# The `build2` Testscript Language

Copyright © 2014-2024 the build2 authors.
Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision `0.17`, May 2024
This revision of the document describes the `build2` Testscript language `0.17.x` series.

# Table of Contents

Preface . . . . . 1
1 Introduction . . . . . 1
2 Build System Integration . . . . . 10
3 Model and Execution . . . . . 14
4 Lexical Structure . . . . . 19
5 Syntax and Semantics . . . . . 23
5.1 Notation . . . . . 23
5.2 Grammar . . . . . 24
5.3 Script . . . . . 27
5.4 Scope . . . . . 27
5.5 Scope-If . . . . . 28
5.6 Directive . . . . . 28
5.6.1 Include . . . . . 28
5.7 Setup and Teardown . . . . . 28
5.8 Test . . . . . 29
5.9 Variable . . . . . 29
5.10 Variable-If . . . . . 29
5.11 Variable-For . . . . . 30
5.12 Variable-While . . . . . 31
5.13 Command . . . . . 31
5.14 Command-If . . . . . 32
5.15 Command-For . . . . . 33
5.16 Command-While . . . . . 34
5.17 Redirect . . . . . 34
5.18 Input Redirect . . . . . 34
5.19 Output Redirect . . . . . 35
5.20 Here-Document . . . . . 36
5.21 Output Regex . . . . . 37
5.22 Cleanup . . . . . 39
5.23 Description . . . . . 40
6 Builtins . . . . . 41
6.1 `cat` . . . . . 41
6.2 `cp` . . . . . 41
6.3 `date` . . . . . 42
6.4 `diff` . . . . . 43
6.5 `echo` . . . . . 43
6.6 `env` . . . . . 43
6.7 `exit` . . . . . 44
6.8 `export` . . . . . 44
6.9 `false` . . . . . 45
6.10 `find` . . . . . 45
6.11 `ln` . . . . . 45
6.12 `mkdir` . . . . . 46
6.13 `mv` . . . . . 46

6.14 `rm` . . . . . 47
6.15 `rmdir` . . . . . 48
6.16 `sed` . . . . . 48
6.17 `set` . . . . . 49
6.18 `sleep` . . . . . 50
6.19 `test` . . . . . 50
6.20 `timeout` . . . . . 50
6.21 `touch` . . . . . 51
6.22 `true` . . . . . 51
7 Style Guide . . . . . 52

# Preface

This document describes the `build2` Testscript language. It starts with a discussion of the motivation behind a separate domain-specific language for running tests and then introduces a number of Testscript concepts with examples. The remainder of the document provides a more formal specification of the language, including its integration into the build system, conceptual model and execution, lexical structure, as well as syntax and semantics. The final chapter describes the testing guidelines and the Testscript style as used in the `build2` project itself.

In this document we use the term *Testscript* (capitalized) to refer to the Testscript language. Just *testscript* means code written in this language. For example: "We can pass additional information to testscripts using target-specific variables." Finally, `testscript` refers to the file name.

We also use the equivalent distinction between *Buildfile* (language), *buildfile* (code), and `buildfile` (file).

# 1 Introduction

The `build2 test` module provides the ability to run an executable target as a test along with passing options and arguments, providing the `stdin` input, as well as comparing the `stdout` output to the expected result. For example:

```
exe{hello}: file{names.txt}: test.stdin = true
exe{hello}: file{greetings.txt}: test.stdout = true
exe{hello}: test.options   = --greeting 'Hi'
exe{hello}: test.arguments = - # Read names from stdin.
```

This works well for simple, single-run tests. If, however, our testing required multiple runs with varying inputs and/or analyzing output, traditionally, we would resort to using a scripting language, for instance Bash or Python. This, however, has a number of drawbacks. Firstly, this approach is not portable (there is no Bash or Python on Windows *out of the box*). It is also hard to write concise tests in a general-purpose scripting language. The result is often a test suite that has grown incomprehensible with everyone dreading adding new tests. Secondly, it is hard to run such tests in parallel without major effort. Usually this involves having a separate script for each test and implementing some kind of a test harness.

Testscript is a domain-specific language for running tests. It vaguely resembles Bash and is optimized for concise test description and fast execution by focusing on the following functionality:

- Supplying input via command line and `stdin`.
- Comparing to expected exit status.
- Comparing `stdout`/`stderr` to expected output, including using regex.
- Setup/teardown commands and automatic file/directory cleanups.
- Simple (single-command) and compound (multi-command) tests.

- Test groups with common setup/teardown.
- Test isolation for parallel execution.
- Portable POSIX-like builtins subset.
- Test documentation.

Note that Testscript is a *test runner*, not a testing framework for a particular programming language. It does not concern itself with how the test executables themselves are implemented. As a result, it is mostly geared towards functional testing but can also be used for unit testing if external input/output is required. Testscript is part of the `build2` build system and is implemented by its `test` module.

As a quick introduction to the Testscript's capabilities, let's *properly* test a "Hello, World" program. For a simple implementation the corresponding `buildfile` might look like this:

```
exe{hello}: cxx{hello}
```

We also assume that the project's `bootstrap.build` loads the `test` module which implements the execution of testscripts.

To start, we create an empty file called `testscript`. To indicate that a testscript file tests a specific target we simply list it as a target's prerequisite, for example:

```
exe{hello}: cxx{hello} testscript
```

Let's assume our `hello` program expects us to pass the name to greet as a command line argument. And if we don't pass anything, it prints an error followed by usage and terminates with a non-zero exit code. We can test this failure case by adding the following line to the `testscript` file:

```
$* 2>- != 0
```

While it sure is concise, it may look cryptic without an explanation. When the `test` module runs tests, it passes to each testscript the path to the target of which this testscript is a prerequisite. So in our case the testscript will receive the path to our `hello` executable. The buildfile can also pass along additional options and arguments (see Build System Integration for details). Inside the testscript, all of this (target path, options, and arguments) are bound to the `$*` variable. So, in our case, if we expand the above line, it would be something like this:

```
/tmp/hello/hello 2>- != 0
```

Or, if we are on Windows, something like this:

```
C:\projects\hello\hello.exe 2>- != 0
```

The `2>-` redirect is the Testscript equivalent of `2>/dev/null` that is both portable and more concise (`2` here is the `stderr` file descriptor). If we don't specify it and our program prints anything to `stderr`, then the test fails (unexpected output).

