Grading Scheme Submissions

Karissa R. McKelvey <krmckelv@indiana.edu>
Aaron W. Hsu <arcfide@sacrideo.us>

(mags grade) The —(mags grade)— library implements the top-level functionality for grading assignment submissions based on SRFI 64 style test suites. The library serves two purposes. Firstly, it provides a set of procedures to use in test suites and auto-grader files. Secondly, it xprovides fairly flexible means of running those tests on code files.

26/3: grade test-suite test-group test-assert test-equal test-set test-eqv test-eq test-approx test-begin test-end current-test-runner test-entry define-equality-test current-test-file current-sandbox-name current-time-limit define-from-submission run-grade-tests numeric numeric-¿roman-lower-case numeric-¿roman-upper-case alphabetic-lower-case alphabetic-upper-case separator formatters

(chezscheme) (except (rename (srfi :64) (test-group srfi:test-group) (test-runner-current current-test-runner)) test-assert test-equal test-eqv test-eq) (mags sandbox) (mags runners)

0Overview This section provides a brief overview of the architecture of the system and the way that it is conceptually formed. First, let's look at the various procedures and discuss each one.

`grade`
This is the main syntax that calls the grading system on a specific submission.

`test-{suite,group,assert,equal,eqv,eq,begin,end,entry}`
These procedures are an extended vocabulary for test suites. It is based directly on the SRFI 64 model and uses SRFI 64 underneath the hood.

`test-runner-*`
These are specific parameters and constants for configuring how the tests are displayed as well as allowing detailed control over exactly how the testing system handles each test result.

`define-equality-test`
This allows user defined equality metrics to be created, such as equivalent or custom versions of —test-equal— and its ilk.

`current-*`
These are procedures used to configure the default behavior of —grade—.

`define-from-submission`
This is a special syntax used to grab bindings out from the student's code environment.

`formatters`
This is a parameter that enables custom control over how the auto-numbering of assignments is formatted and displayed.

The exact documentation for each specific export is documented where that procedure or macro is defined. However, the first few sections do not contain any code and exist solely for exposition on the basic workings of the program. They focus on examples and higher-level overview material rather than the exact features and behavior for each procedure. Some users may be able to get by entirely on the content of the first few sections, but it is expected that users will read the documentation for each individual procedure or macro that they intend to use, and not just the examples and lighter descriptions in the front of this document.0Auto-numbering assignment problems The macro —test-suite— creates assignments from files that contain (srfi :64) test cases. Each string represents a file in the —(cd)—. Here is a quick example of how —test-suite— is used:

(test-group "Assignment 10" (test-suite "insert.ss" "switcheroo.ss" ("merge.ss" "mergesort.ss") "bst-insert.ss")) —endverbatim

This creates a problem set that would be recognized by a test runner like so:

Assignment 10: 1: insert 2: switcheroo 3: 3.a: merge 3.b: mergesort 4: bst-insert —endverbatim

Merge and mergesort become grouped together as subproblems of one problem — —test-suite— recognizes this because they are included within a nested list. Order matters.

After your test file has been created, you can run the test cases contained within your problem sets against a given submission by using the following procedure:

(grade "submission.ss" "a10.ss" '(chezscheme) 1000000) —endverbatim

—grade— uses, by default, the '(chezscheme) library as the sandbox that the submission will be graded in. You most definitely don't want to be using this as the default library, because then the submission will have complete access to all (chezscheme) procedures, which could cause a whole host of unpredicatbles. 0An Example Assignment For this example assignment, we are going to create seperate files for each problem set. We then create a file for the assignment which will simply include these files in a either a —test-suite— or with nested —test-groups—. An example of this can be found later in this section. Having each problem set separated into its own file allows you to store them and use them in different assignments or semesters, in order to keep a library of problem sets. You can then easily create and modify assignments by modifying just one file, never having to look at the test cases once you have created them! This is also an optional way to do this; you can also put the test-groups in order directly into the top-level test-group.

