NST 3.0 User Manual

March 23, 2011

This document is the manual and users' guide to the 3.0.x series of the NST test framework, last updated for 3.0.1. NST is a unit test system for Common Lisp which provides support for test fixture data, stateful setup and cleanup of tests, grouping of tests, and (we think!) a useful runtime interface. Suggestions and comments are welcome. The files in the NST distribution's `self-test` directory, especially `self-test/core/builtin-checks.lisp`, holds the NST tests for NST and contain many examples (some of which we have adapted for this manual). Known bugs and infelicities, platform-specific release notes, and other technical materials are available via the link on NST's CLiki page, *cliki.net/NST* .

Contents

Contributors. The primary author of both NST and this manual is John Maraist[1]. Robert P. Goldman provided guidance, comments and suggestions through the development. Other contributors include Michael J. S. Pelican, Steven A. Harp, Michael Atighetchi and Patrick Stein.

[1]Smart Information Flow Technologies, 211 North First Street, Suite 300, Minneapolis, MN 55401; *jmaraist* at *sift.info*.

1 Fixtures

Fixtures are data structures and values which may be referred to by name during testing. NST provides the ability to use fixtures across multiple tests and test groups, and to inject fixtures into the runtime namespace for debugging. A set of fixtures is defined using the `def-fixtures` macro:

```
(def-fixtures fixture-name
              ( [ :special ( NAME ... NAME
                             (:fixture NAME ... NAME) ) ]
                [ :outer FORM ] [ :inner FORM ]
                [ :setup FORM ] [ :cleanup FORM ]
                [ :startup FORM ] [ :finish FORM ]
                [ :documentation STRING ]
                [ :cache FLAG ] [ :export-names FLAG ]
                [ :export-fixture-name FLAG ]
                [ :export-bound-names FLAG ] )
  ( [ ( [ :cache FLAG ] ) ] NAME FORM )
  ...
  ( [ ( [ :cache FLAG ] ) ] NAME FORM ))
```

fixture-name The name to be associated with this set of fixtures.

inner List of declarations to be made inside the let-binding of names of any use of this fixture. Do not include the "declare" keyword here; NST adds these declarations to others, including a special declaration of all bound names.

outer List of declarations to be made outside the let-binding of names of any use of this fixture.

documentation A documentation string for the fixture set.

special Specifies a list of names which should be declared `special` in the scope within which this set's fixtures are evaluated. The individual names are taken to be single variable names. Each `(:fixture NAME)` specifies all of the names of the given fixture set. This declaration is generally optional under most platforms, but can help supress spurious warnings. Note that multiple `(:fixture NAME)`s may be listed, and these lists and the bare names may be intermixed. If only one name or fixture is specified, it need not be placed in a list

export-fixture-name When non-nil, the fixture name will be added to the list of symbols exported by the current package.

export-bound-names When non-nil, the names bound by this fixture will be added to the list of symbols exported by the current package.

export-names When non-nil, sets the default value to t for the two options above.

cache If specified with the group options, when non-nil, the fixture values are cached at their first use, and re-applied at subsequent fixture application rather than being recalculated.

When a fixture is attached to a test or test group, each `NAME` defined in that fixture becomes available in the body of that test or group as if `let*`-bound to the corresponding `FORM`. A fixture in one set may refer back to other fixtures in the same set (again *à la* `let*`) but forward references are not allowed.The four arguments `:startup`, `:finish`, `:setup` and `:cleanup` specify forms which are run everytime the fixture is applied to a group or test. The `:startup` (respectively `:finish`) form is run before fixtures are bound (after their bindings are released). These forms are useful, for example, to initialize a database connection from which the fixture values are drawn. The `:setup` form is run after inclusion of names from fixture sets, but before any tests from the group. The `:cleanup` form is normally run after the test completes, but while the fixtures are still in scope. Normally, the `:cleanup` form will not be run if the `:setup` form raises an error, and the `:finish` form will not be run if the `:startup` form raises an error; although the user is able to select (perhaps unwisely) a restart which disregards the error.The names of a fixture and the names it binds can be exported from the package where the fixture is defined using the `export-bound-names` and `export-fixture-name` arguments. The default value of both is the value of `export-names`, whose default value is `nil`.The `cache` option, if non-nil, directs NST to evaluate a fixture's form one single time, and re-use the resulting value on subsequent applications of the fixture. Note that if this value is mutated by the test cases, test behavior may become unpredictable! However this option can considerably improve performance when constant-valued fixtures are applied repeatedly. Caching may be set on or off (the default is off) for the entire fixture set, and the setting may vary for individual fixtures.Examples of fixture definitions:

```
(def-fixtures f1 ()
  (c 3)
  (d 'asdfg))
(def-fixtures f2 (:special ((:fixture f1)))
  (d 4)
  (e 'asdfg)
  (f c))
(def-fixtures f3 ()
  ((:cache t)   g (ackermann 1 2))
  ((:cache nil) h (factorial 5)))
```

To cause a side-effect among the evaluation of a fixture's name definitions, `nil` can be provided as a fixture name. In uses of the fixture, NST will replace `nil`

with a non-interned symbol; in documentation such as form `:whatis`, any `nil`s are omitted.

The `with-fixtures` macro faciliates debugging and other non-NST uses of fixtures sets:

```
(with-fixtures ( FIXTURE ... FIXTURE )
  FORM
  ...
  FORM)
```

This macro evaluates the forms in a namespace expanded with the bindings provided by the fixtures.

2 Test groups

The `def-test-group` form defines a group of the given name, providing one instantiation of the bindings of the given fixtures to each test. Groups can be associated with fixture sets, stateful initiatization, and stateful cleanup.

```
(def-test-group NAME ( FIXTURE ... FIXTURE )
  (:setup FORM ... FORM)
  (:cleanup FORM ... FORM)
  (:startup FORM ... FORM)
  (:finish FORM ... FORM)
  (:each-setup FORM ... FORM)
  (:each-cleanup FORM ... FORM)
  (:include-groups GROUP ... GROUP)
  (:documentation STRING)
  TEST
  ...
  TEST)
```

group-name Name of the test group being defined

given-fixtures List of the names of fixtures and anonymous fixtures to be used with the tests in this group.

forms Zero or more test forms, given by def-check.

setup These forms are run once, before any of the individual tests, but after the fixture names are bound.

cleanup These forms are run once, after all of the individual tests, but while the fixture names are still bound.

startup These forms are run once, before any of the individual tests and before the fixture names are bound.

finish These forms are run once, after all of the individual tests, and after the scope of the bindings to fixture names.

each-setup These forms are run before each individual test.

each-cleanup These forms are run after each individual test.

include-group The test groups named in this form will be run (respectively reported) anytime this group is run (reported).

documentation Docstring for the class.

