References

- Basic structure
  - Example group
    - `ExampleGroup` / `Describe` / `Context`
  - Example
    - `Example` / `It` / `Specify`
  - Evaluation
    - `When call`
    - `When run`
      - about calling shell function with run
    - `When run command`
    - `When run script`
    - `When run source`
    - Comparison
  - Expectation
    - `The ... should (not)`
    - `Assert`
  - Subjects
    - `stdout` (`output`) subject
    - `stderr` (`error`) subject
    - `status` subject
    - `line` subject
    - `word` subject
    - `path` / `file` / `directory` subject
    - `function` subject
    - `value` subject
    - `variable` subject
  - Modifiers
    - `line` modifier
    - `lines` modifier
    - `word` modifier
    - `length` modifier
    - `contents` modifier
    - `result` modifier
  - Matchers
    - `satisfy` matcher
    - stat matchers
      - `be exist` matcher
      - `be file` matcher
      - `be directory` matcher
      - `be empty file` matcher
      - `be empty directory` matcher
      - `be symlink` matcher
      - `be pipe` matcher
      - `be socket` matcher
      - `be readable` matcher
      - `be writable` matcher

            - `be executable` matcher
            - `be block_device` matcher
            - `be character_device` matcher
            - `have setgid` matcher
            - `have setuid` matcher
        * status matchers
            - `be success` matcher
            - `be failure` matcher
        * string matchers
            - `equal` matcher
            - `start with` matcher
            - `end with` matcher
            - `include` matcher
            - `match pattern` matcher
        * `successful` matcher
        * valid matchers
        * variable matchers
            - `be defined` matcher
            - `be undefined` matcher
            - `be present` matcher
            - `be blank` matcher
            - `be exported` matcher
            - `be readonly` matcher
- Helper
    - Skip / Pending
        * `Skip`
        * `Skip if`
        * `Pending`
        * `Todo`
    - Data
        * `Data[:raw]`
        * `Data:expand`
        * `Data <FUNCTION>`
        * `Data "<STRING>"`
        * `Data < "<FILE>"`
    - Parameters
        * `Parameters[:block]`
        * `Parameters:value`
        * `Parameters:matrix`
        * `Parameters:dynamic`
    - Others
        * `Include`
        * `Path` / `File` / `Dir`
        * `Intercept`
        * `Set`
        * `Dump`
- Hooks
    - `Before` / `After`

    - `BeforeAll` / `AfterAll`
    - `BeforeCall` / `AfterCall`
    - `BeforeRun` / `AfterRun`
- Directive
    - `%const` (`%`)
    - `%text`
    - `%puts` (`%-`) / `%putsn` (`%=`)
    - %preserve
    - `%logger`
- Special environment Variables

Basic structure

You can write a structured *Example* by using the DSL shown below:

Example group

DSL	Description
ExampleGroup ... End	Define an example group.
Describe ... End	Synonym for `ExampleGroup`.
Context ... End	Synonym for `ExampleGroup`.

`ExampleGroup` / `Describe` / `Context` Example groups are nestable.

Example

DSL	Description
Example ... End	Define an example.
It ... End	Synonym for `Example`.
Specify ... End	Synonym for `Example`.

`Example` / `It` / `Specify`

Evaluation

The line beginning with `When` is the evaluation.

Evaluation	Description
When call	Call shell function without subshell.
When run	Run shell function or external command within a subshell.
When run command	Run external command (including non-shell scripts).
When run script	Run shell script by new process of the current shell.
When run source	Run shell script in the current shell by `.` command (aka `source`).

When call

`When call <FUNCTION> [ARGUMENTS...]`

This is primarily designed for shell function calls. It is the recommended evaluation as a unit test. It does not use a subshell, therefore it is the fastest evaluation variant and you can assert variables.

When run

`When run <FUNCTION | COMMAND> [ARGUMENTS...]`

This is primarily designed for external command calls. The external command does not have to be a shell script. Even shell scripts are executed as external commands according to the shebang, so they are not covered by the coverage.

about calling shell function with run If a shell function is specified, it will be executed in a subshell. The slight advantage of executing shell functions with `run` is that you can trap errors with `set -e`. Unlike `call`, it does not cause an error, so you can assert the exit status.

Also, because of the execution in the subshell, the variables which change values in the function are restored once `run` finishes. This is often a disadvantage, but tests of ShellSpec itself intentionally use `run` because changing internal variables confuses ShellSpec's behavior.

If you want to assert variables with `run`, use the `%preserve` directive in function called by `AfterRun` hook. It can preserve variables even if `run` exits the subshell.

When run command

`When run command <COMMAND> [ARGUMENTS...]`

Run an external command explicitly. The external command does not have to be a shell script. Even shell scripts are executed as external commands according to the shebang, so they are not covered by the coverage.