The remainder of the command (`!= 0`) is the exit status check. If we don't specify it, then the test is expected to return zero exit code (which is equivalent to specifying `== 0`).

If we run our test, it will pass provided our program behaves as expected. One thing our test doesn't verify, however, is the diagnostics that gets printed to `stderr` (remember, we ignored it with `2>-`). Let's fix that assuming this is the code that prints it:

```
cerr << "error: missing name" << endl
     << "usage: " << argv[0] << " <name>" << endl;
```

In Testscript you can compare output to the expected result for both `stdout` and `stderr`. We can supply the expected result as either a *here-string* or *here-document*, both which can be either literal or regex. The here-string approach works best for short, single-line output and we will use it for another test in a minute. For this test let's use the here-document since the expected diagnostics has two lines:

```
$* 2>>EOE != 0
error: missing name
usage: hello <name>
EOE
```

Let's decrypt this: the `2>>EOE` is a here-document redirect with `EOE` (stands for End-Of-Error) being the string we chose to mark the end of the here-document fragment. Next comes the here-document fragment followed by the end marker.

Now, when executing this test, the `test` module will check two things: it will compare the `stderr` output to the expected result using the `diff` tool and it will make sure the test returns a non-zero exit code. Let's give it a go:

```
$ b test
testscript:1:1: error: hello stderr doesn't match expected
  info: stderr: test-hello/1/stderr
  info: expected stderr: test-hello/1/stderr.orig
  info: stderr diff: test-hello/1/stderr.diff
--- test-hello/1/stderr.orig
+++ test-hello/1/stderr
@@ -1,2 +1,2 @@
 error: missing name
-usage: hello <name>
+usage: /tmp/hello/hello <name>
```

While not what we hoped for, at least the problem is clear: the program name varies at runtime so we cannot just hardcode `hello` in our expected output. How do we solve this? The best fix would be to use the actual path to the target; after all, we know it's the first element in `$*`:

```
$* 2>>"EOE" != 0
error: missing name
usage: $0 <name>
EOE
```

You can probably guess what `$0` expands to. But did you notice another change? Yes, those double quotes in `2>>"EOE"`. Here is what's going on: similar to Bash, single-quoted strings (`'foo'`) are taken literally while double-quoted ones (`"foo"`) have variable expansions, escaping, and so on. In Testscript this semantics is extended to here-documents in a curious way: if the end marker is single-quoted then the here-document lines are taken literally and if it is double-quoted, then there can be variable expansions, etc. An unquoted end marker is treated as single-quoted (note that this is unlike Bash where here-documents always have variable expansions).

This example illustrated a fairly common testing problem: output variability. In our case we could fix it perfectly since we could easily calculate the varying part exactly. But often figuring out the varying part is difficult if not outright impossible. A good example would be a system error message based on the `errno` code, such as file not being found. Different C runtimes can phrase the message slightly differently or it can be localized. Worse, it can be a slightly different error code, for example `ENOENT` vs `ENOTDIR`.

To handle output variability, Testscript allows us to specify the expected output as a regular expression. For example, this is an alternative fix to our usage problem that simply ignores the program name:

```
$* 2>>~/EOE/ != 0
error: missing name
/usage: .+ <name>/
EOE
```

Let's explain what's going here: to use a regex here-string or here-document we add the `~` *redirect modifier*. In this case the here-document end marker must start and end with the regex introducer character of your choice (`/` in our case). Any line inside the here-document fragment that begins with this introducer is then treated as a regular expression rather than a literal (see Output Regex for details).

While this was a fairly deep rabbit hole for a first example, it is a good illustration of how quickly things get complicated when testing real-world software.

Now that we have tested the failure case, let's test the normal functionality. While we could have used a here-document, in this case a here-string will be more concise:

```
$* 'World' >'Hello, World!'
```

It's also a good idea to document our tests. Testscript has a formalized test description that can capture the test *id*, *summary*, and *details*. All three components are optional and how thoroughly you document your tests is up to you.

The description lines precede the test command. They start with a colon (`:`), and have the following layout:

```
: <id>
: <summary>
:
: <details>
: ...
```

The recommended format for `<id>` is `<keyword>-<keyword>...` with at least two keywords. The id is used in diagnostics, to name the test working directory, as well as to run individual tests. It can only contain alphanumeric characters as well as underscores, pluses, and minuses. The recommended style for `<summary>` is that of the `git(1)` commit summary. The detailed description is free-form. Here are some examples (`#` starts a comment):

```
# Only id.
#
: missing-name
$* 2>>"EOE" != 0
...

# Only summary.
#
: Test handling of missing name
...

# Both id and summary.
#
: missing-name
: Test handling of missing name
...

# All three: id, summary, and a detailed description.
#
: missing-name
: Test handling of missing name
:
: This test makes sure the program detects that the name to greet
: was not specified on the command line and both prints usage and
: exits with non-zero code.
...
```

The recommended way to come up with an id is to distill the summary to its essential keywords by removing generic words like "test", "handle", and so on. If you do this, then both the id and summary will convey essentially the same information. As a result, to keep things concise, you may choose to drop the summary and only have the id (this is what we often do in `build2` tests). If the id is not provided, then it will be automatically derived from the line number in testscript (we have already seen one in the earlier failed test diagnostics).

Either the id or summary (but not both) can alternatively be specified inline in the test command after a colon (`:`), for example:

```
$* 'World' >'Hello, World!' : command-name
```

Similar to handling output, Testscript provides a convenient way to supply input to the test's `stdin`. Let's say our `hello` program recognizes the `-` argument as an instruction to read the names from `stdin`. This is how we could test this functionality:

```
$* - <<EOI >>EOO : stdin-names
Jane
John
EOI
Hello, Jane!
Hello, John!
EOO
```

As you might suspect, we can also use here-strings to supply `stdin`, for example:

```
$* - <'World' >'Hello, World!' : stdin-name
```

Let's say our `hello` program has a configuration file that captures custom name-to-greeting mappings. A path to this file can be passed with the `-c` option. To test this functionality we first need to create a sample configuration file. This calls for a multi-command or *compound* test, for example:

```
cat <<EOI >=hello.conf;
John = Howdy
Jane = Good day
EOI
$* -c hello.conf 'Jane' >'Good day, Jane!' : config-greet
```

Notice the semicolon (`;`) at the end of the first command: it indicates that the following command is part of the same test.