3 Tests and test criteria

Individual unit tests are encoded with the `def-test` form:

```
(def-test ( NAME
            [ :group GROUP-NAME ] [ :setup FORM ]
            [ :cleanup FORM ] [ :startup FORM ]
            [ :finish FORM ]
            [ :fixtures ( FIXTURE ... FIXTURE ) ]
            [ :documentation STRING ] ) criterion
  FORM
  ...
  FORM)

(def-test NAME criterion
  FORM
  ...
  FORM)
```

The `SETUP`, `CLEANUP`, `STARTUP`, `FINISH` and `FIXTURES` are just as for fixtures and test groups, but apply only to the one test. The `CRITERION` is a list or symbol specifying the properties which should hold for the `FORM`s.When a test is not enclosed within a group body, a group name must be provided by the `GROUP` option. When a test is enclosed within a group body, the `GROUP` option is not required, but if provided it must agree with the group name.When there are no `SETUP`, `CLEANUP`, `STARTUP`, `FINISH` or `FIXTURES` arguments, the `NAME` may be given without parentheses. Likewise, any criterion consisting of a single symbol, e.g. `(:pass)`, may be abbreviated as just the symbol without the parentheses, e.g. `:pass`.The `:documentation` form provides a documentation string in the standard Lisp sense. Since documentation strings are stored against names,

and since the same name can be used for several tests (so long as they are all in different packages), documentation strings on tests may not be particularly useful.The `def-check` form is a deprecated synonym for `def-test`.

3.1 Basic criteria

The `true` criterion expects one form, which is evaluated at testing time; the criterion requires the result to be non-nil.

```
(:true )
```

The `eq` criterion checks a form using `eq`. The criterion argument and the form under test are both evaluated at testing time.

```
(:eq target)
```

Example:

```
(def-test eq1 (:eq 'b) (cadr '(a b c)))
```

The `symbol` criterion checks that its form under test evaluates to a symbol which is `eq` to the symbol name given as the criterion argument.

```
(:symbol name)
```

Example:

```
(def-test sym1  (:symbol a) (car '(a b c)))
```

A example of a test which fails:

```
(def-test sym1x (:symbol a) (cadr '(a b c)))
```

The `eql` criterion checks a form using `eql`. The criterion argument and the form under test are both evaluated at testing time.

```
(:eql target)
```

Example:

```
(def-test eql1 (:eql 2) (cadr '(1 2 3)))
```

The `equal` criterion checks a form using `eql`. The criterion argument and the form under test are both evaluated at testing time.

```
(:equal target)
```

The `equalp` criterion checks a form using `equalp`. The criterion argument and the form under test are both evaluated at testing time.

```
(:equalp target)
```

The `forms-eq` criterion compares its two forms under test using `eq`. The forms are both evaluated at testing time.

```
(:forms-eq )
```

Example:

```
(def-test eqforms1 :forms-eq (cadr '(a b c)) (caddr '(a c b)))
```

The `forms-eql` criterion compares its two forms under test using `eql`. The two forms under test are both evaluated at testing time.

```
(:forms-eql )
```

Example:

```
(def-test eqlforms1 :forms-eql (cadr '(a 3 c)) (caddr '(a c 3)))
```

The `forms-equal` criterion compares its two forms under test using `equal`. The forms are both evaluated at testing time.

```
(:forms-equal )
```

The `predicate` criterion applies a predicate to the result of evaluating its form under test. The criterion argument is a symbol (unquoted) or a lambda expression; at testing time, the forms under test are evaluated and passed to the denoted function. The criterion expects that the result of the function is non-nil.

```
(:predicate pred)
```

Example:

```
(def-test pred1 (:predicate numberp) 3)
```

A example of a test which fails:

```
(def-test pred2 (:predicate eql) (+ 1 2) 3)
```

The `err` criterion evaluates the form under test, expecting the evaluation to raise some condition. If the *CLASS* argument is supplied, the criterion expects the raised condition to be a subclass. Note that the name of the type should *not* be quoted; it is not evaluated.

```
(:err [ :type CLASS ])
```

Examples:

```
(def-test err1 (:err :type error) (error "this should be caught"))

(def-test err2 (:err) (error "this should be caught"))
```

The `perf` criterion evaluates the forms under test at testing time, checking that the evaluation completes within the given time limit.

```
(:perf [ :ms MILLISECS ] [ :sec SECONDS ]
       [ :min MINUTES ])
```

Example:

```
(def-test perf1 (:perf :min 2) (ack 3 5))
```

3.2 Compound criteria

The `not` criterion passes when testing according to `subcriterion` fails (but does not throw an error).

```
(:not subcriterion)
```

Example:

```
(def-test not1 (:not (:symbol b)) 'a)
```

The `all` criterion brings several other criteria under one check, and verifies that they all pass.

```
(:all subcriterion ... subcriterion)
```

Example:

```
(def-check not1 ()
    (:all (:predicate even-p)
          (:predicate prime-p))
  2)
```

The `any` criterion passes when any of the subordinate criteria pass.

```
  (:any subcriterion ... subcriterion)
```

Example:

```
(def-check not1 ()
    (:any (:predicate even-p)
          (:predicate prime-p))
  5)
```

The `apply` criterion first evaluates the forms under test, applying `FUNCTION` to them. The overall criterion passes or fails exactly when the subordinate `CRITERION` with the application's multiple result values.

```
  (:apply FUNCTION CRITERION)
```

Example:

```
(def-test applycheck (:apply cadr (:eql 10)) '(0 10 20))
```

The `check-err` criterion is like `:err`, but proceeds according to the subordinate criterion rather than simply evaluating the input forms.

```
  (:check-err criterion)
```

Example:

```
(def-test check-err1
    (:check-err :forms-eq)
  'asdfgh (error "this should be caught"))
```

The `progn` criterion first evaluates the `FORM`s in order, and then proceeds with evaluation of the forms under test according to the subordinate criterion.

```
  (:progn form ... form subcriterion)
```

Example:

```
(def-test form1 (:progn (setf zz 3) (:eql 3)) zz)
```

The `proj` criterion rearranges the forms under test by selecting a new list according to the index numbers into the old list. Checking of the reorganized forms continues according to the subordinate criterion.

```
  (:proj indices criterion)
```

Example:

```
(def-test proj-1
    (:proj (0 2) :forms-eq)
  'a 3 (car '(a b)))
```