The files containing the test cases, in this example, contain a —test-group— which has many —test-equal— clauses. The problems we are grading will be insert, switcheroo, merge, mergesort, and bst-insert. 0Problem sets Let's begin by explaining how to create a 'problem set', which could be described as a set of test cases contained within a —test-group—. We will be testing —insert— in this example.

Insert takes a relation, the item to insert, and a list and inserts that item according to the given relation. Here is an example of what our problem-set, insert.ss could contain:

(test-group "insert" (test-equal "Insert into middle, increasing" '(1 3 5 6 7 9 11) (insert ¡ 6 '(1 3 5 7 9 11))) (test-equal '(0 -1 -2 -5) (insert ¿ 0 '(0 -1 -2 -5)))) —endverbatim

In the second —test-equal— test case, the name would be auto-generated. When no name is provided to a test-¡equality-pred¿ test, it takes the name as a —pretty-print— of the full tested expression.

(test-equal '(1 3 5 6) (insert ¡ 6 '(1 3 5))) =¿ (test-equal "(insert ¡ 6 '(1 3 5)) '(1 3 5 6) (insert ¡ 6 '(1 3 5)))" ... —endverbatim

0What about my special ADT? Let's say our Abstract Data Type, trees, is defined in a library —tree.ss—. We can create our own test to use the ADT's pre-defined predicates using —define-equality-test—.

—define-equality-test— takes a name and a procedure, and uses that procedure to create a —(srfi :64)— test case. More on using these test-cases and other types of tests can be found in Section 4, titled "Using the Testing Framework". These new tests you define will work exactly the same way as —test-equal—. Here are some examples:

¿ (define-equality-test test-name (lambda (x y) (equal? x y))) ¿ (test-name "these are equal" 3 3) PASS ¿ (define-equality-test test-equal equal?) ¿ (test-equal "this should fail" 3 4) FAIL ¿ (define-equality-test test-same-shape same-shape?) ¿ (test-same-shape "these are trees of the same shape" (tree 1 (leaf 1) (leaf 0)) (tree 3 (leaf 0) (leaf 1))) PASS —endverbatim

You can also use —define-equality-test— to handle programs that have special behavior. You can simply pass a function that takes two arguments, with the second argument being the —actual value—. 0Building a test-file Now that we understand the basics of creating test cases and problem sets, lets look at building the individual parts of an assignment's —test-file—. In this example, we will build the file in parts in order to understand the entire file, and then display the finished file at the end of this section.

The —test-file—, named —a10.ss— in this example, holds all of these test cases. This file must call —(define-from-submission procedure-name ...)—, which declares which names of the procedures that will be graded from the submission. This might be easily extended to instead automatically pull these names from the —user-sandbox—, however that is not currently the case. They must be explicitly stated in the assignment file. *Order doesn't matter.*

(define-from-submission merge mergesort insert bst-insert switcheroo) —endverbatim

When you call a —test-group— without giving it a name, the software will automatically number the internal —test-groups— in the order they appear, using —(separator)— between the problem numbers, formatted according to —(formatters)—. This library provides basic formatting presets that you can use in your assignments, which is described in the section titled Automatic Formatting Presets.

(formatters '((0 . ,numeric) (1 . ,alphabetic-upper-case) (2 . ,alphabetic-lower-case) (3 . ,numeric-¿roman-upper-case) (4 . ,numeric-¿roman-lower-case))) —endverbatim

By default, —mags grade— uses a "." to separate each depth of the tests. For example, on a nested problem set, we would see this: x

Assignment 10: ... 3: 3.a: merge 3.b: mergesort —endverbatim If we instead set —(separator "-")— in the —test-file—:

Assignment 10: ... 3: 3-a: merge 3-b: mergesort —endverbatim

Now, we want to get the same assignment as the one mentioned in the above section, but this example will not use —test-suite—, in order to understand how —test-suite— behaves. Each of these strings should represent a path to a file that contains —(srfi :64)— test cases. *Order maters*

(test-group "Assignment 10" (include "insert.ss") (include "switcheroo.ss") (test-group (include "merge.ss") (include "mergesort.ss")) (include "bst-insert.ss")) —endverbatim

Here is the completely finished example assignment, or test-file, —a10.ss—:

(define-from-submission merge mergesort insert bst-insert switcheroo)
(separator "-") (formatters '((0 . ,numeric) (1 . ,alphabetic-upper-case) (2 . ,alphabetic-lower-case) (3 . ,numeric-¿roman-upper-case) (4 . ,numeric-¿roman-lower-case)))
(include "tree.ss") (define treeequal? ...) (define same-shape? ...) (define-equality-test test-treequal treequal?) (define-equality-test test-same-shape same-shape?)
(test-group "Assignment 10" (include "insert.ss") (include "switcheroo.ss") (test-group (include "merge.ss") (include "mergesort.ss")) (include "bst-insert.ss")) —endverbatim

Notice how we —include—, not —load—, files we want to use in our tests. You could instead include —treequal?— and —same-shape?— in —tree.ss— and you could still use them in defining your own test using —define-equality-test—. 0Using the Testing Framework: Specifics Using the improved —(srfi :64)— library provided in mags is a lot like using JUnit tests. There are many different types of tests, which take an —expected— vaue and the —actual— expression that will be tested. The —time-limit— is the time allotted for the —actual— expression to evaluate before automatically failing (if none provided, default is the —current-time-limit—.)

(test-assert [test-name] expression [time-limit]) (test-equal [test-name] expected actual [time-limit]) (test-eq [test-name] expected actual [time-limit]) (test-eqv [test-name] expected actual [time-limit]) (test-pred

pred? [test-name] expected actual [time-limit]) (test-approx [test-name] expected actual error [time-limit]) (test-set [test-name] expected-set actual [time-limit]) —endverbatim

—test-set— takes a list of expected values, and the test passes if —actual— expression's value is a member of the —expected-set—.

(test-set '(1 4 5 6) (lambda (x) 3)) ==¿ FAIL (test-set '(1 4 5 6) (lambda (x) 4)) ==¿ PASS

(test-equal "reverse on list of size 3" '(1 2 3) (reverse '(1 2 3))) —endverbatim

Tests can be between —test-begin— and —test-end— declarations, or nested inside of —test-groups—.

(test-group "Assignment 4" (test-begin "Reverse") (test-equal '(5 4 3 2 1) (reverse '(1 2 3 4 5))) (test-end "Reverse")
(test-group "Rot-2!") (test-eq "Rot-2! on a string" "ecv" (rot-2! "cat")))) —endverbatim

The object that is passed over the tests and handles the output for the tests is called a —test-runner—. A —test-runner— simply outputs its pass and fail data to the repl. When running tests, they must be executed after a —test-begin— and before a —test-end— or contained within a —test-group—, as seen above.

The —test-runner— is initialized to the —current-test-runner—, which is by default, —test-runner-verbose—, and can be changed by setting the —current-test-runner— to one of the runners defined in —(mags runners)— or by defining your own.

The full list of supported procedures, including more about test-groups, skipping tests, testing with cleanup, and customizing your own test-runners can be found on the SRFI-64 Documentation <http://srfi.schemers.org/srfi-64/srfi-64.html>. 0Define-equality-test: Examples and ImplementationAs discussed earlier, there are times when you may want to test for an ADT which you have created in another library.

This library's extenstion of —(srfi :64)— allows the use of —test-pred— which is used in the definition of —define-equality-test. It can be used like so:

¿ (test-pred pred? name expected actual) —endverbatim

To define an equality test under a different name is allowed in —mags—. This procedure, —(define-equality-test)—, takes two arguments: —test-name—, the name of the test, and —pred—, the predicate it will use. It defines a test which can be used like any other —(srfi :64)— test, but using the given —predicate— instead. Here is an example of its use with —treequal?—:

¿ (define-equality-test test-treequal treequal?) ¿ (test-treequal "These trees are treequal" '(tree 0 (leaf 3) (leaf 4)) '(tree 0) (leaf 3) (leaf 4)) PASS —endverbatim

Here is the definition of —define-equality-test—. It is implemented as a macro that can follows the same form as the —(srfi :64)— tests. We are simply redefining a given test-name, using the macro —test-name—, which uses the macro defined in the following section. This is where we implement the —pretty-print— when no name is given. The exports —test-equal—, —-eq—, and the rest are redefined later using this macro.

