot easily be repeated when a need arises in the future?

One way to approach these problems would be to collect and systematize the tests in the direction of single entry point execution. Thus, instead of doing interactive and incremental program testing, the programmer aggregates the individual tests in *test abstractions*. Such test abstractions are eventually initiated from a single entry point, possibly via a "smart test driver" in the realm of JUnit [JUnit 09] or NUnit [NUnit 09]. In our opinion this is the wrong way to go, because it will ultimately ruin the interactive, experimental, and exploratory programming process which is so valuable for creative program development in Lisp and similar languages.

As an alternative approach, we will outline support for systematic and interactive testing using a read-eval-print loop. The idea is that the programmer continues to test-execute—*try out* and *play with*— abstractions as soon as it makes sense. Following each such test execution the programmer should decide if the test case is worth keeping for future use. If a test case should be preserved, the tested fragment and its result(s) are enrolled in a test bookkeeping system—a *test case repository*. Test cases enrolled in the system can easily be re-executed, and they can be conveniently reused for other purposes.

In the rest of the paper we will describe a concrete tool, through which we have gained experience with systematic unit testing of Scheme programs in a read-eval-print loop. This will show how it is possible to manage the interactive test-execution in a systematic way.

The contributions of our work are twofold. The first contribution is the idea of systematic testing via a read-eval-print loop, the necessary support behind such a facility, and additional derived benefits from the approach. The second is the actual testing tools for Scheme, as implemented in Emacs Lisp and hosted in the Emacs text editor.

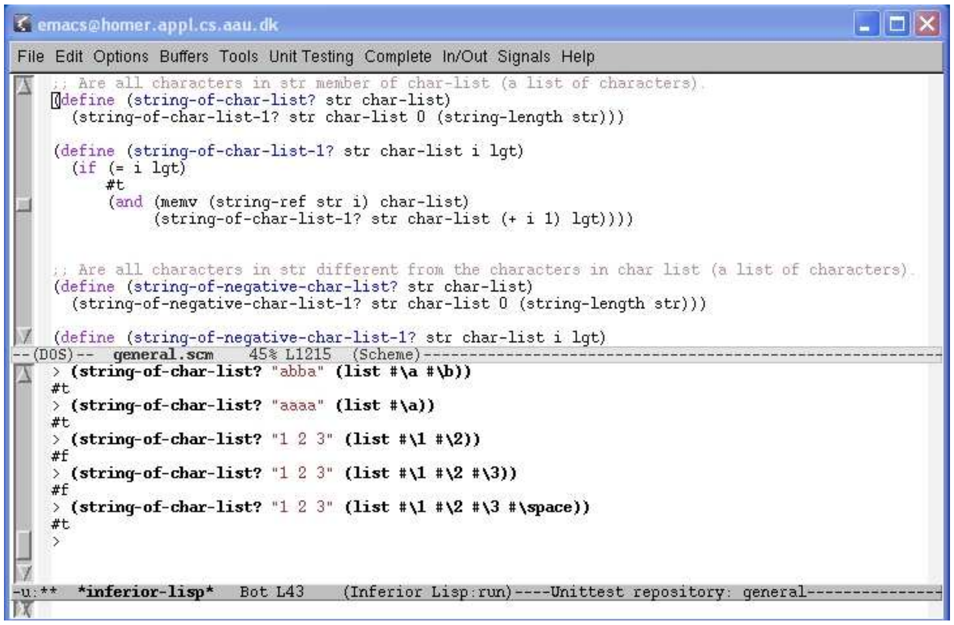
The rest of the paper is structured as follows. First, in Section 2, we discuss the idea of collecting test cases in a read-eval-print loop. Next, in Section 3 we introduce support for test case management. In Section 4 we describe how to use accumulated test cases as examples in library interface documentation. Section 5 describes the implemented tools behind our work. Related work is discussed in Section 6, and the conclusions are drawn in Section 7.

2 Collecting Tests in a Read-eval-print Loop

The slogan of test driven development [Beck 98], which is an important part of extreme programming [Beck 04], is "*test a little, code a little*". Following this approach the test cases are written before the actual programming takes place. In a less radical variant, "*code a little, test a little*", test cases are written after a piece of code has been completed.

For Lisp programmers, who work incrementally in a read-eval-print loop, the slogan has always been "*code a little, experiment a little*". We do not want to change that, because this way of working represents the spirit of dynamic, exploratory, interactive programming. It is also consistent with Paul Graham's notion of bottom-up programming in Lisp [Graham 93]. We will, however, propose that a subset of the experiments are turned into a more systematic and consolidated testing effort. In addition we will propose that the result of this effort is organized and preserved such that regression testing becomes possible. With appropriate support from the read-eval-print loop this only requires a minimum of extra work from the programmer.