The `applying-common-criterion` criterion applies one criterion to several pairs of criterion arguments and data forms.

```
  (:applying-common-criterion [ criterion  |
                                ( criterion arg ... arg ) ]
                              ( ( ( arg ... arg )
                                  ( form ... form ) ) ...
                                ( ( arg ... arg )
                                  ( form ... form ) ) )
                              ...
                              ( ( ( arg ... arg )
                                  ( form ... form ) ) ...
                                ( ( arg ... arg )
                                  ( form ... form ) ) ))
```

The `with-common-criterion` criterion applies one criterion to several data forms.

```
  (:with-common-criterion [ criterion  |
                            ( criterion arg ... arg ) ]
                          ( form ... form ) ...
                          ( form ... form ))
```

3.3 Criteria for multiple values

:seq `values` criterion (The is) checks each of the forms under test according to the respective subordinate criterion.

```
  (:values subcriterion ... subcriterion)
```

The `drop-values` criterion checks the primary value according to the subordinate criterion, ignoring any additional returned values from the evaluation of the form under test.

```
(:drop-values criterion)
```

The`value-list` criterion converts multiple values into a single list value.

```
(:value-list further)
```

3.4 Criteria for lists

The `permute` criterion evaluates the form under test, expecting to find a list as a result. The criterion expects to find that some permutation of this list will satisfy the subordinate criterion.

```
(:permute criterion)
```

Examples:

```
(def-test permute1 (:permute (:each (:eq 'a))) '(a a))
```

```
(def-check permute2
    (:permute (:seq (:symbol b)
                    (:predicate symbolp)
                    (:predicate numberp)))
  '(1 a b))
```

The `each` criterion evaluates the form under test, expecting to find a list as a result. Expects that each argument of the list according to the subordinate `criterion`, and passes when all of these checks pass.

```
(:each criterion)
```

Example:

```
(def-test each1 (:each (:symbol a)) '(a a a a a))
```

The `seq` criterion evaluates its input form, checks each of its elements according to the respective subordinate criterion, and passes when all of them pass.

```
(:seq subcriterion ... subcriterion)
```

Example:

```
(def-check seqcheck
    (:seq (:predicate symbolp) (:eql 1) (:symbol d))
  '(a 1 d))
```

3.5 Criteria for vectors

The `across` criterion is like `:seq`, but for a vector instead of a list.

```
  (:across subcriterion ... subcriterion)
```

Example:

```
(def-check across1
    (:across (:predicate symbolp) (:eql 1))
  (vector 'a 1))
```

3.6 Criteria for classes

The `slots` criterion evaluates its input form, and passes when the value at each given slot satisfies the corresponding subordinate constraint.

```
  (:slots ( slot-name subcriterion ) ...
          ( slot-name subcriterion ))
```

Example:

```
(defclass classcheck ()
  ((s1 :initarg :s1 :reader get-s1)
   (s2 :initarg :s2)
   (s3 :initarg :s3)))
 (def-test slot1
     (:slots (s1 (:eql 10))
             (s2 (:symbol zz))
             (s3 (:seq (:symbol q) (:symbol w)
                       (:symbol e) (:symbol r))))
   (make-instance 'classcheck
     :s1 10 :s2 'zz :s3 '(q w e r)))
```

3.7 Criteria for processes

The `process` criterion allows simple interleaving of Lisp function calls and NST checks, to allow checking of intermediate states of an arbitrarily-long process.

```
(:process form ... form)
```

This criterion takes as its body a list of forms. The first element of each form should be a symbol:

- `:eval` — Heads a list of forms which should be evaluated.
- `:check` — Heads a list of criteria which should be checked.
- `:failcheck` — If checks to this point have generated any errors or failures, then the `process` criterion is aborted.
- `:errcheck` — If checks to this point have generated any errors (but not failures), then the `process` criterion is aborted.

The `:process` criterion takes no value arguments in a `def-test`.Example:

```
(def-test process-1
    (:process (:eval (setf zzz 0))
              (:check (:true-form (eql zzz 0)))
              (:eval (incf zzz))
              (:check (:true-form (eql zzz 1)))
              (:eval (incf zzz))
              (:check (:true-form (eql zzz 2)))))
```

3.8 Programmatic and debugging criteria

The `pass` is a trivial test, which always passes.

Example:

```
(def-test passing-test :pass 3 4 "sd")
```

The `info` criterion adds an informational note to the check result.

```
(:info string subcriterion)
```

Example:

```
(def-test known-bug (:info "Known bug" (:eql 3)) 4)
```

The `dump-forms` criterion is for debugging NST criteria. It fails after writing the current forms to standard output.

```
(:dump-forms blurb)
```

The `warn` criterion issues a warning. The format string and arguments should be suitable for the Lisp `format` function.

```
(:warn format-string form ... form)
```

Example:

```
(:warn "~{}d is not a perfect square" 5)
```

4 Defining test criteria

The criteria used in test forms decide whether, when and how to use the forms under test and the forms and subcriteria provided to each test criterion. Criteria receive their arguments as forms, and may examine them with or without evaluation, as the particular criterion requires. NST provides two mechanisms for defining new criteria, and a number of support functions for use within these definitions. The simpler, but more limited, way to define a new criterion is by specifying how it should be rewritten to another criterion. The `def-criterion-alias` macro provides this mechanism, which we discuss in Section 4.1. The `def-criterion` macro provides the more general mechanism for criteria definition, where Lisp code produces a result report from the forms under test and criterion's forms and subcriteria. We discuss `def-criterion` in Section 4.2. We discuss the NST API for creating these result reports in Section 4.3, and for recursive processing of subcriteria in Section 4.4.

4.1 Aliases over criteria

The simplest mechanism for defining a new criterion involves simply defining one criterion to rewrite as another using `def-criterion-alias`:

```
(def-criterion-alias ( name arg ... arg )
  [ documentation ]
  expansion)
```

The body of the expansion should be a Lisp form which, when evaluated, returns an S-expression quoting the new criterion which the rewrite should produce. The `arg`s are passed as for Lisp macros: they are not evaluated and are most typically comma-inserted into a backquoted result. For example:

```
(def-criterion-alias (:forms-eq) `(:predicate eq))
(def-criterion-alias (:symbol name) `(:eq ',name))
```

4.2 Reporting forms

NST provides functions both for building test reports, and for adding information to a report.

- The `make-success-report` function indicates a successful test result.

  ```
  (make-success-report )
  ```

  Note that some older examples show `(make-check-result)`, `(emit-success)` or `(check-result)`. The former is an internal function and should not be used from outside the core NST files. The latter two are deprecated.

