1. Introduction. This 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 easy-to-use language to aid in the creation and deployment of test suites and problem sets. Secondly, it provides flexible means of running those tests on scheme code files.

⟨ *grade.sls* 1 ⟩ ≡

```
(library (mags grade)
  (export
    grade
    test-suite
    test-group
    test-assert
    test-equal
    test-set
    test-eqv
    test-eq
    test-approx
    test-load
    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)
  (import
    (chezscheme)
    (except
    (rename (srfi :64)
    (test-assert srfi:test-assert)
    (test-group srfi:test-group)
    (test-runner-current current-test-runner))
    test-equal
    test-eqv
    test-eq)
    (mags sandbox)
    (mags runners))
  (include "grade.ss"))
```

2. Overview. 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,entry,assert,equal,set,}`
These procedures are an extended vocabulary for test suites. They are based directly on the SRFI 64 model and use SRFI 64 as the basic framework for test reporting.

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

`define-equality-test`
This procedure allows user-defined equality predicates to be used during testing, such as equivalent or custom versions of `test-equal`.

`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 submission'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 proecdure or macro is documented where it is defined. However, the first few sections do not contain any code, and exist solely for an 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 preferred to read the documentation for each individual procedure or macro that you intend to use, and not just the examples and lighter descriptions in the next few following sections.")

3. An Example Assignment. The following few sections go into a high-level overview of the process of creating an assignment, by example. For the example assignment, each problem set needs to be created as a separate file.

We then create a file for the assignment, "a10.ss," which will simply list these files in a `test-suite`. 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!

Note that this is not how it needs to be done; your tests can simply be ran by `mags` in a simple `test-group`. However, we want to show the full power of the library by this example.

The files containing the test cases contain a `test-group` which has many `test-*` statements. We call these `Problem Sets`.

4. Problem Sets. A "problem set" is a set of test cases contained within a `test-group`. We will be creating a problem set for `insert` in this section.

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 our problem-set, "insert.ps":

```
(test-group "inser"
  (test-equal "middle, increasing" '(1 3 5 6 7 9 11) (insert < 6 '(1 3 5 7 9 11)))
  (test-equal '(5 0 -1 -2 -5) (insert > 5 '(0 -1 -2 -5)))
  (test-equal '(0 -1 -2 -4 -5) (insert > -4 '(0 -1 -2 -5)))
  (test-equal '(0 -1 -2 -5 -7) (insert > -7 '(0 -1 -2 -5)))
  (test-equal '(0 1 2 3 4 6) (let ([ls '(0 1 2 3 4)])
                               (insert < 6 ls))))
```

In the unnamed `test-equal` test cases, the name would be auto-generated by `mags`, becoming a `pretty-print` of the full tested expression. Here is an example of this:

```
(test-equal '(1 3 5 6) (insert < 6 '(1 3 5))) becomes
(test-equal "(insert < 6 '(1 3 5))" '(1 3 5 6) (insert < 6 '(1 3 5)))
```

It is important that you check in the later sections about how these testing functions work and how to create them.

5. Creating a Testing Suite. The next two sections will go over how to make a basic assignment using this library. It is important to know that a `test-group` is a function provided by SRFI 64, which simply wraps a group of tests and gives them a name.

The macro `test-suite` creates an `assignment` from any number of files which should contain (srfi :64) test cases. We can call these files `problem sets`. Here is a quick example of how `test-suite` can be used to create an assignment:

```
(test-group "Assignment 10"
  (test-suite
     "insert.ps"
     "switcheroo.ps"
     ("merge.ss"
      "mergesort.scm")
     "bst-insert.ss"))
```

Each path represents a file which contains a `test-group`, relative to the directory where the assignment file is located. The above `test-group` creates an assignment that would be recognized by a `test-runner` like so:

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

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 in this context.

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

```
(grade "submission.ss" "a10.ss" '(chezscheme) 1000000)
```

`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. Please see the `(mags sandbox)` with more information about how to create your own sandboxes.