As a typical and simple setup, a programmer who uses a read-eval-print loop works in a two-paned window, see Figure 1. In one pane a program source file is edited, and in the other pane the read-eval-print loop is running. (Variations with


emacs@homer.appl.cs.aau.dk
File Edit Options Buffers Tools Unit Testing Complete In/Out Signals Help

```
;; Are all characters in str member of char-list (a list of characters).
(define (string-of-char-list? str char-list)
  (string-of-char-list-1? str char-list 0 (string-length str)))

(define (string-of-char-list-1? str char-list i lgt)
  (if (= i lgt)
      #t
      (and (memv (string-ref str i) char-list)
           (string-of-char-list-1? str char-list (+ i 1) lgt))))


;; Are all characters in str different from the characters in char list (a list of characters).
(define (string-of-negative-char-list? str char-list)
  (string-of-negative-char-list-1? str char-list 0 (string-length str)))

(define (string-of-negative-char-list-1? str char-list i lgt)
--(DOS)--  general.scm    45% L1215  (Scheme)----------------------------------------
> (string-of-char-list? "abba" (list #\a #\b))
#t
> (string-of-char-list? "aaaa" (list #\a))
#t
> (string-of-char-list? "1 2 3" (list #\1 #\2))
#f
> (string-of-char-list? "1 2 3" (list #\1 #\2 #\3))
#f
> (string-of-char-list? "1 2 3" (list #\1 #\2 #\3 #\space))
#t
>

-u:**  *inferior-lisp*   Bot L43    (Inferior Lisp:run)----Unittest repository: general-------------
```

Figure 1: *A Scheme read-eval-print loop (bottom) and part of Scheme source program (top). The menu entry* `Unit Testing` *in the top-bar pertains to systematic unit testing.*

multiple editor panes, multiple read-eval-print loops, or combined editor pane and read-eval-print loop pane are supported in more advanced environments). Whenever an abstraction, typically a function, is completed in the upper pane of Figure 1, it is loaded and tried out in the lower pane.

In order to be concrete, we will assume that we have just programmed the Scheme function `string-of-char-list?`, as shown in the upper pane of Figure 1. The function `string-of-char-list?` examines if the string `str` consists exclusively of characters from the list `char-list`. We load this function into the read-eval-print loop, and we assure ourselves that it works correctly. This can, for instance, be done via the following interactions.

```
> (string-of-char-list? "abba" (list #\a #\b))
#t
> OK

> (string-of-char-list? "abbac" (list #\a #\b))
#f

> (string-of-char-list? "aaaa" (list #\a))
#t
> OK
> (string-of-char-list? "1 2 3" (list #\1 #\2))
#f
> OK

> (string-of-char-list? "1 2 3" (list #\1 #\2 #\3))
#f
> OK

> (string-of-char-list? "1 2 3" (list #\1 #\2 #\3 #\space))
#t
> OK

> (string-of-char-list? "cde" (list #\a #\b))
#f
> OK
```

The `OK` commands, issued after most of the interactions, are *test case acceptance commands* which signify that the latest evaluation is as expected, and that the expression and its value should be preserved. The command can be issued in a number of different ways depending on the style of the tool and the preferences of the programmer. In our concrete, Emacs-based tool we support test case acceptance commands both via a menu entry and via a terse control sequence (such as `C-t C-t`). Issuing the test acceptance command asserts that the returned value is equal to the value of the evaluated expression, using the default assertion (the `equal?` Scheme function in our setup).

Working in a testing context, we will turn experimentation into a more systematic testing effort. Continuing the example, it will make good sense to consolidate the test with test cases that care about extreme values of the text string and the character list:

```
> (string-of-char-list? "" '())
#t
> OK

> (string-of-char-list? "" (list #\a #\b))
#t
> OK

> (string-of-char-list? "ab" '())
#f
> OK
```

As it can be seen, the programmer accepts all these interactions as test cases.

It is also possible to accept an *error test case*. As an example, the read-eval-print loop interaction

```
> (string-of-char-list? #\a (list #\a))
string-length: expects argument of type <string>; given #\a
> ERROR
```

is categorized as an error, because `string-of-char-list?` only accepts a string as its first parameter. The `ERROR` command signals that an error must occur when the expression from above is evaluated.

In the spirit of test-driven development it is also possible—in a read-eval-print loop—to write test cases before the function under test is implemented. Using this variant, it is necessary to provide the expected value of an expression, which cannot yet be successfully evaluated. Here follows a possible interaction in case the function `string-of-char-list?` has not yet been implemented:

```
> (string-of-char-list? "abba" (list #\a #\b))
reference to undefined identifier: string-of-char-list?
> VALUE: #t

> (string-of-char-list? "abbac" (list #\a #\b))
reference to undefined identifier: string-of-char-list?
> VALUE: #f
```

The `VALUE:` commands are issued with the purpose of providing the intended value of the expressions which fail.

The approach described above will be called *interactive unit testing*. Interactive unit testing works well for testing of pure functions. The reason is that a test of a function f can be fully described by (1) an expression e that invokes f and (2) the value of the e.

The interactive unit testing approach is less useful for testing of procedures or functions with side effects. A test of a procedure p involves, besides step 1 from above, (0) establishment of the initial state, and (3) possible cleaning up after p has been called. Non-interactive unit testing frameworks typically prepare for step 0 and 3 via *setup* and *teardown* actions for each individual test case. It is not easy, nor natural, to collect these four pieces in a read-eval-print loop. Therefore we recommend that test cases for imperative abstractions are captured at the level of more conventional, non-interactive unit testing. This may harmonize better than expected with interactive unit testing, because a traditional unit testing tool is in fact part of the tool-chain of the proposed implementation, see Section 5.

3 Test Case Management

In the previous section we have described the basic idea of turning selected expressions and values, as they appear in a read-eval-print loop, into test cases. We will now describe how our tool keeps track of these test cases, and how the program developer can deal with the accumulated test cases. We start with an explanation of test case management that involves pure functions. At the end of this section we will touch on management of test cases in imperative programming.