- The `make-failure-report` function returns a report of test failure.

  ```
  (make-failure-report [ :format format-string ]
                       [ :args arg-form-list ])
  ```

  The `format-string` and `args` are as to the Common Lisp function `format`. The `emit-failure` function is an older, deprecated version of this function.

- Function `make-warning-report` is like `make-failure-report`, but provides supplimentary information as a warning.

  ```
  (make-warning-report [ :format format-string ]
                       [ :args arg-form-list ])
  ```

  The `emit-warning` function is an older, deprecated version of this function.

- Function `make-error-report` produces a report of an error during test execution.

  ```
  (make-error-report [ :format format-string ]
                     [ :args arg-form-list ])
  ```

- The `add-error` function adds an error note to a result record.

  ```
  (add-error result-report
             [ :format format-string ]
             [ :args argument-list ])
  ```

- The `add-failure` function adds a failure note to a result record.

```
(add-failure result-report
             [ :format format-string ]
             [ :args argument-list ])
```

- The `add-info` function adds auxiliary information to a result record.

```
(add-info result-report info-item)
```

4.3 Processing subcriteria

The criterion itself can contain *subcriteria* which can be incorporated into the main criterion's assessment. NST provides two functions which trigger testing by a subcriterion, each returning the check's result report.

The `check-criterion-on-value` function can be called from within a criterion body to verify that a value adheres to a criterion.

```
(check-criterion-on-value criterion value)
```

The `check-criterion-on-form` function verifies that an unevaluated form adheres to a criterion.

```
(check-criterion-on-form criterion form)
```

4.4 General criteria definitions

The `def-criterion` macro defines a new criterion for use in NST tests. These criteria definitions are like generic function method definitions with two sets of formal parameters: the forms provided as the actual parameters of the criterion itself, and the values arising from the evaluation of the forms under test.

```
(def-criterion ( name criterion-lambda-list
                 values-lambda-list )
  [ documentation ]
  form
  ...
  form)
```

name Name of the criterion.

criterion-lambda-list Lambda list for the arguments to the criterion. Optionally, the first element of the list is a symbol specifying the parameter-passing semantics for the criterion arguments: `:values` for call-by-value, or `:forms` for call-by-name (the default). The list may include the keywords `&key`, `&optional`, `&body` and `&rest` but may not use `&whole` or `&environment`. Apart from this restriction, in the former case the list may be any ordinary lambda list as for `defun`, and in the latter case the list may be any macro lambda list as for `defmacro`.

values-lambda-list Lambda list for the forms under test. Optionally, the first element of the list is a symbol specifying the parameter-passing semantics for the criterion arguments: `:values` for call-by-value (the default), or `:form` for call-by-name. In the former case, the list may include the keywords `&key`, `&optional`, `&body` and `&rest`, but not `&whole` or `&environment`; apart from that restriction, list may be any ordinary lambda list as for `defun`. In the latter case, the remainder of the list must contain exactly one symbol, to which a form which would evaluate to the values under test will be bound.

If the criterion ignores the values, then instead of a lambda list, this argument may be the symbol `:ignore`. On many platforms, listing a dummy parameter which is then `declare`d `ignore` or `ignorable` will produce a style warning: the body of a `def-criterion` should not be assumed to correspond directly to the body of a `defmethod`; in general there will be surrounding `destructuring-bind`s.

documentation An optional documentation string for the criterion.

form The body of the criterion definition should return a test result report contructed with the `make-success-report`, etc. functions.

Examples:

```
(def-criterion (:true () (bool))
  (if bool
      (make-success-report)
      (make-failure-report :format "Expected non-null, got: ~s"
                   :args (list bool))))

(def-criterion (:eql (target) (actual))
  (if (eql (eval target) actual)
      (make-success-report)
      (make-failure-report :format "Not eql to value of ~s"
                   :args (list target))))
```

5 Verifying invariants against sampled data

The `sample` criterion provides random generation of data for validating program properties. Our approach is based on Claessen and Hughes's Quickcheck[2].

This style of testing is somewhat more complicated than specific tests on single, bespoke forms. There are two distinct efforts, which we address in the next two sections: describing how the sample data is to be generated, and specifying the test itself.

5.1 Generating sample data

Data generation is centered around the generic function `arbitrary`.

This function takes a single argument, which determines the type of the value to be generated. For simple types, the name of the type (or the class object, such as returned by `find-class`) by itself is a complete specification. For more complicated types, `arbitrary` can also take a list argument, where the first element gives the type and the remaining elements are keyword argument providing additional requirements for the generated value.

NST provides method of *arbitrary* for many standard Lisp types, listed in Table 1. Types in the first column — the standard numeric types plus the common supertype `t` are not associated with additional keyword arguments.

```
(nst:arbitrary t)
(nst:arbitrary 'complex)
(nst:arbitrary 'integer)
(nst:arbitrary 'ratio)
(nst:arbitrary 'single-float)
```

Keyword arguments for other NST-provided type specifiers are as follows:

- Types `character` and `string`:
  - Argument `noncontrol`. Excludes the control characters associated with ASCII code 0 through 31.
  - Argument `range`. Allows the range of characters to be restricted to a particular subset:

[2]Koen Claessen and John Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," from *Proceedings of the International Conference on Functional Programming*, 2000. QuickCheck papers, code and other resources are available at *www.cs.chalmers.se/ rjmh/QuickCheck* .

[3]Not available on Allegro Lisp.

Standard Lisp types					Other types
number	character	symbol	cons	hash-table	scalar
real	string		list		
rational			vector		
integer			array		
float				t	
fixnum					
bignum					
ratio					
short-float[3]					
single-float					
double-float[3]					
long-float					
complex					
Considered scalar					

Table 1: NST provides methods of generic function `arbitrary` generating values of the types in this table.

Value	Meaning
`:standard`	Codes up to 96
`:ascii`	Codes through 127
`:ascii-ext`	Codes through 255

Omitted or with any other value, characters with any code up to `char-code-limit` can result. Examples:

```
(nst:arbitrary 'character)
(nst:arbitrary '(character :noncontrol t
                           :range :standard))
```

- Type `symbol`.
  - Argument `existing`. If non-nil, requires that the result be a previously-interned symbol.
  - Argument `exported`. Requires that the result be not only a previously-interned symbol, but also one exported by its package. Ignored if `existing` is explicitly set to nil.
  - Argument `package`. Specifies the package from which the symbol will be generated. If omitted, a package is selected at random from the existing ones.
  - Argument `nonnull`. If non-nil, allows `arbitrary` to ignore other restriction to guarantee returning a non-nil symbol. When null, `arbitrary` may return nil.
  - Argument `gensym`. If non-nil, and if `arbitrary` is explicitly set to nil, returns a new uninterned symbol.

