# Reference Manual for TESS Version 0.1.2

Michael S. Noble, mnoble@space.mit.edu Nov 2, 2005

# Preface

TESS is the (Te)st (S)ystem for (S)-Lang.

Copyright (C) 2004-2005  Massachusetts Institute of Technology
Michael S. Noble <mnoble@space.mit.edu>

This software was partially developed at the MIT Center for Space
Research, under contract SV1-61010 from the Smithsonian Institution.

Permission to use, copy, modify, distribute, and sell this software
and its documentation for any purpose is hereby granted without fee,
provided that the above copyright notice appear in all copies and
that both that copyright notice and this permission notice appear in
the supporting documentation, and that the name of the Massachusetts
Institute of Technology not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.  The Massachusetts Institute of Technology makes
no representations about the suitability of this software for any
purpose.  It is provided "as is" without express or implied warranty.

THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS.  IN NO EVENT SHALL THE MASSACHUSETTS
INSTITUTE OF TECHNOLOGY BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

# Contents

# Chapter 1

# Introduction

TESS is the (Te)st (S)ystem for (S)-Lang, which aims at reducing the workload and *ad-hoc* nature of regression testing S-Lang software ( http://www.s-lang.org ), by collecting common testing elements into a single, easy-to-use framework. TESS provides the S-Lang developer nominal mechanisms for tailoring the S-Lang environment and invoking functions with arbitrary inputs, while transparently inspecting and cleaning the stack, gathering pass/fail statistics, and providing error recovery from selected exceptions.

TESS is primarily a development tool, so we assume the reader is familiar with scripting in S-Lang. Knowledge of how to create S-Lang modules is also helpful. Since TESS is intended to assist in automating and simplifying regression testing, it is most often utilized in conjunction with the `make` tool via a `Makefile`. If these are unfamiliar terms then you will benefit from learning about them first prior to reading further.

## 1.1 Motivation

Suppose you had a function defined within some file `add.sl` as

```
define add()
{
   variable i, j;
   if (_NARGS < 2) usage("add(i,j)");
   (i,j) = ();
   return i+j;
}
```

and that you wanted to exercise it in automated fashion on a range of input conditions. To test it with insufficient arguments, for instance, one might write a script `add.t`

```
() = evalfile("./add");
add(1);
```

which, when invoked from `slsh` as

```
slsh ./add.t
```

yields something like

```
Usage: add(i,j)
called from line 2, file: ./add.t
```

### 1.1.1 Model: Multiple Tests per Script

Ok, so far things look reasonable. Now, suppose you appended the line

```
add(1,"two");
```

to the test and reran it in `slsh`. Curiously enough, the output generated would look no different from above. Why? Because the first `add()` call generates a usage exception, which causes the interpreter to unwind the S-Lang stack and then exit. In other words, the second `add()` call *is never even executed!* One way to address this is to modify the script by adding an `ERROR_BLOCK`, such as:

```
() = evalfile("./add");

define test1()
{
   ERROR_BLOCK { _clear_error(); return; }
   add(1);
}

test1;
add(1, "two");
```

Now both tests will be exercised when the script is invoked, generating a usage exception in the first case and a type mismatch error in the second. Progress, for sure, but the script has grown longer, and we needed to introduce a wrapper for the first test case in order to use the error block. In this model, where one script contains multiple test cases, each test would need to be invoked within such a wrapper, which explains why TESS offers the `tess_invoke()` function.

### 1.1.2 Model: Single Test per Script

An alternative to the model used above would be to each test case within its own `.t` file and invoke `slsh` upon each. This approach avoids the need for error blocks, but introduces a number of other concerns which collectively steer the author towards a preference for the first model.

For example, the resultant file proliferation makes it more cumbersome to enumerate/name and effectively organize the test suite. Packages of only moderate size might conceivably contain scores or perhaps hundreds of relatively tiny files, needlessly wasting disk resources, increasing sizes of software distributions, and wasting CPU cycles by launching `slsh` *once per test* instead of only *once per group of tests*. Moreover, each test invocation would also need to load both TESS and the package being tested, resulting in yet more CPU waste and code duplication.