6. What about my special ADT. Let's say our Abstract Data Type is defined in a library `tree.ss`. We can create our own test that will use any predicate we wish by defining a new test with `define-equality-test`.

`define-equality-test` takes a name and a ppredicate, and uses that predicate 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:

```
> (current-test-runner test-runner-simple)
> (define-equality-test foo-test (lambda (x y) (equal? x y)))
> (foo-test "these are equal" 3 3)
PASS
> (define-equality-test my-test-equal equal?)
> (my-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
```

You can also use `define-equality-test` to handle programs that have special behavior. The second argument, the predicate, should take two arguments, the first being the expected and the second being the actual value [note: expressions aren't passed directly to the predicate, just their values after evaluation.].

7. Building an Assignment Test File. Now that have created the problem sets, we need to now start the 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. Here is the assignment we want to create:

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

Each testing environment may need to have certain libraries loaded in order for the code to run. The `test-file`, `a10.ss`, will set up the submission's sandbox environment with the needed libraries and pull the terms from the student code into that new environment. These terms must be explicitly listed in the test-file. This is done in `(define-from-submission (<dependencies>) <variable-name> ...)`, which declares the names of the procedures from the submission that need to be available in the sandbox. *Order doesn't matter.* An example of this follows:

```
(define-from-submission ("\u\classes\lib\tree.ss")
  merge
  mergesort
  insert
  bst-insert
  switcheroo)
```

This might be extended to instead automatically pull these names from the `current-sandbox`; however, that is not currently the case. In its current state, they must be explicitly stated in the assignment file.

When you call a `test-group` without giving it a name, `mags` 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. Follows is the default settings:

```
(formatters
 `((0 . ,numeric)
   (1 . ,alphabetic-lower-case)
   (2 . ,alphabetic-upper-case)
   (3 . ,numeric->roman-upper-case)
   (4 . ,numeric->roman-lower-case)))
```

We want to create the same assignment as the one mentioned above. 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"))
```

This could also be rewritten like so, using `test-suite`:

```
(test-group "Assignment 10"
  (test-suite
    "insert.ss"
   "switcheroo.ss"
```

```
    ("merge.ss"
     "mergesort.ss")
    "bst-insert.ss"))
```

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

```
(define-from-submission
  merge
  mergesort
  insert
  bst-insert
  switcheroo)
(separator "-")

(include "tree.ss")
(define-equality-test test-treequal treequal?)
(define-equality-test test-same-shape same-shape?)

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

Notice how we `include`, not `load`, files we want to use in our tests. This is important in regards to the implementation of SRFI 64.

8. Using 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, the 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])
```

`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

\medskip
\verbatim
(test-equal "reverse on list of size 3" '(1 2 3) (reverse '(1 2 3)))
```

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 "ecv" (rot-2 "cat"))) )
```

The object that is passed over the tests and handles the output for the tests is called a `test-runner`. A `test-runner` is a hook-based object that simply collects and records the result of tests, and outputs them according to its defined callback functions. These are customizable according to your needs. The `(mags runners)` library provides a few default test runners as well as conventions to use in your own test runners.

The `test-runner` is initialized to the `current-test-runner`, which is by default, `test-runner-quiet`, and can be changed by setting the `current-test-runner` to one of the runners defined in `(mags runners)` or by defining your own. Here is an example of grading with an explicit test-runner:

```
(parameterize ([current-test-runner (test-runner-quiet "submission.graded"
                                                       "submission.mail")])
 (grade "submission.ss"))
```

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 [1].

[1] <http://srfi.schemers.org/srfi-64/srfi-64.html>

9. Define-equality-test: Implementation. As discussed earlier, there are times when you may want to test for an ADT which you have created in another library.

This library's extention of `srfi :64` allows the use of `test-pred` which is used to define new equality test forms. It can be used like so:

```
> (test-pred pred? name expected actual)
```

The 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
```

Here follows 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`, by defining a macro under that name. This uses the chunk 'Defining a new test', which is defined in the following section.