Other than that, you may be wondering what exactly is `cat`? While most POSIX systems will have a program with this name, there is no such thing, say, on vanilla Windows. To help with portability Testscript provides a subset (both in terms of the number and supported features) of POSIX utilities, such as, `echo`, `touch`, `cat`, `mkdir`, `rm`, and so on (see Builtins for details).

You may also be wondering why we don't have a third command, such as `rm`, that removes `hello.conf`? It is not necessary because this file will be automatically registered for cleanup that happens at the end of the test. We can also register our own files and directories for automatic cleanup. For example, if the `hello` program created the `hello.log` file on unsuccessful runs, then this is how we could have cleaned it up:

```
$* ... &hello.log != 0
```

What if we wanted to run two tests for this configuration file functionality? For example, we may want to test the custom greeting as above but also make sure the default greeting is not affected. One way to do this would be to repeat the `cat` command in each test. But there is a better way: in Testscript we can combine related tests into groups. For example:

```
: config
{
  conf = $~/hello.conf

  +cat <<EOI >=$conf
  John = Howdy
  Jane = Good day
  EOI

  $* -c $conf 'John' >'Howdy, John!' : custom-greet
  $* -c $conf 'Jack' >'Hello, Jack!' : default-greet
}
```

A test group is a scope that contains several tests. Variables set inside a scope (like our `conf`) are only in effect until the end of this scope. Groups can also perform common, non-test actions with *setup* and *teardown* commands. The setup commands start with the plus sign (+) and must come before the tests while teardown – with minus (−) and must come after the tests.

Note that setup and teardown commands are not part of any test (notice the lack of `;` after `+cat`), rather they are associated with the group itself. Their automatic cleanup only happens at the end of the scope (so our `hello.conf` will only be removed after all the tests in the group have completed).

A scope can also have a description. In particular, assigning a test group an id (`config` in our example) allows us to run tests only from this specific group.

The last thing we need to discuss in this example is `$~`. This variable stands for the scope working directory (we will talk more about working directories at the end of this introduction).

Besides explicit group scopes, each test is automatically placed in its own implicit test scope. However, we can make the test scope explicit, for example, for better visual separation of complex tests:

```
: config-greet
{
  conf = hello.conf

  cat <'Jane = Good day' >=$conf;
  $* -c $conf 'Jane' >'Good day, Jane!'
}
```

We can conditionally exclude sections of a testscript using the `if-else` branching. This can be done both at the scope level to exclude test or group scopes as well as at the command level to exclude individual commands or variable assignments. Let's start with a scope example by providing a Windows-specific implementation of a test:

```
: config-empty
:
if ($cxx.target.class != windows)
{
  $* -c /dev/null 'Jane' >'Hello, Jane!'
}
else
{
  $* -c nul 'Jane' >'Hello, Jane!'
}
```

Note that the scopes in the `if-else` chain are treated as variants of the same test or group thus the single description at the beginning.

Let's now see an example of command-level `if-else` by reimplementing the above as a single test with some branching and without using the `nul` device on Windows (notice the semicolon after `end`):

```
: config-empty
:
if ($cxx.target.class != windows)
  conf = /dev/null
else
  conf = empty
  touch $conf
end;
$* -c $conf 'Jane' >'Hello, Jane!'
```

You may have noticed that in the above examples we referenced the `cxx.target.class` variable as if we were in a buildfile. We could do that because the testscript variable lookup continues in the buildfile starting from the target being tested, then the testscript target, and continuing with the standard scope lookup (see Model and Execution for details). In particular, this means we can pass arbitrary information to testscripts using target-specific variables. For example, this is how we can move the above platform test to `buildfile`:

```
# buildfile

exe{hello}: cxx{hello} testscript

testscript{*}: windows = ($cxx.target.class == windows)

# testscript

if! $windows
  conf = /dev/null
else
  ...
```

Note also that in cases where you simply need to conditionally pick a value for a variable, the `build2` evaluation context will often be a more concise option. For example:

```
: config-empty
:
conf = ($windows ? nul : /dev/null);
$* -c $conf 'Jane' >'Hello, Jane!'
```

Similar to Bash, test commands can be chained with pipes (`|`) and combined with logical operators (`||` and `&&`). Let's say our `hello` program provided the `-o` option to write the result to a file instead of `stdout`. Here is how we could test it:

```
$* -o hello.out - <<EOI &hello.out && cat hello.out >>EOO
John
Jane
EOI
Hello, John!
Hello, Jane!
EOO
```

Similarly, if it had the `-r` option to reverse the greetings back to their names (as every `hello` program should), then we could write a test like this:

```
$* - <<EOI | $* -r - >>EOO
John
Jane
EOI
John
Jane
EOO
```

To conclude, let's put all our (sensible) tests together so that we can have a complete picture:

```
$* 'World' >'Hello, World!' : command-name

$* 'John' 'Jane' >>EOO      : command-names
Hello, Jane!
Hello, John!
EOO

$* - <<EOI >>EOO            : stdin-names
Jane
John
EOI
Hello, Jane!
Hello, John!
EOO

: config
{
  conf = $~/hello.conf

  +cat <<EOI >=$conf
  John = Howdy
  Jane = Good day
  EOI

  $* -c $conf 'John' >'Howdy, John!' : custom-greet
  $* -c $conf 'Jack' >'Hello, Jack!' : default-greet
}

$* 2>>"EOE" != 0            : missing-name
error: missing name
usage: $0 <name>
EOE
```

Testscript isolates tests from each other by running each test in its own temporary working directory under `out_base`. For the above `testscript` the working directory structure will be as follows:

```
$out_base/
·-- test-hello/
    |-- command-name/
    |-- command-names/
    |-- stdin-names/
    |-- config/
    |   |-- hello.conf
    |   |-- custom-greet/
    |   ·-- default-greet/
    ·-- missing-name/
```

If all the tests succeed, then this working directory structure is automatically removed. In case of a failure, however, it is left behind in case you need to examine the output of the failed tests. It will be automatically cleaned on the subsequent run, before executing any tests.

The execution of tests happens in parallel. In the above case Testscript can start running all the top-level tests as well as the `config` group immediately. Inside `config`, once the setup command (`cat`) is completed, the two inner tests are executed in parallel as well. Refer to Model and Execution for details on the working directory structure and test execution.