When run script

`When run script <SCRIPT> [ARGUMENTS...]`

Run the external shell script in the same shell as the currently running shell.

When run source

`When run source <SCRIPT> [ARGUMENTS...]`

This is similar to `run script`, but simulates the running of shell scripts using the `.` command instead of running directly.

The advantage over `run script` is that you can use `Intercept` to intercept at any point in the external shell script. This is useful for preparation testing and mocking with shell functions.

Comparison

	`call`	`run`	`run command`	`run script`	`run source`
Run in subshell	No	Yes	Yes	Yes	Yes
Target	function	function / command	command	shell script	shell script
Stop with `set -e`	No	Yes	-	Yes	Yes
Catch `exit`	No	Yes	-	Yes	Yes
Expectation	BeforeCall / AfterCall	BeforeRun / AfterRun	BeforeRun / AfterRun	BeforeRun / AfterRun	BeforeRun / AfterRun
Hooks	No	No	-	No	Yes
Intercept	Yes	Yes (function only)	No	Yes	Yes
Coverage					

Expectation

`The ... should (not)` The line beginning with `The` is the evaluation. The *subject* or the *modifier* follows after `The`. And last is the *matcher*.

```
The [MODIFIER of...] <SUBJECT> should <MATCHER>
The [MODIFIER of...] <SUBJECT> should not <MATCHER>
```

`Assert`

```
Assert <FUNCTION> [ARGUMENTS...]
```

Subjects

Subject	Description
stdout / output	Use the stdout of *Evaluation* as subject.
line	Same as `line NUMBER of stdout`.
word	Same as `word NUMBER of stdout`.
stderr / error	Use the stderr of *Evaluation* as subject.
status	Use the status of *Evaluation* as subject.
path / file / directory	Use the alias resolved path as the subject.
value	Use the value as the subject.
function	Use the function name as the subject.
variable	Use the value of the variable as the subject.

`stdout` (`output`) subject

```
The stdout should equal "foo"
The output should equal "foo"
```

`stderr` (`error`) subject

```
The stderr should equal "foo"
The error should equal "foo"
```

`status` subject

```
The status should be success
```

`line` subject

When combined with line, `stdout` can be omitted.

```
The line 1 of stdout should equal foo
The line 1 should equal foo # stdout omitted
```

`word` subject

When combined with word, `stdout` can be omitted.

```
The word 1 of stdout should equal foo
The word 1 should equal foo # stdout omitted
```

`path` / `file` / `directory` subject

```
Path data-file=/tmp/data.txt
The path data-file should exist
```

`function` subject

```
The result of function foo should be successful
The result of "foo()" should be successful # shorthand
```

`value` subject

```
The value "foo" should equal "foo"
```

I do not recommend using this subject as it may not generate clear failure messages. Use the `variable` subject instead.

`variable` subject

```
The variable var should equal "foo"
```

Modifiers

Modifier	Description
line	The specified line of the subject.
lines	The number of lines of the subject.
word	The specified word of the subject.
length	The length of the subject.
contents	The contents of the file as subject.
result	The result of the function as subject.

`line` modifier

```
The line 1 of stdout should equal "line1"
```

`lines` modifier

```
The lines of stdout should equal 5
```

`word` modifier

```
The word 2 of stdout should equal "word2"
```

`length` modifier

```
The length of value "abcd" should equal 5
```

`contents` modifier

```
The contents of file "/tmp/file.txt" should equal "temp data"
```

`result` modifier

```
get_version() {
  # The result of the evaluation is passed as arguments
  # $1: stdout, $2: stderr, $3: status
  echo "$1" | grep -o '[0-9.]*' | head -1
}

When call echo "GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)"
The result of function get_version should equal "4.4.20"
The result of "get_version()" should equal "4.4.20" # shorthand

check_version() {
  # The result of the evaluation is passed as arguments
  # $1: stdout, $2: stderr, $3: status
  [ "$("$1" | grep -o '[0-9.]*' | head -1)" = "4.4.20" ]
}

When call echo "GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)"
The result of function check_version should be successful
The result of "check_version()" should be successful # shorthand
```

