Getting started with NST

The key elements to getting started with NST are:
— The syntactic forms for specifying tests (below).
— The criteria used in tests (p. 5).
— Using NST for interactive debugging (p. 9).
— Combining NST with ASDF to help automate testing (p. 9).

Nomenclature

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. Fixtures are defined using the def-fixture form.

Groups of tests can be associated with fixture sets, stateful initialization, and stateful cleanup. Test groups are defined using the def-test-group form.

Individual tests are declared within test groups using the def-test form.

Examples of NST fixtures, groups and tests are available in self-test/core/simple-mnst.lisp.

The figure on the next page shows the changes involved in using NST in a system definition.

Forms

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:

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

1This document was last updated for NST version 3.0.1. Written by John Maraist, Smart Information Flow Technologies; 211 N. First St. Suite 300, Minneapolis, MN 55401; jmarais at sift.net.
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 suppress 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 every time 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:

```lisp
(defun fixtures f1 ()
  (c 3)
  (d 'asdfg))
_RDONLY fixtures f2 (:special ((:fixture f1)))
  (d 4)
  (e 'asdfg)
  (f c))
_RDONLY 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 nils are omitted. 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 initialization, and stateful cleanup.

```lisp
(defun 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.

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
```

```
(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 FORMs. 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. Like-
wise, any criterion consisting of a single symbol, e.g. (:pass), may be abbrevi-
ated 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. The startup, :setup, :cleanup, :finish and :fixture options are as above, but apply to only this test (and note that for multiple forms for the first two must be wrapped in a prog). NST’s built-in criteria are listed below; see the manual for a discussion of the forms which define new criteria.

Criteria

Criteria forms have the following structure:

(CRITERIA-NAME ARG ARG ... ARG)

but a no-argument criterion use (NAME) can be abbreviated as NAME. NST’s built-in criteria include:

Basic criteria.

:true The form is evaluated at testing time; the criterion requires the result to be non-nil.
Syntax: :true
Applicable to: A single form.

eq The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be eq.
Syntax: (:eq FORM)
Applicable to: A single form.

:symbol The form under test is evaluated at testing time. The criterion requires that the result be a symbol which is eq to the symbol name given as the criterion argument.
Syntax: (:symbol NAME)
Applicable to: A single form.

eql The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be eql.
Syntax: (:eql FORM)
Applicable to: A single form.

:equal The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be equal.
Syntax: (:equal FORM)
Applicable to: A single form.
:equalp The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be equalp.

Syntax: (:equalp FORM)
Applicable to: A single form.

/forms-eq The two forms under test are both evaluated at testing time; the criterion requires that the results be eq.

Syntax: :forms-eq
Applicable to: Exactly two forms.

/forms-eql The two forms under test are both evaluated at testing time; the criterion requires that the results be eql.

Syntax: :forms-eql
Applicable to: Exactly two forms.

/forms-equal The two forms under test are both evaluated at testing time; the criterion requires that the results be equal.

Syntax: :forms-equal
Applicable to: Exactly two forms.

/predicate 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.

Syntax: (:predicate FUNCTION-FORM)
Applicable to: Forms matching the input lambda list of the FUNCTION-FORM.

/err At testing time, 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.

Syntax: (:err [:type CLASS])
Applicable to: Any.

/perf Evaluates the forms under test at testing time, and expects the evaluation to complete within the given time limit.

Applicable to: Any.

Compound criteria.

/not Passes when testing according to CRITERION fails (but does not throw an error).

Syntax: (:not CRITERION)
Applicable to: As required by the subordinate criterion.
:all  This criterion brings several other criteria under one check, and verifies that
they all pass.
Syntax: (:all CRITERION CRITERION ... CRITERION)
Applicable to: As required by the subordinate criteria.

:any  Passes when any of the subordinate criteria pass.
Syntax: (:any CRITERION CRITERION ... CRITERION)
Applicable to: As required by the subordinate criteria.

:apply  At testing time, 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.
Syntax: (:apply FUNCTION CRITERION)
Applicable to: Forms matching the input lambda list of the FUNCTION.

:check-err  Like :err, but proceeds according to the subordinate criterion rather
than simply evaluating the input forms.
Syntax: (:check-err CRITERION)
Applicable to: As required by the subordinate criterion.

:progn  At testing time, first evaluates the FORMs in order, and then proceeds
with evaluation of the forms under test according to the subordinate criterion.
Syntax: (:progn FORM FORM ... FORM CRITERION)
Applicable to: As required by the subordinate criterion.