- Type `cons`.
  - Arguments `car` and `cdr` should be additional type specifications, used direct the generation of respectively the left and right elements of the result. Each defaults to `t`.
- Type `list` and `vector`.
  - Argument `length` specifies the length of the structure. If omitted, will be randomly generated.
  - Argument `elem` directs the generation of the container's elements. For both, the default element type is `t`.
- Type `array`.
  - Argument `elem`. As for `list` and `vector`.
  - Argument `dimens`. Should be a list of nonnegative integers specifying the length of each dimension of the array. If omitted, will be randomly generated.
  - Argument `rank`. Specifies the number of dimensions. If omitted but `dimens` is given, will be set to the length of `dimens`. If both `rank` and `dimens` are omitted, then both are randomly generated.
- Type `hash-table`.
  - Argument `size`. Specifies the number of entries in the table. If omitted, will be randomly generated.
  - Argument `test`. Specifies the hash table's test function. If omitted, will be randomly selected from `eq`, `eql`, `equal` and `equalp`.
  - Arguments `key` and `val` direct the generation of the table's keys and values, respectively. For the keys, the default element type is textttt when the test function is texttteq or texttteql, and textttscalar otherwise. For the values, the default element type is textttt.

Beyond those standard Lisp types, NST provides the type `scalar` as a supertype of the numeric types plus `character`, `string` and `symbol`. Users may extend this definition to include additional type specifications, as we discuss below. Types are not associated with `scalar` are referred to as `compound` (although there is no corresponding type specification). To avoid generating structures too large to hold in memory, NST provides the global variable `*max-compound-structure-depth*` and the macro `compund-structure`.

The `*max-compound-structure-depth*` variable sets the maximum nesting depth of compound data structures: beyond that depth, `scalar` rather than `t` is the default element generator. This restriction does not apply to explicitly specified element types, only to the use of defaults.

The `compound-structure` macro wraps substructure which should be considered compound for the limits set by `*max-compound-structure-depth*`.

New type specifications for invariant-testing. are defined with the `def-arbitrary-instance-type` macro.

```
(def-arbitrary-instance-type ( spec-name
                               [ :params formals ]
                               [ :scalar bool ]
                               [ :key key ] )
  form
  ...
  form)
```

formals Formal parameter definition used to pass subcomponent types.

scalar When a non-null value is provided for the `scalar` argument, the new specifier is taken to be generable by the `scalar` specification.

```
(def-arbitrary-instance-type (ratio :scalar t)
  (/ (arbitrary 'integer)
     (let ((raw (arbitrary (find-class 'integer))))
       (cond
         ((< raw 0) raw)
         (t (+ 1 raw))))))
```

key The `key` argument gives a list of keyword arguments which may accompany the new specification. For the `cons` type, keyword arguments allow specifications for the left and right components:

```
(def-arbitrary-instance-type (cons :key ((car t car-supp-p)
                                         (cdr t cdr-supp-p)))
  (compound-structure
   (when (and (not car-supp-p)
              (>= *current-compound-structure-depth*
                  *max-compound-structure-depth*))
     (setf car 'scalar))
   (when (and (not cdr-supp-p)
              (>= *current-compound-structure-depth*
                  *max-compound-structure-depth*))
     (setf cdr 'scalar))
   (cons (arbitrary car) (arbitrary cdr))))
```

form Construct and return (as if through `progn`) the arbtrary instance.

5.2 Invariants as tests

Invariants to be tested, and the domains over which they range, are specified with the `sample` criterion:

```
(:sample [ :verify FORM ] [ :value LAMBDA-LIST ]
         [ :domains ( NAME SPEC ) ... ( NAME SPEC ) ]
         [ :where FORM ]
         [ :where-ignore ( NAME ... NAME ) ]
         [ :where-declare ( DECLARATION ...
                            DECLARATION ) ]
         [ :sample-size NUMBER ]
         [ :qualifying-sample NUMBER ]
         [ :max-tries NUMBER ])
```

verify The the expression to be (repeatedly) evaluated, which is expected always to return a non-null value. This is the sole required argument, although in any particular use it is unlikely to be the only argument given.

domains Declares the variables in the `verify` expression which are to be given multiple randomized values. The default value is `nil`, denoting an empty list.

value A lambda list to which the values given by the argument form should be applied. The default value is `nil`, denoting no such arguments.

where A condition which determines the validity of the input argument. For example, the condition would assert that a number is positive in an application where a negative value would be known to cause a failure. The default value is `t`, allowing any values.

where-ignore List of domain variables which are not mentioned in the `where` clause. These names will be declared as ignored in appropriate bindings, suppressing warnings under Lisps which check for such things in interpreted expressions. This list need not be given explicitly when no `where` argument is given. Similarly, the `where-declare` argument accepts a list of declarations to be associated with the `where` form.

sample-size Gives the base specification of the number of value sets which will be generated. Two further arguments have some bearing on the number of generation attempts when the `where` argument is non-`t`. The `qualifying-sample` argument gives the minimum acceptable size of actual tested values, not counting sets rejected via the `where` expression. The `max-tries` argument gives the maximum number of value sets to be generated.

Examples:

```
(:sample :sample-size 10
  :domains ((x (list :elem symbol)))
  :verify (equal x (reverse (reverse x))))

(:sample :domains ((x real))
  :where (> x 1)
  :verify (< (sqrt x) x)
  :sample-size 10
  :max-tries 12)
```

6 The runtime system

The runtime system provides several operations for scheduling and running tests, and debugging failing and erring tests.

User-level NST operations are accessible from the REPL via the `nst-cmd` macro.

```
(nst-cmd nst-command arg ... arg)
```

Where a particular system supports the facility,[4] the top-level alias `:nst` provides a shorthand to this function.

For the sake of brevity we use the `nst` shorthand below.

The `:help` command gives a complete inventory of runtime system commands.

```
:nst :help
```

There are a number of commands for running tests, but most of the time only one will be needed:

- The `:run` command executes all tests in the named package, or in the named group, or runs the named test. It is not necessary to prefix the name with a package prefix. The name does not need to be prefix-qualified, but if the name is ambiguous then `:run` will simply report the possible interpretations.

  ```
  :nst :run name
  ```

- The `:run-package` command executes all tests associated with groups in the named packages, and reports the test results afterwards.

  ```
  :nst :run-package package ... package
  ```

[4] Currently Allegro, and SBCL under ACL-REPL.