Identifying semantically related test cases would not be as easy, since groupings are now distinguished by like-named files within a directory, instead of by cohabitation of test cases within a single file.

Collecting useful failure statistics becomes more difficult, since in this model aggregate counts can be be obtained only by metascripts, e.g. invoked within Makefiles at the operating system level to keep track of the 1 or 0 returned from each test, instead of within the test scripts themselves. In contrast, the `tess_invoke` function mentioned above transparently tallies pass/fail statistics.

For small test suites these issues may be negligible, but as packages grow their cumulative effect may not be so easily ignored, making it better to cultivate the preferable habit of "starting clean," rather than one of "cleaning up later".

### 1.1.3 Consistency and Reuse

TESS emerged as the coalescence of scripts used in testing a number of existing S-Lang packages. In fact, development versions of packages such as SLgtk and SLxpa have already been migrated from their original testing scheme towards TESS, and as such they serve as the wealthiest source of supplemental examples to this documentation.

By distilling common patterns from existing test suites into a self-contained distribution TESS fosters reuse, reduces duplicative busy work, and hopefully serves to ease the burden of testing (perhaps one of the least loved aspects of writing and maintaining software).

# Chapter 2

# Examples

Before continuing it is worthwhile noting that the examples in this chapter are bundled within the `examples` directory of the TESS distribution (along with a Makefile and baseline *test.ref* output), and that they have been written under the assumption that an operational version of TESS is installed on your system.

It should also be noted that a TESS script is merely a S-Lang script which has, somewhere along its execution path, loaded TESS. Normally S-Lang scripts are named with a `.sl` suffix, however the author has adopted the convention of suffixing them with `.t` instead. Not only does this serve as a useful mnemonic, it also prevents tests from being accidently loaded by S-Lang through its default script loading mechanism (fostering the use of nearly identical names for both test scripts and the package components which they exercise).

## 2.1 Scripts

Let's begin our example by augmenting the `add()` function defined above with a corresponding `subtract()` function, and migrating the definition of both into a S-Lang script named `sillymath.sl`, as follows:

```
define add()
{
   variable i, j;
   if (_NARGS < 2) usage("add(i,j)");
   (i,j) = ();
   return i+j;
}

define subtract()
{
   variable i, j;
   if (_NARGS < 2) usage("subtract(i,j)");
   (i,j) = ();
   return i - j;
}
```

Two TESS scripts which more thoroughly exercise `sillymath.sl`, then, are:

`add.t:`

```
require("tess");

tess_invoke(1, &add);
tess_invoke(1, &add, "hi there!");
tess_invoke(1, &add, 2);
tess_invoke(0, &add, 2, 3);

tess_invoke(1, &add, "one", 2);
tess_invoke(0, &add, "hello", " there!");
```

`subtract.t:`

```
require("tess");

variable f = &subtract;
tess_invoke(1, f);
tess_invoke(1, f, "hi there!");
tess_invoke(1, f, 2);
tess_invoke(0, f, 2, 3);

tess_invoke(1, f, "one", 2);
tess_invoke(1, f, "hello", " there!");
```

## 2.2 Output

The first of these should emit feedback resembling that given below. Since the result of `add.t` closely resembles that of `subtract.t` (differing only in that case 5 signals an exception in one but not the other, since string subtraction is undefined while string addition is equivalent to concatenation) we omit output from the latter.

```
Usage: add(i,j)

Test Case 1: add: PASSED (SHOULD signal error, DID)

Usage: add(i,j)

Test Case 2: add: PASSED (SHOULD signal error, DID)

Stack Contents:
(0)[String_Type]:hi there!

Usage: add(i,j)

Test Case 3: add: PASSED (SHOULD signal error, DID)
```

```
Stack Contents:
(0)[Integer_Type]:2


Test Case 4: add: PASSED (SHOULD NOT signal error, DID NOT)

Stack Contents:
(0)[Integer_Type]:5

S-Lang Error: Type Mismatch: String_Type + Integer_Type is not possible

Test Case 5: add: PASSED (SHOULD signal error, DID)


Test Case 6: add: PASSED (SHOULD NOT signal error, DID NOT)

Stack Contents:
(0)[String_Type]:hello there!


=============== add Test Summary ===============

            Number of Failures: 0
            Number of Passes  : 6
```