Matchers

`satisfy` matcher

Matcher	Description
satisfy	The subject should satisfy <FUNCTION>

```
satisfy <FUNCTION> [ARGUMENTS...]
```

satisfy examples

```
value() {
  # The subject is stored in the same variable name as the function name
  test "${value:?}" "$1" "$2"
}

formula() {
  value=${formula:?}
  [ $(($1)) -eq 1 ]
}

When call echo "50"
The output should satisfy value -gt 10
The output should satisfy formula "10 <= value && value <= 100"
```

stat matchers the subject expected file path

Matcher	Description
exist	The file should exist.
~~be exist~~	The file should exist. (deprecated)
be file	The file should be a file.
be directory	The file should be a directory.
be empty file	The file should be an empty file.
be empty directory	The directory should be an empty directory.
be symlink	The file should be a symlink.
be pipe	The file should be a pipe.
be socket	The file should be a socket.
be readable	The file should be readable.
be writable	The file should be writable.
be executable	The file should be executable.
be block_device	The file should be a block device.
be character_device	The file should be a character device.
~~has setgid~~	The file should have the setgid flag set. (deprecated)
have setgid	The file should have the setgid flag set.
~~has setuid~~	The file should have the setuid flag set. (deprecated)
have setuid	The file should have the setuid flag set.

`exist` matcher

```
The path /target/path should exist
```

or

```
The path /target/path should be exist
```

`be file` matcher

```
The path /target/path should be file
```

`be directory` matcher

```
The path /target/path should be directory
The path /target/path should be dir
```

`be empty file` matcher

```
The path /target/path should be empty file
```

`be empty directory` matcher

```
The path /target/path should be empty directory
The path /target/path should be empty dir
```

`be symlink` matcher

```
The path /target/path should be symlink
```

`be pipe` matcher

```
The path /target/path should be pipe
```

`be socket` matcher

```
The path /target/path should be socket
```

`be readable` matcher

```
The path /target/path should be readable
```

`be writable` matcher

```
The path /target/path should be writable
```

`be executable` matcher

```
The path /target/path should be executable
```

`be block_device` matcher

```
The path /target/path should be block_device
```

`be character_device` matcher

```
The path /target/path should be character_device
```

`have setgid` matcher

```
The path /target/path should have setgid
```

`have setuid` matcher

```
The path /target/path should have setuid
```

status matchers the subject expected status

Matcher	Description
be success	The status should be success (`0`).
be failure	The status should be failure (`1 - 255`).

`be success` matcher

```
The status should be success
```

`be failure` matcher

```
The status should be failure
```

string matchers

Matcher	Description
equal <STRING>eq <STRING>	The subject should equal <STRING>
start with <STRING>	The subject should start with <STRING>
end with <STRING>	The subject should end with <STRING>
include <STRING>	The subject should include <STRING>
match pattern <PATTERN>	The subject should match pattern <PATTERN>

`equal` matcher

```
The output should equal <STRING>
The output should eq <STRING>
```

`start with` matcher

```
The output should start with <STRING>
```

`end with` matcher

```
The output should end with <STRING>
```

`include` matcher

```
The output should include <STRING>
```

`match pattern` matcher

```
The output should match pattern <PATTERN>
```

PATTERN examples

- `foo*`

- `foo?`
- `[fF]oo`
- `[!F]oo`
- `[a-z]`
- `foo|bar`

`successful` matcher Use with result modifier.

valid matchers Plan to deprecate in the future.

Matcher	Description
be valid number	The subject should be a valid number.
be valid funcname	The subject should be a valid funcname.

variable matchers the subject expect variable

Matcher	Description
be defined	The variable should be defined (set).
be undefined	The variable should be undefined (unset).
be present	The variable should be present (non-zero length string).
be blank	The variable should be blank (unset or zero length string).
be exported	The variable should be exported.
be readonly	The variable should be readonly.

`be defined` matcher

```
The variable VAR should be defined
```

`be undefined` matcher

```
The variable VAR should be undefined
```

`be present` matcher

```
The variable VAR should be present
```

`be blank` matcher

```
The variable VAR should be blank
```

`be exported` matcher

```
The variable VAR should be exported
```

`be readonly` matcher

```
The variable VAR should be readonly
```

Helper

Skip / Pending

DSL	Description
Skip <REASON>	Skip current block.
Skip if <REASON> <FUNCTION> [ARGUMENTS...]	Skip current block with conditional.
Pending <REASON>	Pending current block.
Todo	Define pending example

Skip

Skip if

Pending

Todo

Data

DSL	Description
Data[:raw]#|...End	Define stdin data for evaluation (without expand variables).
Data:expand#|...End	Define stdin data for evaluation (with expand variables).
Data <FUNCTION> [ARGUMENTS...]	Use function for stdin data for evaluation.
Data "<STRING>"Data '<STRING>'	Use string for stdin data for evaluation.
Data < <FILE>	Use file for stdin data for evaluation.