- The `:run-group` command executes all tests associated with the name groups, and reports the test results afterwards. The group name should be package-qualified.

  ```
  :nst :run-group group ... group
  ```

- The `:run-test` command executes the given test. Both the group and test name should be package-qualified.

  ```
  :nst :run-test group test
  ```

One further command for running a test is useful when writing and debugging the tests themselves:

- The `apply` criterion first evaluates the forms under test, applying `FUNCTION` to them. The overall criterion passes or fails exactly when the subordinate `CRITERION` with the application's multiple result values.

  ```
  (:apply FUNCTION CRITERION)
  ```

  Example:

  ```
  (def-test applycheck (:apply cadr (:eql 10)) '(0 10 20))
  ```

- The `:apply` command assesses whether a test criterion prints the uses to which a particular name has been applied in an NST session.

  ```
  :nst :apply name
  ```

There are two commands for (re)printing the results of tests:

- The `:report` command summarizes successes, failures and errors in tests. It reports either for the named artifact, or for all recently-run tests.

  ```
  :nst :report

  :nst :report package

  :nst :report group

  :nst :report group test
  ```

- The `:report` command gives detailed information about individual test results.

  ```
  :nst :detail

  :nst :detail package

  :nst :detail group

  :nst :detail group test
  ```

The `undef` and `clear` commands allow removal of groups, tests and results.

- The `:undef` command retracts the definition of an NST group or test.

  ```
  :nst :undef group-name

  :nst :undef group-name test-name
  ```

  Currently, NST does require that the symbols passed to `undef` be correctly package-qualified.

- The `:clear` command empties NST's internal record of test results.

  ```
  :nst :clear
  ```

The `set` and `unset` display and adjust NST's configuration.

- The `:set` command assigns or displays the values of NST runtime switches.

  ```
  :nst :set property

  :nst :set property value
  ```

- The `:unset` command clears the values of NST runtime switches.

  ```
  :nst :unset property
  ```

There are currently three properties which can be manipulated by `set` and `unset`:

- The `:verbosity` switch controls the level of NST's output.

```
:nst :set :verbose setting
```

Valid settings are:

- `:silent` (aka `nil`)
- `:quiet` (aka `:default`)
- `:verbose` (aka `t`)
- `:vverbose`
- `:trace`

The `:report` and `:detail` commands operate by setting minimum levels of verbosity.

- The `:debug-on-error` switch controls NST's behavior on errors. When non-nil, NST will break into the debugger when it encounters an error.

  ```
  :nst :set :debug-on-error flag
  ```

  The `:debug` command is a short-cut for setting this property.

- The `:debug-on-fail` switch controls NST's behavior when a test fails When non-nil, NST will break into the debugger when it encounters a failing test.

  ```
  :nst :set :debug-on-fail flag
  ```

  This behavior is less useful than it may seem; by the time the results of the test are examined for failure, the stack from the actual form evaluation will usually have been released. Still, this switch is useful for inspecting the environment in which a failing test was run.Note that both `:debug-on-error` and `:debug-on-fail` apply in the case of an error; if the latter is set but the former is not, then the debugger will be entered after an erring test completes.The `:debug` command is a short-cut for setting this property.

- The `:backtraces` switch, when non-`nil`, directs NST to attempt to capture the Lisp backtrace of errors in tests.

  ```
  :nst :set :backtraces flag
  ```

  This property is only available on platform which allow programmatic examination of backtraces, which is not standardized in Common Lisp; currently we have implemented this feature on Allegro only.This property has a complicated default setting. Firstly, if the symbol `'common-lisp-user::*nst-generate-backtraces*` is bound when NST loads, NST will use its value as the initial value for this property. Otherwise by default, on MacOS systems the property initializes to `nil` because of a known error on that system, but this setting can be overriden by the property `:nst-unsafe-allegro-backtraces`. Finally, if none of these issues apply, the initial value is `t`.

The above NST commands are governed by a number of global variables. In general, interactive use of NST should not require direct access to these variables, but when automating NST operations may require changing, or creating a new dynamic scope for, their settings.

- User variable `*debug-on-error*`: if non-nil, will break into the Lisp REPL debugger upon encountering an unexpected error. If nil, will record the error and continue with other tests.
- User variable `*debug-on-fail*`: if non-nil, will break into the Lisp REPL debugger upon encountering a test which fails. If nil, will record the failure and continue with other tests. This variable is useful inspecting the dynamic environment under which a test was evaluated.
- User variable `*default-report-verbosity*` determines the default value for *nst-verbosity* when printing reports (2 by default).
- User variable `*nst-output-stream*` determines the output stream to which NST should print its output (`*standard-output*` by default).

Fixtures can be *opened* into the interactive namespace for debugging with the `:nst :open`
Syntax: `:nst :open FIXTURE-NAME FIXTURE-NAME ...  FIXTURE-NAME`
Example:

```
CL-USER(75): (nst:def-fixtures small-fixture ()
               (fix-var1 3)
               (fix-var2 'asdfg))
NIL
CL-USER(76): (boundp 'fix-var1)
NIL
CL-USER(77): :nst :open small-fixture
Opened fixture SMALL-FIXTURE.
CL-USER(78): fix-var1
3
CL-USER(79):
```

Fixtures can be opened into a different package than where they were first defined, but these bindings are in addition to the bindings in the original package, and are made by a symbol import to the additional package.

Calling `nst` or `nst-cmd` without a command argument repeats the last test-executing command.

7 Integration with ASDF

NST's integration with ASDF is a work in progress. This section described the current integration, the ways we expect it to change, and a less-flexible and lower-level, but likely more stable, alternative integration technique.

7.1 NST's ASDF systems

From version 1.2.2, the system `:asdf-nst` provides two classes for ASDF system definitions, `asdf:nst-test-runner` and `asdf:nst-test-holder`.

Up to NST 1.2.1 `:asdf-nst` provided a single class `asdf:nst-testable`, and in the future we plan to reunify the current two classes into a single class again. However our first implementation required NST to be loaded even when a system was *not* being tested, because we had no way to distinguish the source code associated with testing from production code. We plan to solve this problem with a new file type `nst-file` in a future version of NST. This file type would *not* be compiled or loaded for the `compile-op` or `load-op` of the system, only for its `test-op`.

7.1.1 Test-running systems

ASDF systems of the `asdf:nst-test-runner` class do not themselves contain NST declarations in their source code, but may identify other systems which do, and which should be tested as a part of testing the given system. These systems also allow local definitions of NST's configuration for the execution of their tests.