3.1 Basic test case management

A test case of a pure function is composed of the following information:

1. The expression which serves as the calling form.
2. The expected value of the expression.
3. The predicate that is used to verify that the expected value is equal to the outcome of executing the expression.
4. A unique id, which contains a time stamp.
5. Additional information which, for instance, pertains to the use of the test case as an example in interface documentation (see Section 4).

In case the expression is supposed to give rise to an error, the test case reflects this as a special case.

The existence—and acceptance—of a test case reflects the fact that the test case represents correct behavior of the involved abstractions. The acceptance of the test case is made by the programmer by issuing a test case acceptance command in the read-eval-print loop, just after the evaluation of the expression (see Section 2).

The basic test management idea is simple. A given session with a read-eval-print loop can be carried out relative to a designated *test case repository*. At an arbitrary point in time the programmer can connect to an existing test case repository, or the programmer can ask for a new one. Subsequent test case acceptance commands add test cases to the designated repository. If a programmer accepts a test case without being connected to a repository, the programmer is prompted for a repository for this particular test case. It is possible, at any point in time, to switch from one test case repository to another.

Each test case repository—corresponding to a test-suite—can associate a setup file and a tear down file. The setup file may contain a common context of all test cases, and as such the context is shared between all the test cases in the repository. The setup file typically also contains some instructions for loading of appropriate source files. The tear down file is typically empty.

It is up to the programmer to decide the relationship between the program source files and test case repositories. It is our experience that it is natural to have one test case repository per library source file, and one per application source file. It is possible to register one or more source files in association with a test case repository. This registration makes it possible to locate a given function in a source file, e.g. in case it is necessary to correct an error.

When the programmer asks for a new test case repository, he or she is prompted for a hosting directory *hd* and a name *n* of the repository. This will, behind the scene, create a directory called *n*`-test` in the directory *hd*. In the vicinity of the interactive unit testing software we keep a list (in a file) that holds information about all registered test case repositories. A new test case repository directory contains a file with all the test cases together with the setup and teardown files. In addition a number of temporary files for various purposes are kept in the directory. The programmer does not need to be aware of the test-related directories.

3.2 Unit testing functionality

The test enabled read-eval-print loop supports a rich collection of functionality. The functionality is organized in the 'Unit Testing' menu of the read-eval-print pane, see Figure 1. The menu appears automatically when the supported read-eval-print loop runs together with our interactive unit testing tool. The 'Unit Testing' menu itself is shown in Figure 2. We will now describe the most important unit testing functionality of the read-eval-print loop. The description will be structured relative to the grouping of the menu entries in Figure 2.

The first group of functionality pertains to 'connecting to' and 'disconnecting from' a test suite, as described in Section 3.1. A test suite is contained in a test case repository. In the next group there are commands for registration and deregistration of test suites. In addition to registering and deregistering of a single test suite, it is possible to deregister all test suites. The tool is also able to search through a directory tree for all test case repository directories, and to register these. This is useful if test cases are brought in from external sources, such as another development machine.

As another category of functionality, it is possible to execute all test cases in the current test case repository from the read-eval-print loop. It is also possible to execute all test cases in all registered repositories. In these ways, execution of all test cases can easily take place on a regular basis.

The current test case repository can be opened for examination and editing. It is, in addition, possible to inspect and edit the setup file, the teardown file, and a selected source file under test. In also possible to inspect the Scheme file that holds all the low-level unit tests (aggregated automatically by the tool), as well as a test report in which test errors and test failures are reported.

The next two groups of commands add test acceptance commands. These commands correspond to the `OK`, `ERROR`, and `VALUE:` commands described in Section 2. The two source file info commands in the following group make it possible connect a test suite to a set of Scheme source files. With this connection it is possible to provide for smooth navigation from a test case to the function under test.

Our tool preserves all entered input to the read-eval-print loop, independent of the built-in history list, and independent of the test cases in the repositories.