# 2 Build System Integration

The integration of testscripts into buildfiles is done using the standard `build2` *target-prerequisite* mechanism. In this sense, a testscript is a prerequisite that describes how to test the target similar to how, for example, the `INSTALL` file describes how to install it. For example:

```
exe{hello}: testscript doc{INSTALL README}
```

By convention, the testscript file should be called either `testscript` if you only have one or have the `.testscript` extension, for example, `basics.testscript`. The `test` module registers the `testscript{}` target type to be used for testscript files. We don't have to use explicit target type for the `testscript` file. For example:

```
exe{hello}: testscript{basics advanced}
```

A testscript prerequisite can be specified for any target. For example, if our directory contains a bunch of executables that we want to test together, then it makes sense to specify the testscript prerequisite for the directory target:

```
./: testscript
```

Similarly, the same testscript can be used to test multiple targets. For example:

```
exe{hello}:      testscript{basics advanced}
exe{hello-lite}: testscript{basics}
```

During variable lookup if a variable is not found in one of the testscript scopes (see Model and Execution), then the search continues in the `buildfile` starting with the target-specific variables of the target being tested (e.g., `exe{hello}`; called *test target*), then target-specific variables of the testscript target (e.g., `testscript{basics}`; called *script target*), and then continuing with the scopes starting with the one containing the script target. As a result, a testscript can "see" all the existing buildfile variables plus we can use target-specific variables to pass additional, test-specific, information to testscrips. As an example, consider this testscript and buildfile pair:

```
# basics.testscript

if ($cxx.target.class == windows)
  test.arguments += $foo
end

if $windows
  test.arguments += $bar
end

# buildfile

exe{hello}: testscript{basics}

# All testscripts in this scope.
#
testscript{*}: windows = ($cxx.target.class == windows)

# All testscripts for target exe{hello}.
#
exe{hello}: bar = BAR

# Only basics.testscript.
#
testscript{basics}@./: foo = FOO
```

Additionally, by convention, a number of pre-defined `test.*` variables are used to pass commonly required information to testscripts, as described next.

Unless set manually as a test or script target-specific variable, the `test` variable is automatically set to the target path being tested. For example, given this `buildfile`:

```
exe{hello}: testscript
```

The value of `test` inside the testscript will be the absolute path to the `hello` executable.

If the `test` variable is set manually to a name of a target, then it is automatically converted to the target path. This can be useful when testing a program that is built in another subdirectory of a project (or even in another project, via import). For example, our `hello` may reside in the `hello/` subdirectory while we may want to keep the tests in `tests/`:

```
hello/
|-- hello/
|   `-- hello*
`-- tests/
    |-- buildfile
    `-- testscript
```

This is how we can implement `tests/buildfile` for this setup:

```
hello = ../hello/exe{hello}

./: $hello testscript
./: test = $hello

include ../hello/
```

The rest of the special `test.*` variables are `test.options`, `test.arguments`, `test.redirects`, and `test.cleanups`. You can use them to pass additional command line options, arguments, redirects, and cleanups to your test scripts. Together with `test` these variables form the *test target command* line which, for conciseness, is bound to the following aliases:

```
$* - $test $test.options $test.arguments $test.redirects $test.cleanups
$0 - $test
$N - (N-1)-th element in the {$test.options $test.arguments} array
```

Note that these aliases are read-only; if you need to modify any of these values from within testscripts, then you should use the original variable names, for example:

```
test.options += --foo

$* bar # Includes --foo.
```

Note also that these `test.*` variables only establish a convention. You could also put everything into, say `test.arguments`, and it will still work as expected.

The `test.redirects`, `test.cleanups`, and `$*` variables are of the special `cmdline` type, see Lexical Structure for details.

The special `test.*` variables make it fairly easy to arrange the testing of a single executable. What if we need to run multiple executables from a single testscript file? For example, we may have a pair of executables, such as `reader` and `writer`, that must be tested together. Or we may have a number of test executables that all require a common setup, for example, cryptographic key generation, which we would like not to repeating for each test. While it is possible to achieve this with target-specific variables similar to `test`, things will be less automatic. In particular, there will be no automatic translation of target names to paths and we will have to do it manually. For example:

```
# buildfile

./: exe{reader}: cxx{reader} ...
./: exe{writer}: cxx{writer} ...

./: testscript
{
  reader = exe{reader}
  writer = exe{writer}
}

# testscript

# Translate targets to paths.
#
reader = $path($reader)
writer = $path($writer)

: pipe
:
$writer | $reader

: file
:
$writer output;
$reader output
```

Strictly speaking, for local executables, there is no need to pass the target names from `buildfile` to `testscript` and instead we could just list them literally in `testscript`. In particular, this could be an attractive approach if we have a large number of such executables. For example:

```
# testscript

$path(exe{test1}) : test1
$path(exe{test2}) : test2
$path(exe{test3}) : test3
...
```

Another pre-defined variable is `test.target`. It is used to specify the test target platform when cross-testing (for example, when running Windows test on Linux under Wine). Normally, you would set it in your `build/root.build` to the cross-compilation target of your toolchain, for example:

```
# root.build
#

using cxx                 # Load the C++ module (sets sets cxx.target).
test.target = $cxx.target # Set test target to the C++ compiler target.
```

If this variable is not set explicitly, then it defaults to `build.host` (which is the platform on which the build system is running) and only native testing will be supported.

All the testscripts for a particular test target are executed in a subdirectory of `out_base` (or, more precisely, in subdirectories of this subdirectory; see Model and Execution). If the test target is a directory, then the subdirectory is called `test`. Otherwise, it is the name of the

target prefixed with`test-`. For example:

```
./:         testscript{foo}  # $out_base/test/
exe{hello}: testscript{bar}  # $out_base/test-hello/
```

# 3 Model and Execution

A testscript file is a set of nested scopes. A scope is either a group scope or a test scope. Group scopes can contain nested group and test scopes. Test scopes can only contain test commands.

Group scopes are used to organize related tests with shared variables as well as setup and teardown commands. Explicit test scopes are normally used for better visual separation of complex tests.

The top level scope is always an implicit group scope corresponding to the entire script file. If there is no explicit scope for a test, one is established implicitly. As a result, a testscript file always starts with a group scope which then contains other group scopes and/or test scopes, recursively.