## 2.3 Analysis

In the above tests the absence of an explicit `ERROR_BLOCK`, and the cleaner code which results, should be immediately apparent. For example, `add.t` contains one fewer non-blank lines than does the example in section 1.1.1 (Motivation), while tripling the number of cases tested (6 versus 2). Also revealed is the fact that `tess_invoke()` is the single most important function in the interface.

### 2.3.1 `tess-common.sl`

A more subtle point of interest is that `sillymath.sl` does not appear to be loaded by either test script. How, then, do they function? The answer is that the `evalfile()` has been pushed into a file `tess-common.sl`, which TESS will transparently load if found within the current directory at startup.

This exploits a common pattern within test suites, namely that prior to testing *any* component within a package a test script must first load the package itself. Furthermore, `tess-common.sl` can be used to define variables, data structures, or functions that might be commonly used amongst all test scripts within a given suite.

### 2.3.2 Type Mismatch Exceptions

Consider the case

```
tess_invoke(1, &add, "one", 2);
```

defined for `add()` (and the similar test defined for `subtract()`), which attempts to add an `Integer_Type` to a `String_Type` and results in a type mismatch exception. Even with an `ERROR_BLOCK` defined S-Lang 1.x would not, on its own, catch this exception (although the forthcoming S-Lang 2.x will). This is because S-Lang regards type mismatches as fatal errors, so the interpreter will not permit execution to continue after they occur.

Because it can be useful to test such conditions, however, TESS relaxes this constraint by installing an error handler which resets S-Lang internal state and allows scripts to continue processing. TESS installs this handler by default at startup, but as noted in the function reference it may be deactivated (or reactivated) by calling `tess_catch_type_errors()`.

### 2.3.3 Stack Maintenance

After each test TESS also reports the number and type (or value, for objects which are not aggregates) of any items remaining on the S-Lang stack. Consider the line

```
Stack Contents:
(0)[String_Type]:hello there!
```

present in the output of `add.t` above. It shows that after test case 6 has completed the S-Lang stack contains 1 item (as it should), namely the result of concatenating the strings "hello" and " there!".

This provides an excellent way of validating the return values of exercised functions, without requiring any additional work on the part of the test developer. Conversely, this feature also identifies functions which create unintended side effects by leaving detritus on the stack.

### 2.3.4 Results Summary

Another point of interest from the `add.t` output is that the pass/fail statistics of each test script are automatically summarized, again with zero work required on the part of the test developer. This happens because TESS installs an *exit handler* which, by default, transparently calls `tess_summary` when the test application terminates. As noted in the function reference, this behavior may be customized.

### 2.3.5 Exit Status

If the test application offers an `exit` intrinsic then TESS will invoke it at completion time, passing in the number of failures observed while running the script. This enables higher-level Makefiles, or `tessrun`, to take appropriate action, such as terminating when a non-zero status is returned.

# Chapter 3

# Function Reference

## 3.1 tess_add_eval_paths

**Synopsis**

Add one or more directories to the S-Lang evalfile() search path

**Usage**

```
tess_add_eval_paths( path1, [path2, ...])
```

**Description**

This function is a convenience wrapper around the set_slang_load_path() function, making it cleaner and simpler to augment the list of directories searched by the S-Lang interpreter when evalfile() is invoked with an ambiguous file specification.

**Notes**

TESS automatically appends the current working directory, as well as ../src, ../share, and ../packages to the load path.

**See Also**

`tess_add_import_paths`

## 3.2 tess_add_import_paths

**Synopsis**

Add one or more directories to the S-Lang import() search path

**Usage**

```
tess_add_import_paths( path1, [path2, ...])
```

**Description**

This function is a convenience wrapper around the set_import_module_path() function, making it cleaner and simpler to augment the list of directories searched by the S-Lang interpreter when import() is invoked.