NOTE: The `Data` helper can also be used with filters.

```
Data | tr 'abc' 'ABC' # comment
#|aaa
#|bbb
#|ccc
End
```

Data[:raw]

```
Describe 'Data helper'
  Example 'provide with Data helper block style'
    Data
      #|item1 123
      #|item2 456
      #|item3 789
```

```
    End
    When call awk '{total+=$2} END{print total}'
    The output should eq 1368
  End
End
```

Data:expand

Data <FUNCTION>

Data "<STRING>"

Data < "<FILE>"

Parameters

DSL	Description
Parameters ... End	Define parameters (block style)
Parameters:block ... End	Same as Parameters
Parameters:value [VALUES...]	Define parameters (value style)
Parameters:matrix ... End	Define parameters (matrix style)
Parameters:dynamic ... End	Define parameters (dynamic style)

NOTE: Multiple `Parameters` definitions are merged.

Parameters[:block]

```
Describe 'example'
  Parameters
    "#1" 1 2 3
    "#2" 1 2 3
  End

  Example "example $1"
    When call echo "$(($2 + $3))"
    The output should eq "$4"
  End
End
```

Parameters:value

```
Parameters:value foo bar baz
```

Parameters:matrix

```
Parameters:matrix
  foo bar
  1 2
```

```
  # expanded as follows
  #   foo 1
  #   foo 2
  #   bar 1
  #   bar 2
End
```

Parameters:dynamic

```
Parameters:dynamic
  for i in 1 2 3; do
    %data "#$i" 1 2 3
  done
End
```

Only `%data` directives can be used within a `Parameters:dynamic` block. You can not call a function or access variables defined within the specfile. You can refer to variables defined with `%const`.

Others

DSL	Description
Include <NAME>	Include other files.
PathFileDir	Define a path alias.
Intercept [NAMES...]	Define an interceptor.
Set [OPTION:<on | off>...]	Set shell option before running each example.
Dump	Dump stdout, stderr and status for debugging.

Include

Path / File / Dir

Intercept

Set

Dump

Hooks

DSL	Description
Before	Define a hook called before running each example.
After	Define a hook called after running each example.

DSL	Description
BeforeAll	
AfterAll	
BeforeCall	
AfterCall	
BeforeRun	
AfterRun	

`Before` / `After`

`BeforeAll` / `AfterAll`

`BeforeCall` / `AfterCall`

`BeforeRun` / `AfterRun`

Directive

Directive	Description
%const, %	Define a constant variable.
%text	Define a multiline texts to output to stdout.
%putsn, %=	Output arguments with the newline.
%puts, %-	Output arguments.
%logger	Output log message.

`%const` (`%`)

```
% <VERNAME>: "<VALUE>"
```

`%text`

`%puts` (`%-`) / `%putsn` (`%=`)

%preserve

Use this with the `When run` evaluation.

`%logger`

Special environment Variables

ShellSpec provides special environment variables with prefix `SHELLSPEC_`. They are useful for writing tests and extensions. I'll try to avoid making breaking changes to these, but can't guarantee it. There are many undocumented variables. You can use them at your own risk.

These variables can be overridden by the `--env-from` option, except for some variables. This is an assumed usage, but has not been fully tested. It is recommended to use it as read-only.

Name	Description	Value
SHELLSPEC_ROOT	ShellSpec root directory	
SHELLSPEC_LIB	ShellSpec lib directory	`${SHELLSPEC_ROOT}/lib`
SHELLSPEC_LIBEXEC	ShellSpec libexec directory	`${SHELLSPEC_ROOT}/libexec`
SHELLSPEC_TMPDIR	Temporary directory	`${TMPDIR}` or `/tmp` if not specified.
SHELLSPEC_TMPBASE	Temporary directory used by ShellSpec	`${SHELLSPEC_TMPDIR}/shellspec.${SHELLSPEC_UNIXTI`
SHELLSPEC_WORKDIR	Temporary directory for each spec number	`${SHELLSPEC_TMPBASE}/${SHELLSPEC_SPEC_NO}.`
SHELLSPEC_PROJECT_ROOT	Where `.shellspec` is located	
~~SHELLSPEC_SPECDIR~~	Specfile directory	`${SHELLSPEC_PROJECT_ROOT}/spec` [depricated]
SHELLSPEC_HELPERDIR	spec_helper directory	`${SHELLSPEC_PROJECT_ROOT}/spec` (default)
SHELLSPEC_LOAD_PATH	Load path of library	`${SHELLSPEC_SPECDIR}:${SHELLSPEC_LIB}:${SHELLSPE`
SHELLSPEC_UNIXTIME	Unix time when ShellSpec starts	
SHELLSPEC_SPECFILE	Current running specfile path	
SHELLSPEC_SPEC_NO	Current specfile number	
SHELLSPEC_GROUP_ID	Current group ID	e.g. `1-2`
SHELLSPEC_EXAMPLE_ID	Current example ID (including group ID)	e.g. `1-2-3`
SHELLSPEC_EXAMPLE_NO	Current serial number of example	