A scope (both group and test) has an *id*. If not specified explicitly (as part of the description), it is derived automatically from the group/test location in the testscript file (see Description for details). The id of the implicit outermost scope is the script file name without the `.testscript` extension, except if the file name is `testscript`, in which case the id is empty.

Based on the ids each nested group and test has an *id path* that uniquely identifies it. It starts with the id of the implied outermost group (unless empty), may include a number of intermediate group ids, and ends with the final test or group id. The ids in the path are separated with a forward slash (`/`). Note that this also happens to be the relative filesystem path to the temporary directory where the test is executed (as described below). Inside a scope its id path is available via the special `$@` variable (read-only).

As an example, consider the following testscript file which we assume is called `basics.testscript`:

```
test0: test0

: group
{
  test1

  : test2
  {
    test2a;
    test2b
  }
}
```

Below is its version annotated with the id paths that also shows all the implicit scopes:

```
# basics
{
  # basics/test0
  {
    test0
  }

  # basics/group
  {
    # basics/group/5
    {
      test1
    }

    # basics/group/test2
    {
      test2a;
      test2b
    }
  }
}
```

A scope establishes a nested variable context. A variable set within a scope will only have effect until the end of this scope. Variable lookup is performed starting from the scope where the variable is referenced (expanded), continuing with the outer testscript scopes, and then continuing in the buildfile as described in Build System Integration.

A scope also establishes a cleanup context. All cleanups (Cleanup) registered in a scope are performed at the end of that scope's execution in the reverse order of their registration.

Prior to executing a scope, a nested temporary directory is created with the scope id as its name. This directory then becomes the scope's working directory. After executing the scope (and after performing cleanups) this temporary directory is automatically removed provided that it is empty. If it is not empty, then the test is considered to have failed (unexpected output). Inside a scope its working directory is available via the special `$~` variable (read-only).

As an example, consider the following version of `basics.testscript`. We also assume that its test target is a directory (so the target test directory is `$out_base/test/`).

```
: group
{
  foo = FOO
  bar = BAR

  +setup &out-setup

  : test1
  {
    bar = BAZ
    test1 $foo $bar
  }
```

```
  test2 $bar: test2
}

test3 $foo &out-test
```

Below is its annotated version:

```
{                           # $~ = $out_base/test/basics/
  {                         # $~ = .../test/basics/group/
    foo = FOO
    bar = BAR

    +setup &out-setup

    {                       # $~ = .../basics/group/test1/
      bar = BAZ
      test1 $foo $bar       # test1 FOO BAZ
    }

    {                       # $~ = .../basics/group/test2/
      test2 $bar            # test2 BAR
    }
  }                         # Remove out-setup.

  {                         # $~ = .../test/basics/17/
    test3 $foo &out-test    # test3
  }                         # Remove out-test.
}
```

A test should normally create files or directories, if any, in its working directory to ensure test isolation. A test can, however, access (but normally should not modify) files created by an outer group's setup commands. Because of this nested directory structure this can be done using `../`-based relative paths, for example:

```
{
  +setup >=test.conf

  test1 ../test.conf
  test2 ../test.conf
}
```

Alternatively, we can use an absolute path:

```
{
  conf = $~/test.conf
  +setup >=$conf

  test1 $conf
  test2 $conf
}
```

Inside the scope working directory filesystem names that start with `stdin`, `stdout`, and `stderr` are reserved.

To execute a test scope its commands (including variable assignments) are executed sequentially and in the order specified. If any of the commands fails, no further commands are executed and the test is considered to have failed.

Executing a group scope starts with performing its setup commands (including variable assignments) sequentially and in the order specified. If any of them fail, the group execution is terminated and the group is considered to have failed.

After completing the setup, inner scopes (both group and test) are executed. Because scopes are isolated and tests are assumed not to depend on each other, the execution of inner scopes can be performed in parallel.

After completing the execution of the inner scopes, if all of them succeeded, the teardown commands are executed sequentially and in the order specified. Again, if any of them fail, the group execution is terminated and the group is considered to have failed.

Currently, the only way to run several executables serially is to place them into a single compound test. See Test for details.

As an example, consider the following version of `basics.testscript`:

```
test0

: group
{
  +setup1
  +setup2

  test1
  test2
  test3

  -teardown2
  -teardown1
}
```

At the top level, both `test0` and `group` can start executing in parallel. Inside `group`, first the two setup commands are executed sequentially. Once the setup is completed, `test1`, `test2`, `test3` can all be executed in parallel (along with `test0` which may still be running). Once the three inner tests complete successfully, the `group`'s teardown commands are executed sequentially. At the top level, the script is completed only when both `test0` and `group` complete.

The following annotated version illustrates a possible thread scheduling for this example:

```
{               # thread 1

  test0         # thread 2

  : group       # thread 1
  {
    +setup1     # thread 1
    +setup2     # thread 1

    test1       # thread 3
    test2       # thread 4
    test3       # thread 1

                # thread 1 (wait for 3 & 4)
```

```
    -teardown2  # thread 1
    -teardown1  # thread 1
  }
              # thread 1 (wait for 2)
}
```

A testscript would normally contain multiple tests and sometimes it is desirable to only execute a specific test or a group of tests. For example, you may be debugging a failing test and would like to re-run it. As an example, consider the following testscript file called `basics.testscript`:

```
$* foo : foo

: fox
{
  $* fox bar : bar
  $* fox baz : baz
}
```

The id paths for these three test will then be:

```
basics/foo
basics/fox/bar
basics/fox/baz
```

To only run individual tests, test groups, or testscript files we can specify their id paths in the `config.test` variable, for example:

```
$ b test config.test=basics                      # All in basics.testscript
$ b test config.test=basics/fox                  # All in fox
$ b test config.test=basics/foo                  # Only foo
$ b test config.test="basics/foo basics/fox/bar" # Only foo and bar
```

The test commands (`$0`, `$*`) can be executed via a *runner program* by specifying the `config.test.runner` variable (see `test` module for details). For example:

```
$ b test config.test.runner="valgrind -q"
```

The script working directory may exist before the execution (for example, because of a failed previous run) or it may be desirable not to clean it up after the execution (for example, to examine test setup, output, etc). Before the execution the default behavior is to warn and then automatically remove the working directory if it exists. After the execution the default behavior is to perform all the cleanups and teardowns and then remove the working directory if it is not empty. This default behavior can, however, be overridden with the `config.test.output` variable.