Specify that a system runs NST tests by providing `:class asdf:nst-test-runner` argument to `asdf:defsystem`. Use the `:nst-systems` argument to name the systems which house the actual unit tests:

- `:nst-systems (`*system system* ⋯ *system*`)`

  Specifies a list of other systems which should be tested when testing this system. These other systems do *not* otherwise need to be identified as a dependency of this system (nor, for that matter, does `:nst` itself); they will be loaded upon `test-op` if they are not yet present.

Another optional argument to an `nst-test-runner` system definition is:

- `:nst-init (`*arg-list ... arg-list*`)`

  Initializing arguments to NST, to be executed after this system is loaded. Each `arg-list` is passed as the arguments as if to a call to the `nst-cmd` macro.

- `:nst-debug-config` *form*

  NST debugging customization for this system. The `FORM` Should be an expression which, when evaluated, returns a list of keyword arguments; note that to give the list itself, it must be explicitly quoted, *which is a change of behavior from pre-1.2.2 versions.*

- `:nst-debug-protect (`*symbol ... symbol*`)`

  Gives a list of variables whose values should be saved before applying any configuration changes from `:nst-debug-config`, and restored after testing.

- `:nst-push-debug-config` *t-or-nil*

  If non-nil, then when this system is loaded its `:nst-debug` and `:nst-debug-protect` settings will be used as NST's defaults.

7.1.2 Test-containing systems

The `asdf:nst-test-holder` class is a subclass of `nst-test-runner` for systems which are not only tested via NST, but also contains NST tests in their source code.

Specify that a system defines NST tests by providing `:class asdf:nst-test-holder` to `asdf:defsystem`. The arguments for `asdf:nst-test-runner` may be used for `asdf:nst-test-holder`, as well as the following:

- `:nst-packages (`*package package* ⋯ *package*`)`

  When the system is tested, all groups and tests in the named packages should be run.

- `:nst-groups ((`*package group*`)` ⋯ `(`*package group*`))`

  When the system is tested, tests in the named groups should be run. Naming the package separately from the group and test in this argument (and in the similar arguments below) allows the group to be named before its package is necessarily defined.

- `:nst-tests ((`*package group test*`)` ⋯ `(`*package group test*`))`

  When the system is tested, all the named tests should be run.

The next three arguments to an `nst-testable` system are mutually exclusive, and moreover exclude any of the above group or `:nst-systems`:

- `:nst-package` *package*

  When the system is tested, all groups and tests in the named package should be run.

```
;; NST and its ASDF interface must be loaded
;; before we can process the defsystem form.
(asdf:oos 'asdf:load-op :asdf-nst)

(defsystem :mnst
    :class nst-test-holder
    :description "The NST test suite's self-test."
    :serial t
    :nst-systems (:masdfnst)
    :nst-groups ((:mnst-simple . g1)
                 (:mnst-simple . g1a)
                 (:mnst-simple . g1a1)
                 (:mnst-simple . core-checks))
    :depends-on (:nst)
    :in-order-to ((test-op (load-op :mnst)))
    :components ((:module "core"
                  :components ((:file "byhand")
                               (:file "builtin-checks")))))
```

Figure 1: Definitions of `nst-testable` ASDF systems.

- `:nst-group` (*package group*)

  When the system is tested, all tests in the named group should be run.

- `:nst-test` (*package group test*)

  When the system is tested, the given test should be run.

Figure 1 gives examples of `nst-testable` ASDF system definitions.

7.2 An alternate ASDF integration technique

We plan to deprecate and then remove `asdf:nst-test-holder` and `nst-test-runner` once we have implemented a unified replacement for them. To avoid the possibility of a bit-rotted test scheme, the link between a system and its unit tests can be made explicit by providing methods for ASDF generic functions which make calls to the NST API. Specifically:

- A method of the ASDF `asdf:perform` generic function specialized to the `asdf:test-op` operation and the system in question will be executed to test a system. So an appropriate method definition would begin:

  ```
  (defmethod asdf:perform ((op asdf:test-op)
                           (sys (eql (asdf:find-system
                                      :SYSTEM-NAME))))
  ```

- NST API functions for running tests are:
- `nst:run-package`
- `nst:run-group`
- `nst:run-test`
- The main NST API function for printing the results of testing is `asdf:report-multiple`. In situations where only a single package, group or test is associated with a system, one of the following function may be more convenient:
- `nst:report-package`
- `nst:report-group`
- `nst:report-test`

When providing an explicit `asdf:perform` method, it is also necessary to explicitly list system dependencies to NST and to the other systems which contain the tested system's unit test definitions.

A The NST API

A.1 Primary macros

`def-criterion` — §4.4, p. 16.

`def-criterion-alias` — §4.1, p. 14.

`def-fixtures` — §1, p. 2.

`def-test` — §3, p. 5.

`def-test-group` — §2, p. 4.

A.2 Functions used in criteria definitions

`add-error` — §4.2, p. 15.

`add-failure` — §4.2, p. 16.

`add-info` — §4.2, p. 16.

`check-criterion-on-form` — §4.3, p. 16.

`check-criterion-on-value` — §4.3, p. 16.

`make-error-report` — §4.2, p. 15.

`make-failure-report` — §4.2, p. 15.

A.3 Programmatic control of testing and output

A.4 Testing randomized samples

A.5 Method-based tests on objects

A.6 Other symbols

A.7 Deprecated macros, functions and variable

B Output to JUnit

NST reports can be formatted as XML for use with JUnit, although the API for this feature is underdeveloped.

The `junit-results-by-group` function writes the NST test results in JUnit XML format, organized by group, aligning test groups with Java classes, and individual tests with `@Test` methods.

```
(junit-results-by-group [ :verbose flag ]
                        [ :dir directory ]
                        [ :file filespec ]
                        [ :stream stream ]
                        [ :if-dir-does-not-exist bool ]
                        [ :if-file-exists bool ])
```

Either `:dir` and `:file` options, or the `:stream` option, but not both, should be used to specify the target for XML output; if none of the three options are given, the function will write to `*standard-output*`.

Function `nst-junit-dump` pushes the entire NST state to a JUnit XML file whose stream is specified by its argument.

C Inheritance-based test methods

This feature is in-progress. It currently does not work under Lispworks or Clisp, and details of the API may change in subsequent versions.

For testing objects in a class hierarchy NST offers xUnit-style test methods dispatching on different classes. The idea is that an object should have all relevant tests applied to it without requiring that the tests be explicitly enumerated in the test definition: all tests applicable to an object's class, or to any of its superclasses, should be discovered and run.