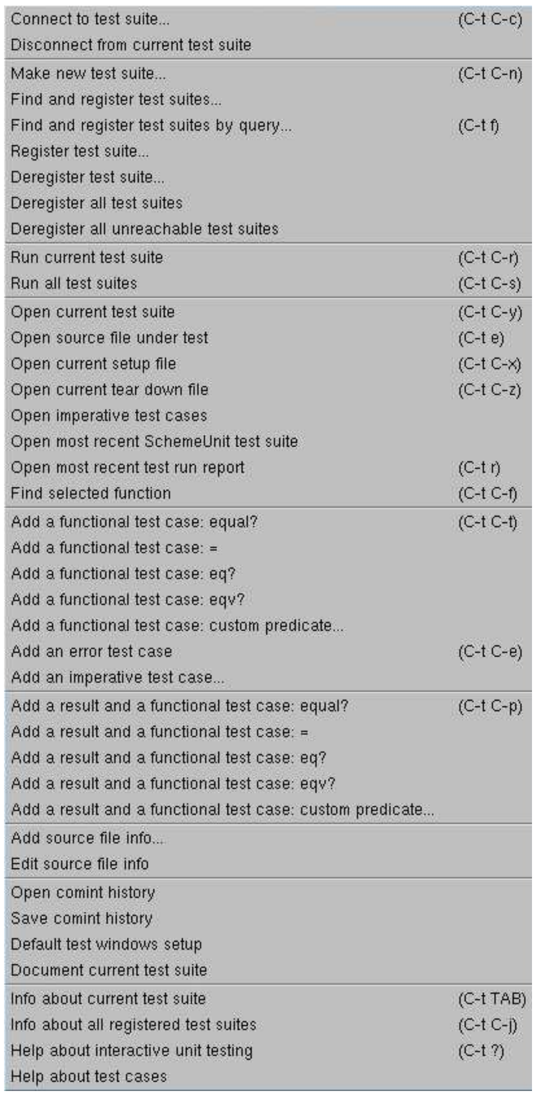

Connect to test suite... (C-t C-c)
Disconnect from current test suite
Make new test suite... (C-t C-n)
Find and register test suites...
Find and register test suites by query... (C-t f)
Register test suite...
Deregister test suite...
Deregister all test suites
Deregister all unreachable test suites
Run current test suite (C-t C-r)
Run all test suites (C-t C-s)
Open current test suite (C-t C-y)
Open source file under test (C-t e)
Open current setup file (C-t C-x)
Open current tear down file (C-t C-z)
Open imperative test cases
Open most recent SchemeUnit test suite
Open most recent test run report (C-t r)
Find selected function (C-t C-f)
Add a functional test case: equal? (C-t C-t)
Add a functional test case: =
Add a functional test case: eq?
Add a functional test case: eqv?
Add a functional test case: custom predicate...
Add an error test case (C-t C-e)
Add an imperative test case...
Add a result and a functional test case: equal? (C-t C-p)
Add a result and a functional test case: =
Add a result and a functional test case: eq?
Add a result and a functional test case: eqv?
Add a result and a functional test case: custom predicate...
Add source file info...
Edit source file info
Open comint history
Save comint history
Default test windows setup
Document current test suite
Info about current test suite (C-t TAB)
Info about all registered test suites (C-t C-j)
Help about interactive unit testing (C-t ?)
Help about test cases

Figure 2: *The unit testing menu entries of the read-eval-print loop illustrated in Figure 1.*

The two comint history command are related to this functionality. With this facility, it is relatively easy to reproduce the interactions "from yesterday" or from "last week", in case these interactions contain contributions that we "today" want to organize in test case repositories. The command 'Document current test suite' pretty prints an aggregate consisting of the setup file, all test cases, and the teardown file to a web page. The command ‘Default windows setup’ normalizes the panes to show the read-eval-print loop and the Scheme source file that contains the functions under test.

The commands in the last group are related to information and help.

3.3 Maintenance of unit tests

During maintenance of the software under test, regression testing may sooner or later reveal failures or errors. A failure occurs if a function under test becomes inconsistent with a test case. An error occurs if, for instance, a tested function is deleted or renamed. Test case failures and errors are revealed in reports of the following form:

```
...
Error:
------
17395-10319
 an error of type exn:variable occurred with message:
 "reference to undefined identifier: blank-string?"

Error:
------
17395-10394
 an error of type exn:variable occurred with message:
 "reference to undefined identifier: blank-string?"

Failure:
--------
19240-61944
/lib/general-test/testsuite.scm:1140:4
  name: "assert"
  location: ("/lib/general-test/testsuite.scm" 1140 4 59367 107 #f)
  expression: (assert equal?
                 (string-of-char-list? "1 2 3" (list #\1 #\2 #\3))
                 (quote #t))
  params: (#<primitive:equal?> #f #t)
...
```

The reported problems may be caused by a renaming of the function `blank-string?` and by a change of `string-of-char-list?`. The numerical tokens following the six dashes are the unique test case ids. Forward and backward navigation between test case ids are provided. It is also possible to access the test case from this id, including the tested expression and the expected value. In addition, it is possible to delete the test case via a test case id. As a special twist, a test case can be automatically transferred to the read-eval-print loop just before deleting it. In that way erroneous test cases can be corrected and re-accepted.

In Figure 3 we show a diagram that illustrates the read-eval-print loop together with test report and other important test-related editor panes. The connections between boxes in the figure illustrate test-related interactions.