The `config.test.output` variable contains a pair of values with the first signifying the *before* behavior and the second – *after*. The valid *before* values are `fail` (fail if the directory exists), `warn` (warn if the directory exists then remove), `clean` (silently remove the existing directory). The valid *after* values are `clean` (remove the directory if it is not empty) and `keep` (do not run cleanups and teardowns and do not remove the working directory). The default behavior is thus equivalent to specifying the `warn@clean` pair.

If only a single value is specified in `config.test.output` then it is assumed to be the *after* value and the *before* value is assumed to be `clean`. In other words:

```
$ b test config.test.output=clean # config.test.output=clean@clean
$ b test config.test.output=keep  # config.test.output=clean@keep
```

Note also that selecting the `keep` behavior may result in some test failures (due to unexpected output) to go undetected.

# 4 Lexical Structure

At the lexical level, testscripts are UTF-8 encoded text restricted to the Unicode graphic characters, tabs (`\t`), carriage returns (`\r`), and line feeds (`\n`).

Testscript is a line-oriented language with a context-dependent lexical structure. It "borrows" several building blocks (variable expansion, function calls, and evaluation contexts; collectively called *expansions* from now on) from the Buildfile language. In a sense, testscripts are specialized (for testing) continuations of buildfiles.

Except in here-document fragments, leading whitespaces and blank lines are ignored except for the line/column counting. A non-empty testscript must end with a newline.

Except in single-quoted strings and single-quoted here-document fragments, the backslash (`\`) character followed by a newline signals the line continuation. Both this character and the newline are removed (note: not replaced with a whitespace) and the following line is read as if it was part of the first line. Note that `'\'` followed by EOF is invalid. For example:

```
$* foo | \
$* bar
```

Except in quoted strings and here-document fragments, an unquoted and unescaped `'#'` character starts a comment; everything from this character until the end of the line is ignored. For example:

```
# Setup foo.
$* foo

$* bar # Setup bar.
```

There is no line continuation support in comments; the trailing `'\'` is ignored except in one case: if the comment is just `'#\'` followed by the newline, then it starts a multi-line comment that spans until the closing `'#\'` is encountered. For example:

```
#\
$* foo
$* bar
#\

$* foo #\
$* bar
$* baz #\
```

Similar to Buildfile, the Testscript language supports two types of quoting: single (`'`) and double (`"`). Both can span multiple lines.

The single-quoted strings and single-quoted here-document fragments do not recognize any expansions or escape sequences (not even for the single quote itself or line continuations) with all the characters taken literally until the closing single quote or here-document end marker is encountered.

The double-quoted strings and double-quoted here-document fragments recognize expansions and escape sequences (including line continuations). For example:

```
foo = FOO

# 'FOO true'
#
bar = "$foo ($foo == FOO)"

# 'FOO bool'
#
$* <<"EOI"
$foo $type($foo == FOO)
EOI
```

Characters that have special syntactic meaning (for example `'$'`) can be escaped with a backslash (`\`) to preserve their literal meaning (to specify literal backslash you need to escape it as well). For example:

```
foo = \$foo\\bar # '$foo\bar'
```

Note that quoting could often be a more readable way to achieve the same result, for example:

```
foo = '$foo\bar'
```

Inside double-quoted strings only the `"\$(` character set needs to be escaped. Inside double-quoted here-document fragments – only `\$(` (since in here-documents quotes are taken literally).

The lexical structure of a line depends on its type. The line type could be dictated by the preceding construct, as is the case for here-document fragments. Otherwise, the line type is determined by examining the leading character and, if that fails to determine the line type, leading tokens, as described next.

A character is said to be *unquoted* and *unescaped* if it is not escaped and is not part of a quoted string. A token is said to be unquoted and unescaped if all its characters are unquoted and unescaped.

The following characters determine the line type if they appear unquoted and unescaped at the beginning of the line:

```
':'  - description line
'.'  - directive line
'{'  - scope start
'}'  - scope end
'+'  - setup command line
'-'  - teardown command line
```

If the line doesn't start with any of these characters then the first token of the line is examined in the `first_token` mode (see below). If the first token is an unquoted word, then the second token of the line is examined in the `second_token` mode (see below). If it is a variable assignment (either +=, =+, or =), then the line type is a variable line. Otherwise, it is a test command line. Note that variables with computed names can only be set using the `set` pseudo-builtin.

The Testscript language defines the following distinct lexing modes (or contexts):

**`command_line`**
: Whitespaces are token separators. The following characters and character sequences (read vertically, for example, ==, != below) are recognized as tokens:

```
:;=!|&<>$(#
  ==
```

**`first_token`**
: Like `command_line` but recognizes variable assignments as separators.

**`second_token`**
: Like `command_line` but recognizes variable assignments as tokens.

**`command_expansion`**
: Subset of `command_line` used for re-lexing expansions (described below). Only the `|&<>` characters are recognized as tokens. Note that whitespaces are not separators in this mode.

**`variable_line`**
: Similar to the Buildfile **`value`** mode. The `;$([]` characters are recognized as tokens.

**`description_line`**
: Like a single-quoted string.

**`here_line_single`**
: Like a single-quoted string except it treats newlines as separators and quotes as literals.

**`here_line_double`**
: Like a double-quoted string except it treats newlines as separators and quotes as literals. The `$(` characters are recognized as tokens.

Besides having a varying lexical structure, parsing some line types involves performing expansions (variable expansions, function calls, and evaluation contexts). The following table summarizes the mapping of line types to lexing modes and indicates whether they are parsed with expansions:

```
variable line                 variable_line      expansions
directive line                command_line       expansions
description line              description_line

test command line             command_line       expansions
setup command line            command_line       expansions
teardown command line         command_line       expansions

here-document single-quoted   here_line_single
here-document double-quoted   here_line_double   expansions
```

Finally, unquoted expansions in command lines (test, setup, and teardown) of the special `cmdline` type are re-lexed in the `command_expansion` mode in order to recognize command line syntax tokens (redirects, pipes, etc). To illustrate this mechanism, consider the following example of a "canned" command line:

```
cmd = [cmdline] echo >-
$cmd foo
```

The test command line token sequence will be `$`, `cmd`, `foo`. After the expansion we have `echo`, `>-`, `foo`, however, the second element (`>-`) is not (yet) recognized as a redirect. To recognize it, the result of the expansion is re-lexed.