Define Equality Tester formDefine23Equality23Tester23form (define-syntax define-equality-test (syntax-rules () $[(_test - namepred?)(define - syntaxtest - name(syntax - rules()[(_nameexpectedtest - exprtime - limit)Definingane wtestDefining23a23new23test][(_nameexpectedtest-expr)Definingane wtestDefining23a23new23test]$ $expr)Definingane wtestDefining23a23new23test]))]))$

define-equality-test

This section specifies some details about the implementation of the protection from infinite loops in the —test-expr—. We want the tests to continue to the next test even if they are infinite loops, since —(srfi :64)— does not have support for this. This is done by guarding against it within a scheme engine. This way, we can grade an entire submission, even if it finds an infinite loop on one of the test cases. The engine will signal the error and fail the test.

When defining a new test, we need to create an expression with the given —test-expr—, the name the new test will take, and either a new —time-limit— or —current-time-limit—. If the —test-expr— times out, we raise a —timeout— exception. Else, we use an extension of the —(srfi :64)— tests, test-pred—.

Note: the —test-expr— is evaluated within the engine BEFORE it is sent to the —test-pred— procedure. This means that the —pred?— only has access to the value, not the actual expression. The —test-expr— can be retrieved from the name of the test, however. This can probably be improved.

Defining a new testDefining23a23new23test (define-syntax test-with-engine (syntax-rules () [($_e xpr$)(($make-$ engine$(lambda()test-expr))time-limit(lambda(tv)v)(lambda(c)(raise(timeoutc))))])))$(ifname(test-predpred?nameexpec$ $with-enginetest-expr))(test-predpred?expected(test-with-enginetest-expr)))$

name expected test-expr time-limit pred?

Here are basic tests for the above: we define a new test called —foobar— which simply uses —eq?— to test equality. Test equality tester formTest23equality23tester23form (test-begin "equality") (let () (define-equality-test foobar eq?) (foobar "infinte loop" '() (let loop ([i 0]) (loop (add1 i)))) (foobar "should pass" 3 3) (foobar "should fail, w/ iota" '(0 1 2 3 4) (iota 5)) (foobar "should fail numbers" 3 6) (foobar "should fail2 strings" "" "1")) (test-end "equality")

Let's throw —(define-equality-test)— into the top level. Define Equality Tester formDefine23Equality23Tester23form0Gra Usage Specifics You can use this procedure, —grade—, to load a test suite into an environment. —grade— takes two arguments: —submission—, a scheme file that is to be graded and —test-file—, that should contain —(srfi :64)— test cases. It will load the submission into a sandbox created from the —current-sandbox-name— and run the test cases against the submitted code.

Grade can take the following forms:

(grade submission)

(grade (submission test-runner)) (grade (submission test-runner) test-file) (grade (submission test-runner) test-file sandbox-name) (grade (submission test-runner) test-file sandbox-name time-limit)

(grade submission test-file) (grade submission test-file sandbox-name) (grade submission test-file sandbox-name time-limit) —endverbatim

If an argument is ommitted, —grade— will use the —(current-*)— parameter in its place.

The inputs should have the following properties:

—submission— A submission file, likely submitted by a student, which will be graded. It must be a string.

—test-file— A file that will contain SRFI :64 test cases that will be ran against the —submission— file. This must be a string.

—sandbox-name— The name of the sandbox into which the submission and test-file will be loaded. Must be a list of symbols and should represent an environment.

—time-limit— The time limit for the grading of the —submission— within the sandbox, this must be an exact positive integer. This protects the grading system from infinite loops, since they are not considered in the —(srfi :64)— library.

An example of using grade:

¿ (load "load.w") ¿ (cd "example$_t ests/example_a$10") > $(grade"menzel.ss""a10.ss"'(chezscheme)1000000)ofexpectedpa$ $(grade"submission.ss""a10.ss"'(chezscheme)1000000)FAILInsertofexpectedfailures...|endverbatim$

0Grade: ConvenienceWe have certain convenience procedures that you can use to minimize the arguments you can pass to grade, such as the —current-test-file—, —current-sandbox-name—, and —current-time-limit—. Here is an example of their usage.

