Lightweight assertion library for testing

Jérôme Laurens

v0.1 2021/09/02

Abstract

Provides a pair of commands to insert assertions in the code. It is mainly intended for package developers but document writer may find this helpful.

Contents

1 Minimal example

The document below compiles without any error and the console output reads "`Test Assert Info:  All 9 tests passed`":

```
\documentclass {article}
\RequirePackage {assert}
\begin {document}
\newcommand \ONE {1}
\Assert {IntEqual} {\ONE+1} {2}
\Assert {IntNotEqual} {\ONE+1} {3}
\Assert {StrEqual} {\ONE+1} {1+1}
\Assert {StrNotEqual} {\ONE+1} {2}
\Assert {StrMatch} {1.1} {\ONE+1}
\Assert {FPNotEqual} {1} {1.0000001}
\Assert {AlmostEqual} {1} {1.0000001}
\Assert {AlmostNotEqual} {1} {1.01}
\AssertSetPrecision {0.01}
\Assert {AlmostEqual} {1} {1.01}
\end {document}
```

Each of the next lines causes a fatal error, the last one due to an extra space.

```
\Assert {IntNotEqual} {\ONE+1} {2}
\Assert {IntEqual} {\ONE+1} {3}
\Assert {StrEqual} {\ONE+1} {1 +1}
```

2 Testing

2.1 Presentation

Testing is a major phase in modern software development. It ensures that the final product fulfills some expectations for the satisfaction of the customers. In general, many tools are available to help automating the testing process depending on the context and this particular package is a testing tool targeting LaTeX code.

A test basically consists in comparing an ⟨*actual*⟩ value to an ⟨*expected*⟩ one. If they conform to each other, the test passed, otherwise the test failed. The goal is to write tests concerning the different features of the code currently in development. When running the tests, none should fail.

There are different kinds of tests for different purposes. Actually, the built in command line tool `l3build` combined with the support of macros in `regression-test.tex`, allows to run a wide range of regression tests. A regression test aims at ensuring that a change made does not break the code. It can be performed only once some working code is already available. In practice with `l3build`, regression tests need many steps:

- write some LaTeX code,
- write ⟨*test name*⟩`.lvt` files to test that code,
- run `l3build save` ⟨*test name*⟩ to produce ⟨*test name*⟩`.lgt` that contains what is expected,
- make any improvements to the initial LaTeX code,

- run `l3build check` ⟨*test name*⟩ to see if the ⟨*test name*⟩`.lgt` produced conforms to what is expected.

This system is very powerful on some respect but it is not really suitable for active development of simple features, which is the starting point for everything.

Here the unique `\Assert` command is a straighformward comparison between what is expected and what is actually available. It will raise a fatal error at the end of the document if they do not conform to each other. The reason of the failure, if any, is detailed in the log. With `\Assert`, we cannot test for boxes of pdf nodes, but we can test dynamic expected values.

qstest is another alternative to writing tests.

2.2 Testing with assertions

`\Assert`

```
\Assert {⟨type⟩}
  [⟨actual expansion rule⟩] {⟨actual⟩}
  [⟨expected expansion rule⟩] {⟨expected⟩}
```

⟨*type*⟩ must be one of the previously added types otherwise a fatal error is raised. See section 2.6. Available hooks are invoked in that order

– `cmd/Assert/before`

– `cmd/Assert/`⟨*type*⟩`/before`

only on failure:

  – `cmd/Assert/before failure`

  – `cmd/Assert/`⟨*type*⟩`/failure`

  – `cmd/Assert/after failure`

– `cmd/Assert/`⟨*type*⟩`/after`

– `cmd/Assert/after`

The ⟨*actual*⟩ and ⟨*expected*⟩ argument names are formal, they just refer to common pratice in testing and you can change their order of use.

For token list assertions, the ⟨*actual*⟩ and ⟨*expected*⟩ arguments may follow an optional ⟨*expansion rule*⟩ argument, as detailed in *interface3.pdf*.

x for *exhaustive expansion*: expand every token of ⟨*actual*⟩ or ⟨*expected*⟩, then every token of the expansion and so on until only unexpandable tokens remain,

e for an alternate *exhaustive expansion*: that may be expandable, contrary to the former,

o for *expansion once*,

v for *value of variable*: a command is constructed first from the given name, then its value is recovered.