This is where we execute a `pretty-print` as the name of the test, when none is given. The exports `test-equal`, `-eq`, and others are redefined later using this macro so that all of the forms are implemented in standard.

⟨ Define Equality Tester form 9 ⟩ ≡

```
(define-syntax define-equality-test
  (syntax-rules ()
    [(_ test-name pred?)
     (define-syntax test-name
       (syntax-rules ()
         [(_ name expected test-expr time-limit)
          ⟨ Defining a new test 10 ⟩] ; name expected test-expr time-limit pred?
         [(_ name expected test-expr)
          ⟨ Defining a new test 10 ⟩] ; name expected test-expr (current-time-limit) pred?
         [(_ expected test-expr)
          ⟨ Defining a new test 10 ⟩]))])) ; (with-output-to-string (lambda () (pretty-print
    'test-expr))) expected test-expr (current-time-limit) pred?
```

This section exports define-equality-test.

This code is used in section 12.

10. This section specifies some details about the implementation of the protection from infinite loops. Tests should continue to the next test even if they produce infinite loops, and `(srfi :64)` does not have support for this, so it is done here. We guard against this within a scheme engine that times out after a certain number of ticks. The engine will signal the error and fail the test.

This chunk takes a name, the expected value, the `test-expr` to be evaluated, and either a `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`, with the given predicate.

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 in some cases, however. This might be improved.

⟨ Defining a new test 10 ⟩ ≡
```
(define-syntax test-with-engine
  (syntax-rules ()
    [(_ expr)
     ((make-engine (lambda () expr))
      time-limit
      (lambda (t v) v)
      (lambda (c) (raise (timeout c))))]))
(if name
    (test-pred pred? name expected (test-with-engine test-expr))
    (test-pred pred? expected (test-with-engine test-expr)))
```
This section captures name, expected, test-expr, time-limit, and pred?.

This code is used in section 9.

11. Here are basic tests for the above: we define a new test called `foobar` which simply uses `eq?` to test equality.

⟨ Test equality tester form 11 ⟩ ≡
```
(test-begin "equality")
(let ()
  (define-equality-test foobar eq?)
  (foobar "should fail, infinte loop" '() (let loop ([i 0]) (loop (add1 i))))
  (foobar "should pass" 3 3)
  (foobar "should fail" '(0 1 2 3 4) (iota 5)))
(test-end "equality")
```
This code is used in section 36.

12. Let's throw `(define-equality-test)` into the top level.

⟨ Define Equality Tester form 9 ⟩

13. Grading Usage. You can use this procedure, `grade`, to load test cases into a safe sandbox 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-file)
(grade submission  test-file sandbox-name)
(grade submission  test-file sandbox-name time-limit)

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

Grading a submission with a given test-runner:

```
> (parameterize ([current-test-runner (test-runner-* <args> ...)])
     (grade "submission.ss"))
```

If any 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:
> (load "load.w")
> (cd "example_tests/example_a10")
> (grade "menzel.ss" "a10.ss" '(chezscheme) 1000000)
number of expected passes ..
> (grade "submission.ss" "a10.ss" '(chezscheme) 1000000)
FAIL Insert
number of expected failures...
```

14. Grading: Convenience. We have certain convenience procedures that you can use to minimize the explicit calls to arguments of grade. These are the `current-test-file`, `current-sandbox-name`, and `current-time-limit`. Here is an example of their usage.

Here we are setting the sandbox environment, time limit, and test-file, to be used in subsequent calls to `grade`:

```
> (current-sandbox-name '(chezscheme))
> (current-time-limit 1000000)
> (current-test-file "a10.ss")
```

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

```
> (grade "submission.ss")
```

Following are the definitions for these convenience parameters.

15. 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)))
```

16. 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)))
```

17. The `current-test-file` is a parameter you can use to store the path to the test file. The `test-file` 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)))
```

18. Grade: Implementation. The grader itself is implemented as a macro, As said above, It takes a `submission`, `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 the 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 pass the fail data to the test runner to be printed out in whichever way the test-runner has specified. This is done by creating a `test-load` form, which tests if a load fails or passes.