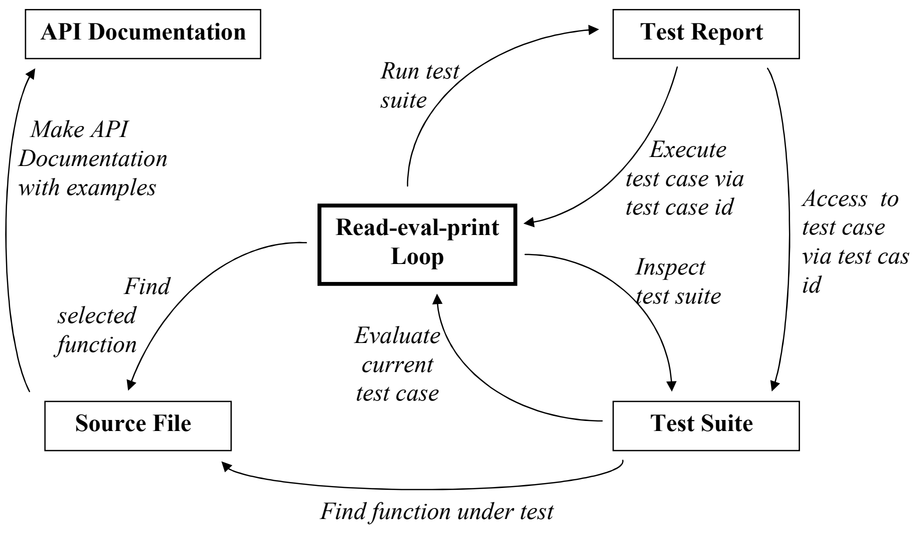


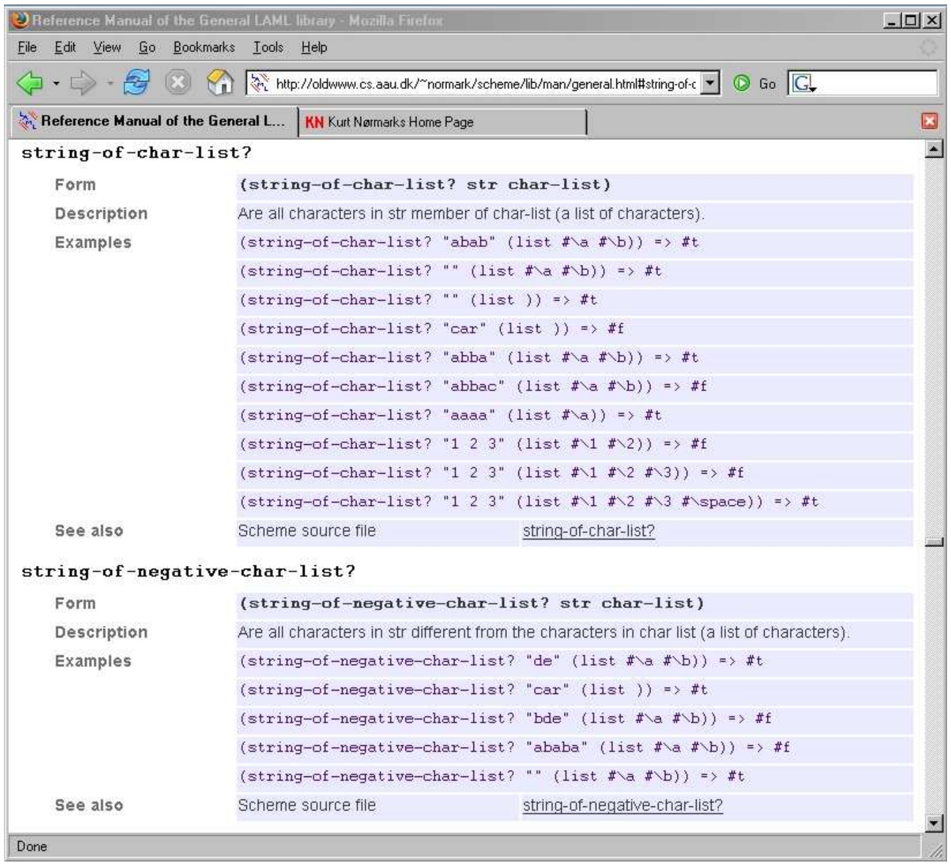
Figure 3: *The read-eval-print loop and other test-related panes, together with the interactive relations (commands) that tie them together.*

3.4 Imperative test cases

Each test case repository also contains test cases for procedures (or functions with side-effects). Test cases that involve imperative programming are handled at the level of the underlying unit-testing tool. It is possible to add a low-level test case to the test case repository, and it is possible to edit the collection of such low-level test case. In addition, it is possible—in a flexible way—to execute a low-level test case in order to eliminate potential errors in the test case at an early point in time. Due to the simple nature of test cases that involve pure functions it is unlikely that there will be errors in such test cases. Test cases that involve procedures and imperative programming are more complex, and therefore it is valuable to try them out in isolation, as early as possible.

4 Test and Documentation

The primary benefit of an interactive unit testing tool in a read-eval-print-loop, as described in Section 2 and 3, is easy and natural collection and preservation of test cases. In addition, an interactive unit testing tool allows for convenient management and execution of the collected test cases.


Reference Manual of the General LAML library - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://oldwww.cs.aau.dk/~normark/scheme/lib/man/general.html#string-of-c
Go
Reference Manual of the General L...
KN Kurt Nørmarks Home Page
Done

string-of-char-list?

Form	`(string-of-char-list? str char-list)`
Description	Are all characters in str member of char-list (a list of characters).
Examples	`(string-of-char-list? "abab" (list #\a #\b)) => #t`
	`(string-of-char-list? "" (list #\a #\b)) => #t`
	`(string-of-char-list? "" (list )) => #t`
	`(string-of-char-list? "car" (list )) => #f`
	`(string-of-char-list? "abba" (list #\a #\b)) => #t`
	`(string-of-char-list? "abbac" (list #\a #\b)) => #f`
	`(string-of-char-list? "aaaa" (list #\a)) => #t`
	`(string-of-char-list? "1 2 3" (list #\1 #\2)) => #f`
	`(string-of-char-list? "1 2 3" (list #\1 #\2 #\3)) => #f`
	`(string-of-char-list? "1 2 3" (list #\1 #\2 #\3 #\space)) => #t`
See also	Scheme source file string-of-char-list?

string-of-negative-char-list?