Setting the sandbox environment and time limit:

¿ (current-sandbox-name '(chezscheme)) ¿ (current-time-limit 1000000) ¿ (current-test-file "a10.ss") —endverbatim

Grading a submission with the (current-test-runner):

¿ (grade "submission.ss") —endverbatim Grading a submission with a given test-runner:

¿ (grade ("submission.ss" "submission.grade" "submission.mail")) —endverbatim In the above example, "menzel.ss" and "submission.ss" are graded against the —test-file— "a10.ss" with the —sandbox-name— '(chezscheme) and —time-limit— 1000000.

Here are the definitions for these convenience parameters and others: The —current-time-limit— is a parameter that is used in the grading engine. (define current-time-limit (make-parameter 1000000 (lambda (x) (assert (integer? x)) x))) The —current-sandbox-name— is a parameter that is used as the submission's sandbox. (define current-sandbox-name (make-parameter '(chezscheme) (lambda (x) (let ([list-of-symbols? (lambda (ls) (and (list? ls) (for-all symbol? ls)))]) (assert (list-of-symbols? x))) x))) The —current-test-file— is a parameter you can use to store the location of the test file, that is, the file that contains —(srfi :64)— test cases and which will be graded against the submissions. (define current-test-file (make-parameter (lambda (file) (assert (string? file)) file))) 0Grade: ImplementationThe grader itself is implemented as a macro, As said above, It takes a —submission—, two —ports—, —test-file—, —sandbox-name—, and —time-limit—. You need to give a substantial amount of time for this to load the —submission— without a timeout - 1000000 ticks is default. It creates a sandbox from the given —sandbox-name—, and loads the submission into this new sandbox with the given —time-limit—.

If the sandbox could not load the submission file, it will handle that according to the specifications of the —Handle submission load error— chunk. Define internal grade macroDefine23internal23grade23macro (define (%grade submission p-port s-port test-file sandbox-name time-limit) (define (get-port p) (cond [(port? p) p] [(string? p) (open-file-output-port p (file-options no-fail) (buffer-mode block) (native-transcoder))] [else (error 'grade "unexpected output type, expected file or port" p)])) (let ([student-port (get-port s-port)] [port (get-port p-port)] [sndbx (sandbox sandbox-name)]) (call/cc (lambda (return) (with-exception-handler Unchecked error handlerUnchecked23error23handler (lambda () (load/sandbox submission sndbx time-limit) Set appropriate test runnerSet23appropriate23test23runner (parameterize ([counter (new-counter)] [source-directories (cons (path-parent test-file) (source-directories))] [user-sandbox sndbx]) (load test-file (let ([env (copy-environment (environment '(chezscheme) '(mags grade) '(mags sandbox)))]) (lambda (e) (eval e env))))))))))

%grade

On top of this we layer the normal multi-arity interface. Define gradeDefine23grade Define internal grade macroDefine23internal23grade23macro (... (define-syntax grade (syntax-rules (%internal) [(ₗ*submissionppsp*)*rest*...)(*grade*%*i datum*#′*submission*))(*grade*%*internal*(*submission*(*string*−*appendsubmission*".*grade*")(*string*−*appendsubmission*".*mail*" *test*−*file*)(*current*−*sandbox*−*name*)(*current*−*time*−*limit*))][(%*internal*(*submissionppsp*)*test*−*file*)(%*gradesubmissionpp file*(*current* − *sandbox* − *name*)(*current* − *time* − *limit*))][(%*internal*(*submissionppsp*)*test* − *file*(*sb* − *name*...))(%*gradesubmissionppsptest*−*file*′(*sb*−*name*...)(*current*−*time*−*limit*))][(%*internal*(*submissionppsp*)*test*− *file*(*sb* − *name*...)*time* − *limit*)(%*gradesubmissionppsptest* − *file*′(*sb* − *name*...)*time* − *limit*)])))

(grade %grade)