For assertion data other then token lists, the ⟨*actual*⟩ and ⟨*expected*⟩ arguments are exhaustively expanded such that no previous ⟨*expansion rule*⟩ is needed.

2.3 Known assertion types

In the sequel, names are self explanatory.

2.3.1 Token lists and strings

TLEqual token list equality,

TLNotEqual token list inequality,

StrEqual string equality,

StrNotEqual string inequality,

2.3.2 Integers

Both arguments may be number expression as defined by *interface3d.pdf*. Before the tests, both ⟨*actual*⟩ and ⟨*expected*⟩ are replaced by their floor integer value.

IntLess integer ordering,

IntNotLess integer ordering,

IntGreater integer ordering,

IntNotGreater integer ordering,

2.3.3 Floats

Both arguments may be number expression as defined by *interface3d.pdf*.

FPEqual float equality,

FPNotEqual float inequality,

FPLess float ordering,

FPNotLess float ordering,

FPGreater float ordering,

FPNotGreater float ordering,

To test if a ⟨*number*⟩ is an integer, consider it as a float and compare it to its floor integer value: `\Assert {FPEqual} {`⟨*number*⟩`} {floor(`⟨*number*⟩`)}`. Use `FPNotEqual` instead to test a ⟨*number*⟩ that should not be an integer.

2.3.4 Matching

Matching allows to test only parts of a token list or string. It follows a different rule: ⟨*actual*⟩ is a regular expression (as defined by l3regex package in *interface3d.pdf*), whereas ⟨*expected*⟩ is either a token list or a string. The latter is exhaustively expanded.

TLMatch token list match,

TLNoMatch token list negative match,

StrMatch string match,

StrNoMatch string negative match.

2.3.5 Almost

Computational errors are managed here, for example the equality $a = b$ is replaced by $|a - b| \leq \varepsilon(1 + |a| + |b|)$ where $\varepsilon > 0$ is the precision. The inequality $a < b$ is replaced by $a - b \leq \varepsilon(1 + |a| + |b|)$ and so on.

AlmostEqual float equality,

AlmostNotEqual float inequality,

AlmostLess float ordering,

AlmostNotLess float ordering,

AlmostGreater float ordering,

AlmostNotGreater float ordering,

The precision can be managed separately.

`\AssertPrecision` Internal precision holder, can be used in float expressions.

`\AssertUsePrecision` ⋆ Precision getter for typesetting.

`\AssertSetPrecision` `\AssertSetPrecision {⟨new precision⟩}`

Precision setter.

2.4 Sharing information

Sharing information between hook codes is possible with a ⟨*key*⟩–⟨*value*⟩ property list. We only provide a high level management interface with a getter and a setter.

2.4.1 Property ⟨*keys*⟩

⟨*key*⟩ can be one of the reserved strings

`/type`

`/actual`

`/expected`

`/operator`

For each key but the last, the corresponding ⟨*value*⟩ is the eponym argument of the `\Assert` command. The operator is the eponym argument used when adding the type with `\AssertAdd`. Changing these values while tpesetting is not supported.

For any other key, the result is what was added to the property list by the client.

2.4.2 Management

`\AssertIn` ⋆

`\AssertIn {⟨key⟩} {⟨in code⟩} [⟨out code⟩]`

Query for properties by ⟨*key*⟩. If the ⟨*key*⟩ is in the property list shared by hooks, then ⟨*in code*⟩ is executed, otherwise the optional ⟨*out code*⟩ is executed.

`\AssertGet` ⋆

`\AssertGet {⟨key⟩}`

Return the ⟨*value*⟩ corresponding to the given ⟨*key*⟩. It can be nothing such that `\AssertIn` must be used sometimes.

`\AssertSet`

`\AssertSet {⟨key⟩} {⟨value⟩}`

Set a new value for the given key. Overriding the ⟨*value*⟩ for a reserved ⟨*key*⟩ is unsupported despite it may kind of work.

2.5 Shared arguments

Some arguments are shared by different functions.

⟨*type*⟩ unique identifier of an assertion. Chosen after the test performed: `StrEqual`, `StrNotEqual`...