Our running examples of this section are tests on objects of these four classes:

```
(defclass top-cls ()
```

```
     ((tc1 :initarg :tc1 :reader tc1)
      (tc2 :initarg :tc2 :reader tc2)))

(defclass mid-cls (top-cls)
     ((mc1 :initarg :mc1 :reader mc1)
      (mc2 :initarg :mc2 :reader mc2)))

(defclass side-cls ()
     ((sc1 :initarg :sc1 :reader sc1)
      (sc2 :initarg :sc2 :reader sc2)))

(defclass bot-cls (mid-cls side-cls)
     ((bc1 :initarg :bc1 :reader bc1)
      (bc2 :initarg :bc2 :reader bc2)))
```

C.1 Declaring methods

There are two macros which define a particular method of a generic test function.

The `def-test-method-criterion` macro provides a simple facility for defining a generic test function method in terms of an NST criterion.

```
(def-test-method-criterion function-name class-name
  criterion)
```

function-name The name of the test function for which we are defining a method.

class-name The class for which we are defining a method.

criterion The criterion to be applied to members of the class.

For example:

```
(nst:def-test-method-criterion for-clses top-cls
      (:predicate (lambda (tc) (< (tc1 tc) (tc2 tc)))))
```

The `def-test-generic` declares a generic test function.

```
(def-test-generic function-name)
```

For example,

```
(nst:def-test-generic for-clses)
```

The `def-test-method` defines a general method for a generic test function.

```
(def-test-method function-name
                 ( test-value class-name )
  form
  ...
  form)
```

function-name The name of the test function for which we are defining a method.

test-value Formal parameter to which the value under test will be bound.

class-name The class for which we are defining a method.

The method body should return a test result report, constructed with `make-success-result`, etc.For example:

```
(nst:def-test-method for-clses (o mid-cls)
  (with-slots (mc1 mc2) o
    (cond
      ((< mc1 mc2) (make-success-report))
      (t (make-failure-report :format "~d not < ~d" :args (list mc1 mc2))))))
(nst:def-test-method for-clses (o side-cls)
  (with-slots (sc1 sc2) o
    (cond
      ((eql sc1 sc2) (make-success-report))
      (t (make-failure-report :format "~d not eql ~d" :args (list sc1 sc2))))))
```

C.2 Invokng methods

The `:methods` criterion runs the test functions applicable to the value under test.

For example:

```
(def-test-group method-tests ()
  (def-test t-p :methods (make-instance 'top-cls :tc1 0 :tc2 2))
  (def-test m-p :methods (make-instance 'mid-cls :tc1 0 :tc2 2 :mc1 0 :mc2 2))
  (def-test s-p :methods (make-instance 'side-cls :sc1 1 :sc2 1))
  (def-test b-p :methods (make-instance 'bot-cls
                           :tc1 0 :tc2 2 :mc1 0 :mc2 2 :sc1 1 :sc2 1))
  (def-test t-f :methods (make-instance 'top-cls :tc1 4 :tc2 2))
  (def-test m-f-t  :methods (make-instance 'mid-cls
                              :tc1 4 :tc2 2 :mc1 0 :mc2 2))
```

```
(def-test m-f-m  :methods (make-instance 'mid-cls
                            :tc1 0 :tc2 2 :mc1 4 :mc2 2))
(def-test m-f-mt :methods (make-instance 'mid-cls
                            :tc1 4 :tc2 2 :mc1 4 :mc2 2))
(def-test s-f :methods (make-instance 'side-cls :sc1 1 :sc2 3))
(def-test b-f-t :methods (make-instance 'bot-cls
                           :tc1 4 :tc2 2 :mc1 0 :mc2 2 :sc1 1 :sc2 1))
(def-test b-f-m :methods (make-instance 'bot-cls
                           :tc1 0 :tc2 2 :mc1 4 :mc2 2 :sc1 1 :sc2 1))
(def-test b-f-s :methods (make-instance 'bot-cls
                           :tc1 0 :tc2 2 :mc1 0 :mc2 2 :sc1 1 :sc2 3))
(def-test b-f-mt :methods (make-instance 'bot-cls
                            :tc1 4 :tc2 2 :mc1 4 :mc2 2 :sc1 1 :sc2 1))
(def-test b-f-ms :methods (make-instance 'bot-cls
                            :tc1 0 :tc2 2 :mc1 4 :mc2 2 :sc1 1 :sc2 3))
(def-test b-f-ts :methods (make-instance 'bot-cls
                            :tc1 4 :tc2 2 :mc1 0 :mc2 2 :sc1 1 :sc2 3))
(def-test b-f-mts :methods (make-instance 'bot-cls
                             :tc1 4 :tc2 2 :mc1 4 :mc2 2 :sc1 1 :sc2 3)))
```

C.3 Method combinations

NST defines a method combination `nst-results` as the default method combination for functions defined by `def-test-generic`. This combination runs *all* applicable methods, and combines all of their results into a single NST result record.

This default can be overridden by specifying `t` as the method combination in the intial declaration.

```
(nst:def-test-generic overridden
    (:method-combination t))
(nst:def-test-method-criterion overridden mid-cls
  (:slots (mc1 (:eql 0))
          (mc2 (:eql 2))))
(nst:def-test-method-criterion overridden bot-cls
  (:slots (sc1 (:eql 1))
          (sc2 (:eql 1))))
```

D Deprecated forms

The macros, functions and variables documented in this section are all deprecated. Some continue to be exported from the NST API; others have already

been removed. This section describes how code using these forms should be ported to the active NST API.

D.1 Older criteria-defining macros

The `def-criterion-unevaluated` macro is deprecated as of NST 2.1.2. It was consolidated into the `def-criterion` macro.

Replace:

```
(def-criterion-unevaluated name (pattern ... pattern) name
  BODY)
```

with:

```
(def-criterion name (:forms pattern ... pattern)
                    (:form name)
  BODY)
```

The `def-values-criterion` macro was deprecated as of NST 1.3.0. For new criteria, use `def-criterion` instead. In the short term, code using `def-values-criterion` should continue to work as before.

The `def-form-criterion` macro was deprecated as of NST 1.3.0. *Code using* ***`def-form-criterion`*** *in any but the simplest ways is very likely to fail.* Use `def-criterion` instead.

D.2 Old test result generators

The `emit-failure` function is deprecated; use `make-failure-report` instead.

The `emit-success` function is deprecated; use `make-success-report` instead.

The `emit-warning` function is deprecated; use `make-warning-report` instead.

Index