:proj  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.
Syntax: (:proj (INDEX INDEX ... INDEX) CRITERION)
Applicable to: At least as many as to be accessible to the largest index.

Criteria for multiple values.

:value-list  Converts multiple values into a single list value.
Syntax: (:value-list CRITERION)
Applicable to: Arbitrarily many values.

:values  Checks each of the forms under test according to the respective subor-
dinate criterion.
Syntax: (:values CRITERION CRITERION ... CRITERION)
Applicable to: Exactly as many forms as subordinate criteria.

:drop-values  Checks the primary value according to the subordinate criterion, ignoring any additional returned values from the evaluation of the form under test.
Syntax: (:drop-values CRITERION)
Applicable to: Any.
Criteria for lists.

:each At testing time, 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.

Syntax: (:each CRITERION)
Applicable to: A single form which evaluates to a list.

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

Syntax: (:seq CRITERION CRITERION ... CRITERION)
Applicable to: A single form which evaluates to a list with the same number of elements as there are subordinate criteria to the :seq.

:permute At testing time, 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.

Syntax: (:permute CRITERION)
Applicable to: A single form evaluating to a list.

Criteria for vectors.

:across Like :seq, but for a vector instead of a list.

Syntax: (:across CRITERION CRITERION ... CRITERION)
Applicable to: A single form evaluating to a vector.

Criteria for classes.

:slots Evaluates its input form, and passes when the value at each given slot satisfies the corresponding subordinate constraint.

Syntax: (:slots (NAME CRT) (NAME CRT) ... (NAME CRT))
Applicable to: A single form evaluating to a class or struct object.

Special criteria.

:sample Experimentally test a program property by generating random data. See the users’ manual for more information.

Syntax: (:sample &key domains where verify values sample-size qualifying-sample max-tries)
Applicable to: Forms must match lambda list values.

Programmatic and debugging criteria.
:info Add an informational note to the check result.

Syntax: (:info MESSAGE SUBCRITERION)
Applicable to: Any.

:pass A trivial test, which always passes.

Syntax: :pass
Applicable to: Any.

:fail A trivial test, which always fails. The format string and arguments should be suitable for the Lisp format function.

Syntax: (:fail FORMAT ARG ... ARG)
Applicable to: Any.

:warn Issue a warning. The format string and arguments should be suitable for the Lisp format function.

Syntax: (:warn FORMAT ARG ... ARG)
Applicable to: Any.

:dump-forms For debugging NST criteria: fails after writes the current forms to standard output.

Syntax: (:dump-forms FORMAT)
Applicable to: Arbitrarily many values, compatible with the given string for the Lisp format function.

Interactive debugging

NST defines a REPL alias :nst under Allegro CL. The general form of commands is:

:nst COMMAND ARGUMENTS

Use :nst :help for a list of commands, and :nst COMMAND :help for details about individual commands.

ASDF

NST tests can be referenced from ASDF systems, allowing easy invocation to unit tests relevant to a system. The system self-test/masdfnst.asd gives an example of its use. The snippet below highlights the difference between non-NST and NST-oriented ASDF system declarations.

;; Force loading NST’s ASDF utilities before processing
(defpackage :masdfnst-asd
  (:use :common-lisp :asdf :asdf-nst))
(in-package :masdfnst-asd)

(defsystem :masdfnst
  ;; Use the NST-oriented ASDF system definition.
  :class nst-testable
  :in-order-to ((test-op (load-op :masdfnst)))
  ;; Any one of the six blocks below is reasonable.
  ;; Use exactly one of :nst-package, :nst-group, or
  ;; :nst-test; or any combination of the plural
  ;; versions.
  ;; (1)
  ;; :nst-package :asdf-nst-test
  ;; (2)
  ;; :nst-group (:asdf-nst-test . core-checks)
  ;; (3)
  ;; :nst-test (:asdf-nst-test core-checks pass-1)
  ;; (4)
  ;; :nst-packages (:asdf-nst-test :asdf-nst-test2)
  ;; (5)
  ;; :nst-packages (:asdf-nst-test)
  ;; :nst-groups ((:asdf-nst-test2 . :g1a))
  ;; (6)
  :nst-groups ((:asdf-nst-test2 . :g1))
  :nst-tests ((:asdf-nst-test2 :g1a :fix0)
               (:asdf-nst-test :core-checks :warn-1))
  :components { ... })}