2.6 Defining assertions

`\AssertAdd`

`\AssertAdd {⟨type⟩} {⟨comparator⟩} {⟨operator⟩}`

⟨*type*⟩ see 2.5,

⟨*comparator*⟩ is the name of an `..._if_eq:...` like conditional function with signature `nnT` or `nnF` and no return value, the code should be executed on failure,

⟨*operator*⟩ is one of `==`, `!=`... It is the relation expected by the test between its two first arguments. Used to display information.

If the ⟨*type*⟩ was already added, then an error is raised. The function registers the new key and declares associated hooks

– `cmd/Assert/`⟨*type*⟩`/before`,

– `cmd/Assert/`⟨*type*⟩`/failure`,

– `cmd/Assert/`⟨*type*⟩`/after`.

3 Implementation

The implementation is private and should not be relied on. Only the public interface documented above is supported.

3.1 DocStrip guards

```
⟨*pkg⟩
```

Identify the internal prefix (LaTeX3 DocStrip convention).

```
⟨@@=assert⟩
```

3.2 Package declarations

```
\NeedsTeXFormat{LaTeX2e}[2020/01/01]
\ProvidesExplPackage
  {assert}
  {2021/09/02}
  {0.1}
  {Lightweight assertion library for testing}
```

3.3 Variables

To count the number of tests performed:

```
\int_zero_new:N \g__assert_int
```

To record test informations on failure in order to display them all at once in the log at the end of the document:

```
\seq_new:N \g__assert_seq
```

To store assertion types and characteristics. Each ⟨*key*⟩ is a type name. Each ⟨*value*⟩ is a token list with 2 groups: a ⟨*comparator*⟩ and a descriptive ⟨*operator*⟩.

```
\prop_new:N \g__assert_type_prop
```

To store arguments to be shared by hooks:

```
\prop_new:N \g__assert_prop
```

Placeholders to retrieve property values and more.

```
\tl_new:N \l__assert_tmpa_tl
\tl_new:N \l__assert_tmpb_tl
```

Common hooks

```
  \NewHook {cmd/Assert/before failure}
  \NewHook {cmd/Assert/after failure}
```

3.4 Add an assertion type

```
\cs_generate_variant:Nn \cs_generate_variant:Nn { cx }
\NewDocumentCommand\AssertAdd {mmm} {
  \prop_put:Nnn \g__assert_type_prop {#1} {
    { \exp_not:n { #2 } }
    { \exp_not:n { #3 } }
  }
  \NewHook {cmd/Assert/#1/before}
  \NewHook {cmd/Assert/#1/failure}
  \NewHook {cmd/Assert/#1/after}
  \tl_map_inline:nn {novex} {
    \tl_map_inline:nn {novex} {
      \cs_generate_variant:cx { #2 } { ##1 ####1 }
    }
  }
}
```

3.5 Sharing information

```
\prop_new:c { dict:assert/shared }
```

```
\cs_generate_variant:Nn \prop_if_in:NnTF { ce }
\NewExpandableDocumentCommand \AssertIn { mmm } {
  \prop_if_in:ceTF { dict:assert/shared } { #2 } { #3 }
  \ignorespaces
}
```

```
\tl_map_inline:nn { novex } {
  \cs_generate_variant:Nn \prop_put:Nnn { cx#1 }
}
\NewDocumentCommand \AssertSet { mO{n}m } {
  \cs:w prop_put:cx#2 \cs_end: { dict:assert/shared } { #1 } { #3 }
  \ignorespaces
}
```

```
\cs_generate_variant:Nn \prop_item:Nn { ce }
\NewExpandableDocumentCommand \AssertGet { m } {
  \prop_item:ce { dict:assert/shared } { #1 }
  \ignorespaces
}
```

3.6 Assertion types

3.6.1 Token lists

```
\AssertAdd { TLEqual     } { tl_if_eq:nnF } { == }
\AssertAdd { TLNotEqual  } { tl_if_eq:nnT } { != }
```

```
\AssertAdd { TLMatch } { regex_match:nnF } { ~ }
\AssertAdd { TLNoMatch } { regex_match:nnT } { ~ }
```