Form	`(string-of-negative-char-list? str char-list)`
Description	Are all characters in str different from the characters in char list (a list of characters).
Examples	`(string-of-negative-char-list? "de" (list #\a #\b)) => #t`
	`(string-of-negative-char-list? "car" (list )) => #t`
	`(string-of-negative-char-list? "bde" (list #\a #\b)) => #f`
	`(string-of-negative-char-list? "ababa" (list #\a #\b)) => #f`
	`(string-of-negative-char-list? "" (list #\a #\b)) => #t`
See also	Scheme source file string-of-negative-char-list?

Figure 4: *Use of the* `string-of-char-list?` *test cases as examples in the interface documentation of the function.*

A clean representation of test cases in repositories allows for additional applications of the test cases. One such application is use of selected test cases as *examples* in documentation of libraries. The documentation we have in mind is *interface documentation*, as produced by tools such as Javadoc [Friendly 95], Doxygen [van Heesch 04] and SchemeDoc [Nørmark 04].

As the name suggests, interface documentation describes the application programmer's interface (API) of program libraries. Interface documentation is typically extracted from function headers/signatures and from the text in designated *documentation comments*. Examples of usages of the documented abstractions (functions, for instance) are very useful for library client programmers, because such examples are close to the actual needs of the programmer. Unfortunately, it is time consuming and error prone to write examples that cover the use of a large collection of functions in a program library.

Test cases in test case repositories can be used directly as examples within interface documentation. Given some function `f` in the library, which we are about to document, there should be means to select a number of test cases which can serve as examples of using `f`. It is a natural and necessary requirement that `f` should be represented in the expression of such test cases. The test case with an expression such as

```
(string-of-char-list? "123" '(#\1 #\2 #\3))
```

is probably a worthwhile example of the function `string-of-char-list?` in its documentation, whereas the test case with the expression

```
(string-of-char-list? "123" (map as-char '("1" "2" "3")))
```

is less likely to add to the documentation of the functions `map` and `as-char`. For each test case *tc* it is possible (but not necessary) to enumerate the functions for which *tc* should act as an example. Without the support of such a selective mechanism it is our experience that some test cases show up at unexpected places in the interface documentation.

The quality of examples derived from test case repositories can be expected to be higher than the quality of examples produced by most other means. The examples extracted from test case repositories are executed on a regular basis, as part of regression testing. Therefore we can be confident that the expressions are lexically and syntactically correct, and that the results displayed are correct relative to the current implementation of the underlying software.

In order to make use of test cases as examples in interface documentation, it is necessary to provide an interface between the test case repository and the interface documentation tool. The interface documentation tool must be able to read selected test case repositories, it must understand the actual representation of test cases, and it must be able to select and extract test cases which are relevant for a given entry in the documentation.

5 The Implemented Tool

The implemented unit testing tool supports interactive unit testing of Scheme libraries and Scheme programs. More specifically, the tool works with MzScheme [Flatt 00] (R5RS) which is part of PLT Scheme and DrScheme [Findler 02]. The interactive unit testing tool interfaces to `SchemeUnit` [Welsh 02], which is a conventional unit testing framework for PLT Scheme. It means that the tool generates test expressions in the format prescribed by `SchemeUnit`, such that the actual test executions can be done by `SchemeUnit`.

The read-eval-print loop of our tool is Olin Shivers' and Simon Marshall's command interpreter—`comint`—which is part of GNU Emacs. Our interactive unit testing tool is implemented in Emacs Lisp, as an extension to `comint`. As shown in Figure 1 and Figure 2 the tool manifests itself via an extra 'Unit Testing' menu of the Emacs command interpreter, and via a number of interactive Emacs commands and associated key bindings.

The `comint` command interpreter has been augmented with a number of command, as discussed in Section 3.2. The test case reports, as generated by `SchemeUnit`, have been associated with an Emacs mode. Via this mode, a number of commands have been implemented on test case ids, see Figure 3. Similarly, the presentations of test suites have been associated with another Emacs mode that defines a number of useful commands on individual test cases. These commands, taken together, bind the facilities of the interactive testing tool together, and they catalyze a smooth test-related workflow from within Emacs.

The test cases produced by our tool can be referred and applied by LAML SchemeDoc [Nørmark 04], which is an interface documentation tool we have created for the Scheme programming language. The screen shot in Figure 4 shows an excerpt of some SchemeDoc interface documentation produced on the basis of test cases similar to those discussed in Section 2. In the LAML software package [Nørmark 09] it is possible to consult examples of real-life API documentation, such as the "General LAML library", that contain many examples. Far the majority of these examples stem from interactive unit testing.

6 Related Work

An early paper about unit testing [Beck 94] is written in the context of Smalltalk. In the introduction to this paper Kent Beck states that he dislikes user interface-based testing. (In this context, user interface-based testing is understood as testing through the textual or graphical user interface of a program). The observation of user interface-based testing motivated a testing approach based on a library in the programming language. This is the testing approach now known as *unit testing*, and supported by *x*Unit tools for various languages *x*. The work described in this paper makes use of one such unit testing tool, SchemeUnit [Welsh 02], for the programming language Scheme.

JUnit [JUnit 09] and NUnit [NUnit 09] represent well-known and widely used unit testing tools for the Java and .Net platforms respectively. In the context of this paper, JUnit and NUnit are classified as non-interactive unit testing tools. The individual test cases supported by JUnit and NUnit are written as parameterless test methods in separate classes. The primary contribution of JUnit and NUnit is to activate these test methods. This is done by *single entry point execution*, as described in Section 1, via an aggregated `Main` method generated implicitly by JUnit and NUnit.