⟨ Define internal grade macro 18 ⟩ ≡

```
(define (internal-grade submission test-file sandbox-name time-limit)
  (let ([sndbx (sandbox sandbox-name)])
    (parameterize ([counter (new-counter)]
   [current-sandbox sndbx]
   [current-submission-file submission]
   [source-directories
     (cons (path-parent test-file)
   (source-directories))])
      (load test-file
(let ([env (copy-environment
     (environment '(chezscheme)
  '(mags grade)
  '(mags sandbox)))])
  (lambda (e) (eval e env)))))))
```

This section exports internal-grade.

This code is used in section 19.

19. On top of this we layer the normal multi-arity interface.

⟨ Define grade 19 ⟩ ≡

```
⟨ Define internal grade macro 18 ⟩
(define-syntax grade
  (syntax-rules (%internal)
    [(_ submission)
     (internal-grade submission
       (current-test-file)
       (current-sandbox-name)
       (current-time-limit))]
    [(_ submission rest ...)
     (grade %internal submission rest ...)]
    [(_ %internal submission)
     (internal-grade submission
       (current-test-file)
       (current-sandbox-name)
       (current-time-limit))]
    [(_ %internal submission test-file)
     (internal-grade submission test-file
       (current-sandbox-name) (current-time-limit))]
    [(_ %internal submission test-file (sb-name ...))
     (internal-grade submission test-file '(sb-name ...)
       (current-time-limit))]
    [(_ %internal submission test-file (sb-name ...) time-limit)
     (internal-grade submission test-file '(sb-name ...) time-limit)]))
```

This section exports grade.

This code is used in section 24.

20. For grading, we need to set which test runner it will use. Right now, it always uses `test-runner-quiet`. 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 20 ⟩ ≡

```
(current-test-runner (test-runner-quiet p-port s-port))
```

This section captures p-port and s-port.

21. For grading, we need to make sure the test runner is appropriate.

⟨ Set test-runner as current-test-runner 21 ⟩ ≡

```
(if (test-runner? test-runner)
    (current-test-runner test-runner))
```

This section captures test-runner.

22. The arguments passed to `grade` must satisfy the following check:

⟨ Verify grader input 22 ⟩ ≡

```
(let ([list-of-symbols?
       (lambda (ls)
    (and
     (list? ls)
     (for-all symbol? ls)))]
      [submission (syntax->datum submissione)]
      [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))))
```

This section captures submissione, test-filee, sandbox-namee, and time-limite.

23. 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 ()
    [(_ test-expr expected error)
     (test-eq #t (and (>= test-expr (- expected error))
        (<= test-expr (+ expected error))))]
    [(_ name test-expr expected error)
     (test-eq name #t (and (>= test-expr (- expected error))
      (<= test-expr (+ expected error))))]
    [(_ name test-expr expected error time)
     (test-eq name #t (and (>= test-expr (- expected error))
      (<= test-expr (+ expected error))) time)]))
(define-syntax test-assert
  (syntax-rules ()
    [(_ expr) (test-eq #t expr)]
    [(_ name expr) (test-eq name #t expr)]
    [(_ name expr time) (test-eq name #t expr time)]))
```

24. Let's push grade to the top level.

⟨ Define grade 19 ⟩

25. Access to user code. When writing test files, it's helpful to have access to the user's code, otherwise the grader is not much use. The following form helps to extract out the user's definitions for a given procedure to be defined in the submission's sandbox. This is required in each grade file in order to specify which defintions should be graded. This ensures safety to the entire system.