3.6.2 Strings

```
\cs_new:Npn \__assert_str_if_eq:nnF #1#2#3 {
  \exp_args:Nxx \str_if_eq:nnF {#1} {#2} { #3 }
}
\AssertAdd { StrEqual    } { __assert_str_if_eq:nnF } { == }
```

```
\cs_new:Npn \__assert_str_if_eq:nnT #1#2#3 {
  \exp_args:Nxx \str_if_eq:nnT {#1} {#2} { #3 }
}
\AssertAdd { StrNotEqual } { __assert_str_if_eq:nnT } { != }
```

```
\cs_new:Npn \__assert_str_match:nnF #1#2#3 {
  \exp_args:Nnx \regex_match:nnF {#1} {#2} { #3 }
}
\AssertAdd { StrMatch } { __assert_str_match:nnF } { ~ }
```

```
\cs_new:Npn \__assert_str_match:nnT #1#2#3 {
  \exp_args:Nnx \regex_match:nnT {#1} {#2} { #3 }
}
\AssertAdd { StrNoMatch } { __assert_str_match:nnT } { ~ }
```

3.6.3 Integers

```
\cs_new:Npn \__assert_int_if_eq:nnF #1#2#3 {
  \int_compare:nNnF {\fp_eval:n {floor(#1)}} = {\fp_eval:n {floor(#2)}} { #3 }
}
```

```
\AssertAdd { IntEqual } { __assert_int_if_eq:nnF } { == }
\cs_new:Npn \__assert_int_if_eq:nnT #1#2 #3 {
  \int_compare:nNnT {\fp_eval:n {floor(#1)}} = {\fp_eval:n {floor(#2)}} { #3 }
}
\AssertAdd { IntNotEqual } { __assert_int_if_eq:nnT } { != }
\cs_new:Npn \__assert_int_if_lt:nnF #1#2#3 {
  \int_compare:nNnF {\fp_eval:n {floor(#1)}} < {\fp_eval:n {floor(#2)}}  { #3 }
}
\AssertAdd { IntLess } { __assert_int_if_lt:nnF } { < }
\cs_new:Npn \__assert_int_if_lt:nnT #1#2#3 {
  \int_compare:nNnT {\fp_eval:n {floor(#1)}} < {\fp_eval:n {floor(#2)}}  { #3 }
}
\AssertAdd { IntNotLess } { __assert_int_if_lt:nnT } { >= }
\cs_new:Npn \__assert_int_if_gt:nnF #1#2#3 {
  \int_compare:nNnF {\fp_eval:n {floor(#1)}} > {\fp_eval:n {floor(#2)}}  { #3 }
}
\AssertAdd { IntGreater } { __assert_int_if_gt:nnF } { > }
\cs_new:Npn \__assert_int_if_gt:nnT #1#2#3 {
  \int_compare:nNnT {\fp_eval:n {floor(#1)}} > {\fp_eval:n {floor(#2)}}  { #3 }
}
\AssertAdd { IntNotGreater } { __assert_int_if_gt:nnT } { <= }
```

3.6.4 Floats

```
\cs_new:Npn \__assert_fp_if_eq:nnF #1#2#3 {
  \fp_compare:nNnF {#1} = {#2} { #3 }
}
\AssertAdd { FPEqual } { __assert_fp_if_eq:nnF } { == }
\cs_new:Npn \__assert_fp_if_eq:nnT #1#2 #3 {
  \fp_compare:nNnT {#1} = {#2} { #3 }
}
\AssertAdd { FPNotEqual } { __assert_fp_if_eq:nnT } { != }
\cs_new:Npn \__assert_fp_if_lt:nnF #1#2#3 {
  \fp_compare:nNnF {#1} < {#2}  { #3 }
}
\AssertAdd { FPLess } { __assert_fp_if_lt:nnF } { < }
\cs_new:Npn \__assert_fp_if_lt:nnT #1#2#3 {
  \fp_compare:nNnT {#1} < {#2}  { #3 }
}
\AssertAdd { FPNotLess } { __assert_fp_if_lt:nnT } { >= }
\cs_new:Npn \__assert_fp_if_gt:nnF #1#2#3 {
  \fp_compare:nNnF {#1} > {#2}  { #3 }
}
\AssertAdd { FPGreater } { __assert_fp_if_gt:nnF } { > }
\cs_new:Npn \__assert_fp_if_gt:nnT #1#2#3 {
  \fp_compare:nNnT {#1} > {#2}  { #3 }
}
\AssertAdd { FPNotGreater } { __assert_fp_if_gt:nnT } { <= }
```