For grading, we need to set which test runner it will use. Right now, it always uses —test-runner-verbose—. The —port— and —student-port— are the open ports, and —p-port— and —s-port— are the original ports or filenames that were passed to —grade— from the user. Set appropriate test runner-Set23appropriate23test23runner (current-test-runner (test-runner-quiet p-port s-port))

p-port s-port

For grading, we need to make sure the test runner is appropriate. Set test-runner as current-test-runnerSet23test-runner23as23current-test-runner (if (test-runner? test-runner) (current-test-runner test-runner))

test-runner

Whenever we load an user's submission there is a chance that their submission will fail to load with some sort of problem or another. In these cases, we want to print out an error that indicates this. Unchecked error handlerUnchecked23error23handler (lambda (c) (cond [(illegal-term? c) (format p "Student uses illegal term a" (illegal-term-name c)) (format sp "You should not be using a" (illegal-term-name c))] [(timeout? c) (format (current-output-port) " Submission failed to load in time. Make sure that the (current-time-limit) is set to its default value. %") (return)] [(warning? c) (format p "There is a warning in the student file.") (format sp "There is a warning in your file. Please fix the bugs in your file until the file loads with no messages.") (return)] [else (format sp "There is a load error in your file. Please fix or comment out the errors until the file loads with no messages.n") (display-condition c p) (display-condition c sp) (display-condition c) (return)]) (display "nn"))

return p sp

The arguments passed to —grade— must satisfy the following check: Verify grader inputVerify23grader23input (let ([list-of-symbols? (lambda (ls) (and (list? ls) (for-all symbol? ls)))] [submission (syntax-¿datum submis-

sione)] [test-file (syntax-¿datum test-filee)] [sandbox-name (syntax-¿list sandbox-namee)] [time-limit (syntax-¿datum time-limite)]) (assert (string? submission)) (assert (string? test-file)) (assert (list-of-symbols? sandbox-name)) (assert (and (integer? time-limit) (exact? time-limit) (positive? time-limit))))

submissione test-filee sandbox-namee time-limite

Here we redefine the test procedures at the top-level using our —define-equality-test—, which embeds the engine. (define-equality-test test-equal equal?) (define-equality-test test-eq eq?) (define-equality-test test-eqv eqv?) (define-equality-test test-set (lambda (exp act) (let loop ([ls exp]) (cond [(null? ls) #f] [(equal? (car ls) act) #t] [else (loop (cdr ls))])))) (define-syntax test-approx (syntax-rules () [($_t$*est* − *exprexpectederror*)(*test*−*eq*#*t*(*and*(>= *test*−*expr*(−*expectederror*))(<= *test*−*expr*(+*expectederror*))))][($_n$*ametest*−
exprexpectederror)(*test*−*eqname*#*t*(*and*(>= *test*−*expr*(−*expectederror*))(<= *test*−*expr*(+*expectederror*))))][($_n$*ametest*−
exprexpectederrortime)(*test*−*eqname*#*t*(*and*(>= *test*−*expr*(−*expectederror*))(<= *test*−*expr*(+*expectederror*)))*time*)]))(
syntaxtest−*assert*(*syntax*−*rules*()[($_e$*xpr*)(*test*−*eq*#*texpr*)][($_n$*ameexpr*)(*test*−*eqname*#*texpr*)][($_n$*ameexprtime*)(*test*−
eqname#*texprtime*)]))*Let*′*spushgradetothetoplevel*.*DefinegradeDefine*23*grade*0*AccesstousercodeWhenwritingtestfiles*,

(define-from-submission ls0 ls1 bst-sort insert-in-order etc) —endverbatim

—user-sandbox— is a parameter that represents the sandbox in which the submission resides. —unbound— is a dummy procedure which unbound procedures are defined as in order to indentify them during grading. (define user-sandbox (make-parameter #f (lambda (maybe-env) (assert (or (not maybe-env) (environment? maybe-env))) maybe-env))) (define-syntax unbound (syntax-rules () [($_i$*d*)(*lambdaargs*(*raise*(*unbound*−*term*′*id*)))])(*define*−
syntaxdefine−*from*−*submission*(*syntax*−*rules*()[($_i$*d*...)(*begin*(*defineid*(*guard*(*x*[(*condition*?*x*)(*unboundid*)][*elsex*])(*eval*′*i*
sandbox))))...)]))0*AutomaticNumberingandFormattingHerewelayoutthefoundationforcustomformattingforeasilycusto*

numeric	1, 2, 3...
lphabetic-lower-case	a, b, c...
c-¿roman-lower-case	i, ii, iii, iv...
phabetic-upper-case	A, B, C...
-¿roman-upper-case	I , II, III, IV...