Note that besides the few command line syntax characters, re-lexing will also "consume" quotes and escapes, for example:

```
cmd = [cmdline] echo "'foo'"  # echo 'foo'
$cmd                          # echo foo
```

To preserve quotes in this context we need to escape them:

```
cmd = [cmdline] echo "\\'foo\\'"  # echo \'foo\'
$cmd                              # echo 'foo'
```

To minimize unhelpful consumption of escape sequences (for example, in Windows paths), re-lexing only performs the *effective escaping* for the `'"\` characters. All other escape sequences are passed through uninterpreted. Note that this means there is no way to escape command line syntax characters in canned commands. The recommendation is to use quoting except for passing literal quotes, for example:

```
cmd = [cmdline] echo \'&foo\'  # echo '&foo'
$cmd                           # echo &foo
```

To make sure that a string is passed as is through both expansions use the *doubled single-quoting* idiom, for example:

```
filter = [cmdline] sed -e \''s/foo (bar|baz)/$&/'\'
$* <<EOI | $filter >>EOO
...
EOI
...
EOO
```

# 5 Syntax and Semantics

## 5.1 Notation

The formal grammar of the Testscript language is specified using an EBNF-like notation with the following elements:

```
foo: ...   - production rule
foo        - non-terminal
<foo>      - terminal
'foo'      - literal
foo*       - zero or more multiplier
foo+       - one or more multiplier
foo?       - zero or one multiplier
foo bar    - concatenation (foo then bar)
foo | bar  - alternation   (foo or bar)
(foo bar)  - grouping
{foo bar}  - grouping in any order (foo then bar or bar then foo)
foo\
bar        - line continuation
# foo      - comment
```

A rule's right-hand-sides that start on a new line describe the line-level syntax and ones that start on the same line describes the syntax inside the line. If a rule contains multiple lines, then each line matches a separate line in the input.

If a multiplier appears in front of a line then it specifies the number of repetitions of the entire line. For example, from the following three rules, the first describes a single line of multiple literals, the second – multiple lines of a single literal, and the third – multiple lines of multiple literals.

```
# foofoofoo
#
text-line: 'foo'+

# foo
# foo
# foo
#
text-lines:
  +'foo'

# foo
# foofoo
# foofoofoo
#
text-lines:
  +('foo'+)
```

A newline in the grammar matches any standard newline separator sequence (CR/LF combinations). An unquoted space in the grammar matches zero or more non-newline whitespaces (spaces and tabs). A quoted space matches exactly one non-newline whitespace. Note also that in some cases components within lines may not be whitespace-separated in which case they will be written without any spaces between them, for example:

```
foo: 'foo' ';'    # 'foo;' or 'foo ;' or 'foo      ;'
bar: 'bar'';'     # 'bar;'
baz: 'baz'' '+';' # 'baz ;' or 'baz      ;'

fox: bar''bar     # 'bar;bar;'
```

You may also notice that several production rules below end with `-line` while potentially spanning several physical lines. The `-line` suffix here signifies a *logical line*, for example, a command line plus its here-document fragments.

# 5.2 Grammar

The complete grammar of the Testscript language is presented next with the following sections discussing the semantics of each production rule.

```
script:
  scope-body

scope-body:
  *setup
  *(scope|directive|test)
  *tdown

scope:
  ?description
  scope-block|scope-if

scope-block:
  '{'
  scope-body
  '}'

scope-if:
  ('if'|'if!') command-line
  scope-block
  *scope-elif
  ?scope-else

scope-elif:
  ('elif'|'elif!') command-line
  scope-block

scope-else:
  'else'
  scope-block

directive:
  '.' include

include: 'include' (' '+'--once')*(' '+<path>)*

setup:
  variable-like|setup-line

tdown:
  variable-like|tdown-line

setup-line: '+' command-like
tdown-line: '-' command-like
```

```
test:
  ?description
  +(variable-line|command-like)

variable-like:
  variable-line|variable-flow

variable-line:
  <variable-name> ('='|'+='|'=+') value-attributes? <value> ';'?

value-attributes: '[' <key-value-pairs> ']'

variable-flow:
  variable-if|variable-for|variable-while

variable-if:
  ('if'|'if!') command-line
  variable-flow-body
  *variable-elif
  ?variable-else
  'end' ';'?

variable-elif:
  ('elif'|'elif!') command-line
  variable-flow-body

variable-else:
  'else'
  variable-flow-body

variable-flow-body:
  *variable-like

variable-for:
  variable-for-args|variable-for-stream

variable-for-args:
  'for' <variable-name> element-attributes? ':' \
  value-attributes? <value>
  variable-flow-body
  'end' ';'?

element-attributes: value-attributes

variable-for-stream:
  (command-pipe '|')? \
  'for' (<opt>|stdin)* <variable-name> element-attributes? (stdin)*
  variable-flow-body
  'end' ';'?

variable-while:
  'while' command-line
  variable-flow-body
  'end' ';'?

command-like:
  command-line|command-flow

command-line: command-expr (';'|(':' <text>))?
  *here-document
```

```
command-expr: command-pipe (('||'|'&&') command-pipe)*

command-pipe: command ('|' command)*

command: <path>(' '+(<arg>|redirect|cleanup))* command-exit?

command-exit: ('=='|'!=') <exit-status>

command-flow:
  command-if|command-for|command-while

command-if:
  ('if'|'if!') command-line
  command-flow-body
  *command-elif
  ?command-else
  'end' (';'|(':' <text>))?

command-elif:
  ('elif'|'elif!') command-line
  command-flow-body

command-else:
  'else'
  command-flow-body

command-flow-body:
  *(variable-line|command-like)

command-for:
  command-for-args|command-for-stream

command-for-args:
  'for' <variable-name> element-attributes? ':' \
  value-attributes? <value>
  command-flow-body
  'end' (';'|(':' <text>))?

command-for-stream:
  (command-pipe '|')? \
  'for' (<opt>|stdin)* <variable-name> element-attributes? (stdin)*
  command-flow-body
  'end' (';'|(':' <text>))?

command-while:
  'while' command-line
  command-flow-body
  'end' (';'|(':' <text>))?

redirect: stdin|stdout|stderr

stdin:  '0'?(in-redirect)
stdout: '1'?(out-redirect)
stderr: '2'(out-redirect)

in-redirect:  '<-'|\
              '<|'|\
              ('<<<'|'<=') <file>|\
              ('<<'|'<<=')<here-end>|\
              ('<'|'<<<=')<text>

out-redirect: '>-'|\
```

```
              '>|'|\
              '>!'|\
              '>=' <file>|\
              '>+' <file>|\
              '>&' ('1'|'2')|\
              ('>>>'|'>?') <file>|\
              ('>>'|'>>?'){':'?'/'?}'~'? <here-end>|\
              ('>'|'>>>?'){':'?'/'?}'~'? <text>

here-document:
  *<text>
  <here-end>

cleanup: ('&'|'&?'|'&!') (<file>|<dir>)

description:
  +(':' <text>)
```

Note that the only purpose of having separate (from the command flow control constructs) variable-only flow control constructs is to remove the error-prone requirement of having to specify + and − prefixes in group setup/teardown.