**Notes**

TESS automatically appends ../src to the import path.

**See Also**

`tess_add_eval_paths`

## 3.3 tess_catch_type_errors

**Synopsis**

Give S-Lang ERROR block mechanism the ability to catch type mismatch errors

**Usage**

```
tess_catch_type_errors( [yes_or_no] )
```

**Description**

This function augments the S-Lang ERROR block mechanism, giving it the ability to catch type mismatch exceptions (which S-Lang 1.x formally considers uncatchably fatal). This feature is useful for a test framework, since it allows functions to be safely exercised against a wide variety of types.

If the first passed argument evaluates to a boolean TRUE then the function will enable type error catching. If either zero arguments are passed, or the first argument evaluates to boolean FALSE, then type error catching will be disabled.

**Notes**

In S-Lang 2 all exceptions may be caught, which makes this function moot.

**See Also**

`tess_invoke`

## 3.4 tess_invoke

**Synopsis**

Execute a test case

**Usage**

```
tess_invoke( expected_to_fail, function_ref [, arg1, arg2, ...])
```

**Description**

Invoke the given function (by dereference), optionally passing in one or more arguments. The first parameter, whose value should be either zero or one, indicates whether the function is expected to signal an error when invoked in the manner given.

If the actual result of the call matches the expected result then the test case is said to "pass," otherwise it is said to "fail". It is important to understand this: a failed test case is not indicated by an error signal itself, but rather by whether or not the test case expected an error to be signaled.

### Notes

### See Also

`tess_catch_type_errors, tess_summary`

## 3.5 tess_load_component

### Synopsis

Evaluate the named S-Lang script, and set the test component name accordingly

### Usage

```
tess_load_component(filename)
```

### Description

This function attempts to evalfile() the named script, using the usual S-Lang load mechansism, and will set the TESS test component name to the filename if found.

The test component name is printed in the heading of results summaries, and uniquely identifies a given test script. Typically the test component name is set to the "basename" of the test script itself (e.g. a script add.t sets Component = "add"). This function provides a means of customizing that default behavior while loading additional functionality to be exercised within the test script.

### Notes

### See Also

## 3.6 tess_summary

### Synopsis

Summarize the results of a suite of tests

### Usage

```
Integer_Type tess_summary()
```

### Description

TESS automatically records the pass/fail result of each test case executed by tess_invoke. By default the results of this tally are emitted to stdout when tess_summary is called, although this may be disabled by setting the _tess_auto_summarize intrinsic variable to 0. The return value indicates the number of failed tests.

### Notes

Under normal circumstances it should not be necessary to call this function explicitly, since TESS transparently installs an exit handler which calls tess_summary at application termination. Its return value is then passed to the operating so that, for example, a non-zero status may be used to fatally terminate a "make test" goal.

**See Also**

`tess_invoke`

# Chapter 4

# Utility Script Reference

## 4.1 tessrun

The `tessrun` script is intended to simplify the invocation of a suite of regression tests, and has a command line usage synopsis of:

```
tessrun [-v] [application_name]
```

Specifying the `-v` (verbose) flag inhibits the redirection of stderr to /dev/null, while specifying an `application_name` overrides the default use of `slsh` as the test application.

The chief benefits of `tessrun` are that it yields simpler Makefiles which require fewer modifications. For example, rather than having the *make test* rule within your regression test Makefile explicitly loop over all *.t* files within its directory, the rule can simply invoke `tessrun`, perhaps as:

```
test:
        tessrun $(TESS_APP) > test.out 2>&1
        diff test.ref test.out
```

Since zero tests are explicitly referenced by name the Makefile need not change as new tests are added to the regression suite. Nor need need it be modified simply to execute the test suite within other applications (again, the default test application is `slsh`), since that can be achieved here simply by overriding the definition of $(TEST_APP) during the invocation of *make*:

```
unix%  make -e TEST_APP=isis
```

The only constraint upon the test application is that it offers the S-Lang *provide* and *require* package management functions.