(define (numeric-¿roman-upper-case numeric) (define (check-tier rel? x y) (or (rel? x y) (rel? (remainder x 10) y))) (let loop ([n numeric] [roman '()]) (cond [(zero? n) (list-¿string roman)] [(check-tier = n 1) (list-¿string (cons #\I roman))] [(¡ n 4) (loop (sub1 n) (cons #\I roman))] [(check-tier = n 4) (loop (- n 4) (cons #\I (cons #\V roman)))] [(check-tier = n 5) (loop (- n 5) (cons #\V roman))] [(¡ n 9) (loop (sub1 n) (cons #\I roman))] [(check-tier = n 9) (loop (- n 9) (cons #\I (cons #\X roman)))] [(check-tier = n 10) (loop (- n 10) (cons #\X roman))] [else "too high"]))) (define (numeric-¿roman-lower-case numeric) (define (check-tier rel? x y) (or (rel? x y) (rel? (remainder x 10) y))) (let loop ([n numeric] [roman '()]) (cond [(zero? n) (list-¿string roman)] [(check-tier = n 1) (list-¿string (cons #\i roman))] [(¡ n 4) (loop (sub1 n) (cons #\i roman))] [(check-tier = n 4) (loop (- n 4) (cons #\i (cons #\v roman)))] [(check-tier = n 5) (loop (- n 5) (cons #\v roman))] [(¡ n 9) (loop (sub1 n) (cons #\i roman))] [(check-tier = n 9) (loop (- n 9) (cons #\i (cons #\x roman)))] [(check-tier = n 10) (loop (- n 10) (cons #\x roman))] [else "too high"]))) (define (alphabetic-lower-case n) (integer-¿char (+ 96 n))) (define (alphabetic-upper-case n) (integer-¿char (64 n))) (define (numeric n) n) You can change the —formatters— parameter in the same environment that will contain an unnamed —test-group—. It must be an association list that associates a depth and a corresponding formatting procedure.

(formatters '((0 . ,numeric) (1 . ,alphabetic-lower-case) (2 . ,alphbatic-upper-case) (3 . ,numeric-¿roman-lower-case) (4 . ,numeric-¿roman-upper-case))) —endverbatim

The input to —formatters— must be not null and also be an association list: Verify formatters inputVerify23formatters23input (let ([assoc-list? (lambda (ls) (and (list? ls) (not (null? ls)) (list? (cdr ls))))]) (assert (not (null? alist))) (assert (assoc-list? alist)) alist)

alist

Test group is redefined in order to allow optional names. If the name is ommitted, the automatic formatting feature will be enabled and will automatically pick the name. Define test-groupDefine23test-group (... (define-syntax test-group (syntax-rules () [($_n$*ametest*−*or*−*expr*...)(*string*?(*syntax*− > *datum*#′*name*))(*srfi* :
test − *groupnametest* − *or* − *expr*...)][($_t$*est* − *or* − *expr*...)(*let*([*i*((*counter*))])(*parameterize*([*counter*(*new* −
counter)][*current* − *names*(*consi*(*current* − *names*))])*test* − *or* − *expr*...))])))

test-group
0Formatting: ConvenienceThe following procedures are used to aid in the automatic formatting feature.

—formatters— The parameter used to hold the procedures that will be used for automatic formatting. This is an association list which associates a depth with a procedre. Each procedure takes a number and returns the desired representation of the number at that depth.