RT is an early regression testing tool for Common Lisp, programmed by Richard C. Waters, and described in a Lisp Pointers paper [Waters 91]. Along the same lines as RT, Gary King describes a Common Lisp framework for unit testing called LIFT [King 01]. LIFT is based on a CLOS representation of test cases and test suites. A number of other unit testing tools exists for Common Lisp. Among these, CLUnit [Adrian 01] was designed to remedy some identified weaknesses in RT. Stefil [Stefil 09] is a Common Lisp testing framework, which can be used interactively from a read-eval-print loop. Stefil provides a number of macros, such as `deftest` and `defsuite`, which act as instrumented function definitions. Test cases defined in Stefil must to embedded in the test abstractions defined by these macros. In our work no such test abstractions are necessary. Instead, special commands supported by the read-eval-print loop are responsible for the management of the test cases.

The Python library called `doctest` [Doctest 09] makes it possible to represent test cases in the documentation string of definitions. The test cases can be copied and pasted directly from an interactive console session, corresponding to a read-eval-print loop. The Python doctest facility is able to identify the test cases within the documentation string, to execute the test cases, and to compare the actual result with the original result. In the documentation of the Python doctest facility, the approach is incisively described as "literate testing" or "executable documentation". The Python `doctest` facility stores the test cases inside the documentation strings, and therefore the documentation-benefit of the approach is trivially obtained. Following the Python approach, pieces of interactions must be copied from the interactive console to the documentation string. No such copying is necessary in our interactive unit testing tool. Our representation of the test cases is more versatile than the proposed Python approach. In our approach it is straightforward to use a single test case as an example in the documentation of several function definitions.

Some work similar to the Python `doctest` library has also been brought up in the context of Common Lisp. Juan M. Bello Rivas [Rivas 09] describes a doctest facility in which test cases, harvested from a read-evel-print loop, is represented in the documentation string of a function. A function called `doctest` is able to carry out regression testing based on this information.

An important theme in the current paper is convenient capturing of test cases. We have proposed that test cases are captured when new or modified functions are *tried out* in a read-eval-print loop. In the paper *Contract Driven Development = Test Driven Development - Writing Test-Cases* [Leitner 07] it is proposed to capture test cases when a contract is violated. The work in the paper is carried out in the context of Eiffel and Design by Contract. In the same way as in our work, it is assumed that a given abstraction (a method) is explicitly tried out. In a single entry point program this may require an explicitly programmed test driver. The contribution of the paper—and the tool behind it—is to collect test cases (including the necessary context) which leads to violation of the contracts in programs. The collected test cases can be executed again, even without the programmed test driver. It is hypothesized that the semi-automatically collected test cases are useful for future regression testing.

In a couple of papers Hoffman and Strooper discuss the use of test cases for documentation and specification purposes, [Hoffman 00, Hoffman 03]. Test cases for Java are written in a particular language, similar to (but different from) JUnit. The approach of the papers is to use such test cases (examples) together with informal "prose" as API documentation. In the papers the combination of informal prose descriptions and concisely formulated test cases is seen as an alternative to formal specifications (such as contracts in terms of preconditions and postconditions). In contrast to the current paper, the papers by Hoffman and Strooper do not propose how to integrate the informal prose descriptions with presentations of the test cases.

In a paper about *Example Centric Programming* Edwards discusses some radical ideas that unify programming editing, debugging, exploring, and testing [Edwards 04]. The ideas are accompanied by a prototype tool which can be seen as a forerunner of the more recent Subtext system [Edwards 05]. In the paper it is discussed how to capture unit tests more easily than in traditional unit testing tools, such as NUnit and JUnit. The paper states directly that "Unit test can serve as examples". In the current paper this is taken to mean examples in API documentation, whereas in Edwards' work, examples are more akin to stack traces known from conventional debuggers. The idea in Edwards' work is to freeze a given method activation and its result—as they occur in an example pane—as an assertion. This is similar to our idea of using and registering an expression and its result—captured during experiments in a read-eval-print loop—as a test case.

7 Summary and Conclusions

The observation and starting point of this paper is that most Lisp programmers—and programmers who use similar languages—experiment with new or modified pieces of programs. The languages in the Lisp family allow multiple entry point execution, and they make use of read-eval-print loops for evaluation of expressions and commands. The idea, which we have elaborated in the paper, is to consolidate these experiments to systematic test cases, and to allow repeated execution of the tests when needed. We have described a facility that organizes test cases in a repository from which conventional unit tests can be derived automatically. The feeding of test cases into the repository—in terms of experimental evaluation of expressions—is done in the read-eval-print loop. The process of collecting test cases does not impose much extra work on the programmer. A little extra work is, however, needed to consolidate the test cases in the direction of systematic testing, to organize the test cases, and to prepare for execution of the test suites (the common setup and the teardown files). We hypothesize that programmers are willing to do this amount of extra work in return for the described benefits.

Conventional test suites, as prepared for unit testing tools such as JUnit, NUnit, CLUnit, and SchemeUnit, do not allow for easy extraction of examples that can be used by interface documentation tools. The reason is that test cases are intermingled with the unit testing vocabulary, such as test abstractions and various assertions. Following the approach described in the current paper, the expression and the expected result of a functional test case are represented in a clean manner, which makes it straightforward to aggregate examples.