3.6.5 Almost

Due to rounding errors, some weaker assertions are available on floating point variables. Tuples are not supported.

```
\fp_gset:Nn \AssertPrecision { 10^-6 }
\cs_new:Npn \AssertUsePrecision {
  \fp_use:N \AssertPrecision
}
\cs_new:Npn \AssertSetPrecision #1 {
  \fp_gset:Nn \AssertPrecision { #1 }
  \ignorespaces
}
```

```
\cs_new:Npn \__assert_almost_if_eq:nnF #1#2#3 {
  \fp_compare:nNnT
    {abs(#1-#2)/(1+abs(#1)+abs(#2))} > { \AssertPrecision }
    { #3 }
}
\AssertAdd { AlmostEqual } { __assert_almost_if_eq:nnF } { == }
```

```
\cs_new:Npn \__assert_almost_if_eq:nnT #1#2 #3 {
  \fp_compare:nNnF
    {abs(#1-#2)/(1+abs(#1)+abs(#2))} > { \AssertPrecision }
    { #3 }
}
\AssertAdd { AlmostNotEqual } { __assert_almost_if_eq:nnT } { != }
```

```
\cs_new:Npn \__assert_almost_if_lt:nnF #1#2#3 {
  \fp_compare:nNnT {(#1-#2)/(1+abs(#1)+abs(#2))} > { \AssertPrecision } { #3 }
}
\AssertAdd { AlmostLess } { __assert_almost_if_lt:nnF } { < }
```

```
\cs_new:Npn \__assert_almost_if_lt:nnT #1#2#3 {
  \fp_compare:nNnF {(#1-#2)/(1+abs(#1)+abs(#2))} > { \AssertPrecision } { #3 }
}
\AssertAdd { AlmostNotLess } { __assert_almost_if_lt:nnT } { >= }
```

```
\cs_new:Npn \__assert_almost_if_gt:nnF #1#2#3 {
  \fp_compare:nNnT {(#2-#1)/(1+abs(#1)+abs(#2))} > { \AssertPrecision } { #3 }
}
\AssertAdd { AlmostGreater } { __assert_almost_if_gt:nnF } { > }
```

```
\cs_new:Npn \__assert_almost_if_gt:nnT #1#2#3 {
  \fp_compare:nNnF {(#2-#1)/(1+abs(#1)+abs(#2))} > { \AssertPrecision } { #3 }
}
\AssertAdd { AlmostNotGreater } { __assert_almost_if_gt:nnT } { <= }
```

3.7 `\Assert` command

The assertions are straightforward wrappers over LaTeX3 conditionals and private conditionals. This command is defined in two steps to allow for testing. The high level command may raise a fatal error whereas the lower won't.

```
\NewDocumentCommand\Assert {mO{n}mO{n}m} {
  \AssertCore {#1} {#2} {#3} {#4} {#5} {} {} { \msg_fatal:nn{Assert}{:n} }
}
```

`\AssertCore`

```
\AssertCore {⟨type⟩}
  {⟨actual expansion⟩} {⟨actual⟩}
  {⟨expected expansion⟩} {⟨expected⟩}
  {⟨failure code⟩} {⟨known type code⟩} {⟨unknown type code:n⟩}
```

```
\cs_generate_variant:Nn \prop_get:NnN { cxN }
\cs_new:Npn \AssertCore #1#2#3#4#5#6#7#8 {
```

Branch according to ⟨*type*⟩: is it acceptable?

```
  \prop_if_in:NnTF \g__assert_type_prop {#1} {
```

⟨*type*⟩ is acceptable, increment the number of tests performed and localize next oparations in a group.

```
    \int_gincr:N \g__assert_int
    \group_begin:
```

Build the shared information

```
    \AssertSet {/type} {#1}
    \AssertSet {/actual} [#2] {#3}
    \AssertSet {/expected} [#4] {#5}
```