—test-suite— This is a convenience procedure created to make the automatic numbering and formatting easier to produce by removing the need to wrap each file that contained test cases into an —include—

—separator— The parameter —separator— can be used to change the symbol or string that lies between each automatically generated test-group identifer. By default, this is set to "." An example with the default —separator—:

(test-group (test-group (test-equal 'foo 'foo))) ==¿ 1.a: PASSED —endverbatim

An example with the separator set to be a dash:

(separator '-) (test-group (test-group (test-equal 'bar 'bar))) ==¿ 1-a: PASSED —endverbatim

—current-names— A parameter used to keep a running track of all the names that have been used in order to maintain depth perception. That is, to be able to find the previous name and to calculate the depth.

—counter— The parameter called when fetching the next number to be used.

—new-counter— Automatically increments the current count by 1. Begins at 1. It can be passed as an argument to —counter— to reset the counter.

—render-num— Takes a number and a depth, fetches the procedure from the —formatters— association list and applies it to the number

—render-name— A procedure that formats the —current-names— list into the proper string representation. It maps —render-num— over this list, finding the proper depth by using —iota—. An example with the default formatters: (render-name '(1 2 3)) ==¿ "1.a.A" —endverbatim

(define formatters (make-parameter '((0 . ,numeric) (1 . ,alphabetic-lower-case) (2 . ,alphabetic-upper-case) (3 . ,numeric-¿roman-upper-case) (4 . ,numeric-¿roman-lower-case)) (lambda (alist) Verify formatters inputVerify23formatters23input))) (define separator (make-parameter ".") (lambda (str) (when (symbol? str) (set! str (symbol-¿string str))) (assert (string? str)) str))) (define current-names (make-parameter '() (lambda (lst) (assert (list? lst)) lst))) (define (new-counter) (let ([i 0]) (lambda () (set! i (add1 i)) i))) (define counter (make-parameter (new-counter) (lambda (n) (assert (procedure? n)) n))) (define (render-num num depth) (let ([proc (assv depth (formatters))]) (if proc ((cdr proc) num) num))) (define (render-name names) (format (string-append " a "(*separator*)" ")(*maprender*−*num*(*reversenames*)(*iota*(*lengthnames*)))))(*define*−
syntax(*test*−*entryx*)(*syntax*−*casex*()[(*kexpr*)(*with*−*implicit*(*kinclude*)#′(*let*([*name*(*render*−*name*(*cons*((*counter*))(*curre*
names))])(*srfi* : *test*−*groupname*(*includeexpr*))))])))(*define*−*syntax*(*test*−*suitex*)(*syntax*−*casex*()[(*k*(*expr*...))(*with*−
implicit(*ktest*−*grouptest*−*suite*)#′(*test*−*group*(*test*−*suiteexpr*...)))][(*kexpr*)(*string*?(*syntax*− > *datum*#′*expr*))(*with*−
implicit(*ktest*−*entry*)#′(*test*−*entryexpr*))][(*k*(*expr*...)*rest*...)(*with*−*implicit*(*ktest*−*grouptest*−*suite*)#′(*begin*(*test*−
group(*test* − *suiteexpr*...))(*test* − *suiterest*...)))][(*kexprrest*...)(*string*?(*syntax*− > *datum*#′*expr*))(*with* −
implicit(*ktest*−*entrytest*−*suite*)#′(*begin*(*test*−*entryexpr*)(*test*−*suiterest*...)))]))*Let*′*sputourversionof*|*test*−
group|*intothetoplevelofthelibrary.Definetest* − *groupDefine*23*test* − *groupFollowingaretestcasesforformattingTestfor*
begin"*Formatters*")(*test*−*eqv*"*names*"′()(*current*−*names*))(*counter*(*new*−*counter*))(*test*−*equal*"*counter*"1((*counter*)))(*te*
equal"*render* − *num*"3(*render* − *num*30))(*test* − *equal*"*render* − *numdepth*1"#\b (render-num 2 1)) (test-equal "render-num depth 2" #\A (render-num 1 2)) (test-end "Formatters"))

The test suite for this library. (define (run-grade-tests) (test-begin "Grade") Test equality tester formTest23equality23tester23form Test formattersTest23formatters (test-end "Grade")) (mags grade)