The interactive unit testing tool has been used for collecting and organizing test cases during the development of the LAML software package [Nørmark 05]. The use of test cases as examples in the documentation of the LAML libraries turns out to be a valuable contribution to the everyday comprehension of the functions in the library.

As can be seen, the idea and the solution described in this paper are fairly simple. Nevertheless, the implications and benefits from exploiting the ideas may be substantial. We conclude that

1. Lisp programmers are already doing substantial work in the read-eval-print loop, which almost automatically can be turned into test cases.

2. With a little extra organization and support, it is possible to preserve and repeat the execution of the test cases.

3. Providers of interactive development environments (IDEs) that support read-eval-print loops should consider an implementation of the solution, which is similar to the approach described in this paper.

The Emacs Lisp program that implements the interactive unit testing tool is bundled with LAML, which is available as free software from the LAML home page [Nørmark 09].

Acknowledgements

I want to thank the anonymous reviewers for valuable feedback to an earlier version of this paper.

References

[Adrian 01] Frank A. Adrian. *CLUnit - A Unit Test Tool for Common Lisp*, 2001. `http://www.ancar.org/CLUnit/docs/CLUnit.html`.

[Beck 94] Kent Beck. *Simple Smalltalk Testing: With Patterns*. The Smalltalk Report, vol. 4, no. 2, 1994.

[Beck 98] Kent Beck & Erich Gamma. *JUnit Test Infected: Programmers Love Writing Tests*. Java Report, vol. 3, no. 7, pages 51–56, 1998.

[Beck 04] Kent Beck. Extreme programming explained: Embrace change (2nd edition). Addison-Wesley Professional, 2004.

[Doctest 09] Phyton Doctest. *doctest. Test interactive Python examples*, 2009. `http://docs.python.org/library/doctest.html`.

[Edwards 04] Jonathan Edwards. *Example centric programming*. In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 124–124, New York, NY, USA, 2004. ACM.

[Edwards 05] Jonathan Edwards. *Subtext: uncovering the simplicity of programming*. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pages 505–518, New York, NY, USA, 2005. ACM.

[Findler 02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler & Matthias Felleisen. *DrScheme: A Progamming Environment for Scheme*. Journal of Functional Programming, vol. 12, no. 2, pages 159–182, March 2002.

[Flatt 00] Matthew Flatt. *PLT MzScheme: Language Manual*. `http://www.cs.-rice.edu/CS/PLT/packages/pdf/mzscheme.pdf`, August 2000.

[Friendly 95] Lisa Friendly. *The Design of Distributed Hyperlinked Programming Documentation*. In Sylvain Frass, Franca Garzotto, Toms Isakowitz, Jocelyne Nanard & Marc Nanard, editors, Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1995.

[Graham 93] Paul Graham. On lisp. Prentice Hall, 1993.

[Hoffman 00] Daniel Hoffman & Paul Strooper. *Prose + Test Cases = Specifications*. In TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), page 239, Washington, DC, USA, 2000. IEEE Computer Society.

[Hoffman 03] Daniel Hoffman & Paul Strooper. *API documentation with executable examples*. The Journal of Systems and Software, vol. 66, no. 2, pages 143–156, 2003.

[JUnit 09] JUnit. *http://junit.org/*, 2009.

[Kelsey 98] Richard Kelsey, William Clinger & Jonathan Rees. *Revised*^{5} *Report on the Algorithmic Language Scheme*. Higher-Order and Symbolic Computation, vol. 11, no. 1, pages 7–105, August 1998.

[King 01] Gary King. *LIFT - the Lisp framework for testing*. Technical report, University of Massachusetts, 2001.

[Leitner 07] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer & Arnaud Fiva. *Contract Driven Development = Test Driven Development - Writing Test-Cases*. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), September 2007.

[Nørmark 04] Kurt Nørmark. *Scheme Program Documentation Tools*. In Olin Shivers & Oscar Waddell, editors, Proceedings of the Fifth Workshop on Scheme and Functional Programming, pages 1–11. Department of Computer Science, Indiana University, September 2004. Technical Report 600.

[Nørmark 05] Kurt Nørmark. *Web Programming in Scheme with LAML*. Journal of Functional Programming, vol. 15, no. 1, pages 53–65, January 2005.

[Nørmark 09] Kurt Nørmark. *The LAML home page*, 2009. `http://www.cs.aau.-dk/~normark/laml/`.

[NUnit 09] NUnit. *http://www.nunit.org*, 2009.

[Rivas 09] Juan M. Bello Rivas. *(incf cl) utilities*, November 2009. `http://-superadditive.com/projects/incf-cl/`.

[Stefil 09] Stefil. *Stefil - Simple test framework in Lisp*, 2009. `http://-common-lisp.net/project/stefil/`.

[van Heesch 04] Dimitri van Heesch. *Doxygen*, 2004. `http://www.doxygen.org`.

[Waters 91] Richard C. Waters. *Supporting the regression testing in Lisp programs.* SIGPLAN Lisp Pointers, vol. IV, no. 2, pages 47–53, 1991.

[Welsh 02] Noel Welsh, Francisco Solsona & Ian Glover. *SchemeUnit and SchemeQL: Two Little Languages.* In Third Workshop and Scheme and Functional Programming, October 2002.