Build the operator

```
    \prop_get:NnN \g__assert_type_prop {#1} \l__assert_tmpa_tl
    \AssertSet {/operator}
      [x] { \exp_last_unbraced:NV \use_ii:nn \l__assert_tmpa_tl }
    \UseHook{cmd/Assert/#1/before}
    \tl_set:Nx \l__assert_tmpb_tl
      { \exp_last_unbraced:NV \use_i:nn \l__assert_tmpa_tl }
    \tl_replace_once:Nnn \l__assert_tmpb_tl { :nn } { :#2#4 }
    \cs:w \l__assert_tmpb_tl \cs_end:
    {#3} {#5} {
      \seq_gput_right:Nx \g__assert_seq { \int_use:N \inputlineno }
      \seq_gput_right:Nx \g__assert_seq {
        \str_if_eq:VnTF\CurrentFile {} {\c_sys_jobname_str}{\CurrentFile}
      }
      \prop_get:cxN { dict:assert/shared } { /actual } \l__assert_tmpa_tl
      \seq_gput_right:NV \g__assert_seq \l__assert_tmpa_tl
      \prop_get:cxN { dict:assert/shared } { /operator } \l__assert_tmpa_tl
      \seq_gput_right:NV \g__assert_seq \l__assert_tmpa_tl
      \prop_get:cxN { dict:assert/shared } { /expected } \l__assert_tmpa_tl
      \seq_gput_right:NV \g__assert_seq \l__assert_tmpa_tl
      \UseHook{cmd/Assert/before failure}
      \UseHook{cmd/Assert/#1/failure}
      \UseHook{cmd/Assert/after failure}
      #6
    }
    \UseHook{cmd/Assert/#1/after}
    \group_end:
```

Finally execute ⟨*known type code*⟩

```
    #7
  } {
```

⟨*type*⟩ is not acceptabl: finally execute ⟨*unknown type code:n*⟩

```
    #8 {Unknown~assertion~type:~#1}
  }
  \ignorespaces
}
```

3.8 Summary of the tests

Log information about the test performed. On failure, informations are displayed all at once and a fatal error is raised.

In order to test this package, a macro is used to raise this fatal error. This macro will be overriden while testing to neutralize the fatal error and continue.

`\__assert_raise:n`

`\__assert_raise:n {`⟨*msg*⟩`}`

When ⟨*msg*⟩ is empty, a fatal error is raised, otherwise a note is printed.

```
\cs_new:Npn \__assert_raise:n #1 {
  \tl_if_eq:nnTF {} {#1} {
    \msg_note:nnx {Assert} {:n} {#1}
  } {
    \msg_fatal:nn{Assert}{:n}{Fatal error}
  }
}
```

```
\AtEndDocument{
  \int_compare:nNnTF { \seq_count:N  \g__assert_seq } > 0 {
    \msg_note:nnx{Assert}{:n}{
      \int_eval:n { \g__assert_int - \seq_count:N \g__assert_seq / 5 }~
        tests~passed~out~of~\int_use:N \g__assert_int
    }
    \int_step_inline:nnnn 1 5 { \seq_count:N \g__assert_seq } {
      \msg_note:nnxxx {Assert} {failure:nnn} {
        \seq_item:Nn \g__assert_seq  {#1}
      } {
        \seq_item:Nn \g__assert_seq { \int_eval:n{#1+1} }
      } {
        \seq_item:Nn \g__assert_seq { \int_eval:n{#1+2} }~
        \seq_item:Nn \g__assert_seq { \int_eval:n{#1+3} }~
        \seq_item:Nn \g__assert_seq { \int_eval:n{#1+4} }
      }
    }
    \__assert_raise:n {}
  } {
    \group_begin:
    \tl_set:Nn \l_tmpa_tl {All~\int_use:N \g__assert_int\space tests~passed}
    \msg_note:nnx {Assert} {:n} {\l_tmpa_tl}
    \cs_if_exist:NT \ASSERTSTR {\l_tmpa_tl}
    \group_end:
  }
}
```

3.9 Messaging

Replace the default 'Package' displayed in the log by 'Test'

```
\prop_gput:Nnn \g_msg_module_type_prop {Assert} {Test}
```

All purposes message with one argument:

```
\msg_new:nnn {Assert} {:n} {#1}
```

Failure message with three arguments:

```
\msg_new:nnn {Assert} {failure:nnn} {
  Failure~at~line~#1~of~file~#2:~#3
}
```

```
⟨/pkg⟩
```