```
(define-from-submission (deps ...)
  id1 id2 ...)
```

The `deps` should be a list of strings pointing to files that should be loaded before the submission is loaded into the sandbox.

Obviously, the loading of the dependencies is a little more sophisticated in the actual macro. Particularly, we want to signal errors appropriately when we load in the files, and we also want to make sure that the errors are reported via the runners, instead of possibly leaving our testing framework to report the errors. `current-sandbox` is a parameter that represents the sandbox in which the submission resides.

```
(define-syntax define-from-submission
  (syntax-rules ()
    [(_ (deps ...) id ...)
     (and (for-all identifier? #'(id ...))
  (for-all string? (map syntax->datum #'(deps ...))))
     (begin
       (revert-interaction-semantics (current-sandbox))
       (test-load deps
                  (load deps (lambda (exp) (eval exp (current-sandbox)))))
       ...
       (test-load (current-submission-file)
                  (load/sandbox
                   (current-submission-file)
                   (current-sandbox)
                   (current-time-limit)))
       (void)
       (define-from-environment id (current-sandbox))
       ...)]))
```

26. `define-from-environment` is defined below. It uses a two flags to identify a given `id-val`, `not-bound` and `not-evaled`. A given `id-val` is the value of the `id` after it has been evaled in the environment. We first set `id-val` to be `not-evaled`, and then eval it. If it comes up to be unbound, we then raise an `&unbound-term` condition. Else, we simply return the value. This could probably be refactored to use more straightforward tactics.

```
(define not-bound '(not-bound))
(define not-evaled '(not-evaled))
(define (not-bound? x) (eq? not-bound x))
(define (not-evaled? x) (eq? not-evaled x))
(define-syntax define-from-environment
  (syntax-rules ()
    [(_ id env)
     (begin
       (define id-val not-evaled)
       (define-syntax id
 (identifier-syntax
   (cond
     [(not-evaled? id-val)
      (set! id-val
(guard (c [(condition? c) not-bound]
  [else (error 'define-from-submission
  "This should never happen."
  c)])
  (eval 'id (current-sandbox))))
      (if (not-bound? id-val)
  (raise (unbound-term 'id))
  id-val)]
     [(not-bound? id-val) (raise (unbound-term 'id))]
     [else id-val])))]))
```

27. The following parameters are set in the `define-from-submission` form to be used internally in the `grade` macro as the path to the submission file that is being graded as well as the sandbox environment associated with it.

```
(define current-sandbox
  (make-parameter #f
    (lambda (maybe-env)
      (assert (or (not maybe-env) (environment? maybe-env)))
      maybe-env)))
(define current-submission-file
  (make-parameter #f
    (lambda (maybe-path)
      (assert (or (not maybe-path) (string? maybe-path)))
      maybe-path)))
(define-condition-type &load &error make-load-condition load-condition?
  (file load-condition-file))
```

28. `test-load` is a test form created in order to have access to any compilation errors when a load fails. This way, they will be accessible by the `current-test-runner`.

```
(define-syntax test-load
  (syntax-rules ()
    [(_ name exp)
     (srfi:test-assert (format "Load ~s" name)
                 (guard (c [(condition? c)
                            (raise (condition (make-load-condition exp) c))]
                           [else
                            (begin
                              (raise
                               (condition
                                (make-load-condition exp)
                                (make-irritants-condition (list c))))
                              (exit))])
                  exp #t))]))
```

29. This syntax, `unbound` is not used right now. It could probably be used in `define-from-environment` to make it simpler.

```
(define-syntax unbound
  (syntax-rules ()
    [(_ id)
     (define-syntax id
       (identifier-syntax (raise (unbound-term 'id))))]))
```

30. Automatic Numbering and Formatting. Here we implement custom formatting and easily customized numbering conventions. These are procedures that take a number and give back a value. This allows easy customization to the format of your assignment. By default, the heirarchy is as follows:

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

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

```
Assignment 10: ...
   3.a: merge
   3.b: mergesort
```

If we instead set `(separator "-")` in the `test-file`, each . would be replaced by - instead.

The following definitions are trivial, but provided to give convenience to the user.

```
(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)
```

31. You can change the `formatters` parameter in the same environment (basically, the same test-file) that contains an unnamed `test-group`. It must be an association list that associates a depth and a corresponding formatting procedure. The unnamed `test-group` will become expanded into a named test-group, where the name has been automatically generated.

```
(formatters
 '((0 . ,numeric)
   (1 . ,alphabetic-lower-case)
   (2 . ,alphbatic-upper-case)
   (3 . ,numeric->roman-lower-case)
   (4 . ,numeric->roman-upper-case)))
```

The input to `formatters` must be not null and also be an association list. The following chunk verifies this property.

⟨ Verify formatters input 31 ⟩ ≡

```
(let ([assoc-list?
  (lambda (ls)
    (and
      (list? ls)
      (not (null? ls))
      (list? (cdr ls))))])
    (assert (not (null? alist)))
    (assert (assoc-list? alist))
    alist)
```

This section captures alist.

This code is used in section 33.

32. Here, `test-group` is redefined in order to allow names to be optional. If the name is ommitted, the automatic formatting feature will be enabled and will automatically generate the name. This is done by using a simple `counter` parameter that counts the number of test-groups seen in this environment so far.

⟨ Define test-group 32 ⟩ ≡

```
(define-syntax test-group
  (syntax-rules ()
     [(_ name test-or-expr ...)
      (string? (syntax->datum #'name))
      (srfi:test-group name test-or-expr ...)]
     [(_ test-or-expr ...)
      (let ([i ((counter))])
        (parameterize ([counter (new-counter)]
                       [current-names (cons i (current-names))])
          test-or-expr ...))]))
```

This section exports test-group.

This code is used in section 34.

33. Formatting: Forms. The following procedures are used as helpers 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
```

An example with the separator set to be a dash:

```
(separator '-)
(test-group
 (test-group
  (test-equal 'bar 'bar))) ==> 1-a: PASSED
```

`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"`

```
(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 input 31 ⟩))) ; alist

(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) "~}")
  (map render-num
(reverse names)
(iota (length names)))))

(define-syntax (test-entry x)
(syntax-case x ()
  [(k expr)
    (with-implicit (k include)
      #'(let ([name (render-name (cons ((counter)) (current-names)))])
  (srfi:test-group name (include expr))))]))

(define-syntax (test-suite x)
(syntax-case x ()
  [(k (expr ...))
    (with-implicit (k test-group test-suite)
      #'(test-group (test-suite expr ...)))]
  [(k expr) (string? (syntax->datum #'expr))
    (with-implicit (k test-entry)
      #'(test-entry expr))]
  [(k (expr ...) rest ...)
    (with-implicit (k test-group test-suite)
      #'(begin
  (test-group (test-suite expr ...))
  (test-suite rest ...)))]
  [(k expr rest ...) (string? (syntax->datum #'expr))
    (with-implicit (k test-entry test-suite)
      #'(begin
  (test-entry expr)
```

```
    (test-suite rest ...)))]))
```

34. Let's put our version of `test-group` into the top level of the library.

⟨ Define test-group 32 ⟩

35. Following are test cases for the formatting.

⟨ Test formatters 35 ⟩ ≡

```
(let ()
  (test-begin "Formatters")
  (test-eqv "names" '() (current-names))
  (counter (new-counter))
  (test-equal "counter" 1 ((counter)))
  (test-equal "render-num" 3 (render-num 3 0))
  (test-equal "render-num depth 1" #\b (render-num 2 1))
  (test-end "Formatters"))
```

This code is used in section 36.

36. The test suite for this library.

```
(define (run-grade-tests)
  (test-begin "grade tests")
  ⟨ Test equality tester form 11 ⟩
  ⟨ Test formatters 35 ⟩
  (test-end "grade tests"))
```

37. Index.

MAGS: Grading Scheme Submissions

(Version 2.0)