# 5.3 Script

```
script:
  scope-body
```

A testscript file is an implicit group scope (see Model and Execution for details).

# 5.4 Scope

```
scope-body:
  *setup
  *(scope|directive|test)
  *tdown

scope:
  ?description
  scope-block|scope-if

scope-block:
  '{'
  scope-body
  '}'
```

A scope is either a test group scope or an explicit test scope. An explicit scope is a test scope if it contains a single test, only variable assignments in setup commands, no teardown commands, and only the scope having the description, if any. Otherwise, it is a group scope. If there is no explicit scope for a test, one is established implicitly.

# 5.5 Scope-If

```
scope-if:
  ('if'|'if!') command-line
  scope-block
  *scope-elif
  ?scope-else

scope-elif:
  ('elif'|'elif!') command-line
  scope-block

scope-else:
  'else'
  scope-block
```

A scope, either test or group, can be executed conditionally. The condition `command-line` is executed in the context of the outer scope. Note that all the scopes in an `if-else` chain are alternative implementations of the same group/test (thus the single description). If at least one of them is a group scope, then all the others are treated as groups as well.

# 5.6 Directive

```
directive:
  '.' include
```

A line that starts with `.` is a Testscript directive. Note that directives are evaluated during parsing, before any command is executed or (testscript) variable is assigned. You can, however, use variables assigned in the buildfile. For example:

```
.include common-$(cxx.target.class).testscript
```

## 5.6.1 Include

```
include: 'include' (' '+'--once')*(' '+<path>)*
```

While in the grammar the `include` directive is shown to only appear interleaving with scopes and tests, it can be used anywhere in the scope body. It can also contain several parts of a scope, for example, setup and test lines.

The `--once` option signals that files that have already been included in this scope should not be included again. The implementation is not required to handle links when determining if two paths are to the same file. Relative paths are assumed to be relative to the including testscript file.

# 5.7 Setup and Teardown

```
setup:
  variable-like|setup-line

tdown:
```

```
  variable-like|tdown-line

setup-line: '+' command-like
tdown-line: '-' command-like
```

Note that variable assignments (including `variable-flow`) do not use the `'+'` and `'-'` prefixes. A standalone (not part of a test) variable assignment is automatically treated as a setup if no tests have yet been encountered in this scope and as a teardown otherwise.

# 5.8 Test

```
test:
  ?description
  +(variable-line|command-like)
```

A test that contains multiple lines is called *compound*. In this case each (logical) line except the last must end with a semicolon to signal the test continuation. For example:

```
conf = test.conf;
cat <'verbose = true' >=$conf;
test1 $conf
```

As discussed in Model and Execution, tests are executed in parallel. Currently, the only way to run several executables serially is to place them into a single compound test.

# 5.9 Variable

```
variable-like:
  variable-line|variable-flow

variable-line:
  <variable-name> ('='|'+='|'=+') value-attributes? <value> ';'?

value-attributes: '[' <key-value-pairs> ']'
```

The Testscript variable assignment semantics is equivalent to Buildfile except that no `{}`-based name-generation is performed. For example:

```
args = [strings] foo   bar 'fox   baz'
echo $args # foo bar fox   baz
```

The value can only be followed by `;` inside a test to signal the test continuation.

# 5.10 Variable-If

```
variable-if:
  ('if'|'if!') command-line
  variable-flow-body
  *variable-elif
  ?variable-else
  'end' ';'?

variable-elif:
  ('elif'|'elif!') command-line
  variable-flow-body
```

```
variable-else:
  'else'
  variable-flow-body

variable-flow-body:
  *variable-like
```

A group of variables can be set conditionally. The condition `command-line` semantics is the same as in `scope-if`. For example:

```
if ($cxx.target.class == 'windows')
  slash = \\
  case = false
else
  slash = /
  case = true
end
```

When conditionally setting a single variable, using the evaluation context with a ternary operator is often more concise:

```
slash = ($cxx.target.class == 'windows' ? \\ : /)
```

## 5.11 Variable-For

```
variable-for:
  variable-for-args|variable-for-stream

variable-for-args:
  'for' <variable-name> element-attributes? ':' \
  value-attributes? <value>
  variable-flow-body
  'end' ';'?

variable-for-stream:
  (command-pipe '|')? \
  'for' (<opt>|stdin)* <variable-name> element-attributes? (stdin)*
  variable-flow-body
  'end' ';'?

variable-flow-body:
  *variable-like
```

A group of variables can be set in a loop while iterating over elements of a list. The iteration semantics is the same as in `command-for`. For example:

```
uvalues =
for v: $values
  uvalues += $string.ucase($v)
end
```

Another example:

```
uvalues =
cat values.txt | for -n v
  uvalues += $string.ucase($v)
end
```

Or using the `stdin` redirect:

```
uvalues =
for -n v <=values.txt
  uvalues += $string.ucase($v)
end
```

# 5.12 Variable-While

```
variable-while:
  'while' command-line
  variable-flow-body
  'end' ';'?

variable-flow-body:
  *variable-like
```

A group of variables can be set in a loop while the condition evaluates to `true`. The condition `command-line` semantics is the same as in `scope-if`. For example:

```
uvalues =
i = [uint64] 0
n = $size($values)
while ($i != $n)
  uvalues += $string.ucase($values[$i])
  i += 1
end
```

# 5.13 Command

```
command-like:
  command-line|command-flow

command-line: command-expr (';'|(':' <text>))?
  *here-document

command-expr: command-pipe (('||'|'&&') command-pipe)*

command-pipe: command ('|' command)*

command: <path>(' '+(<arg>|redirect|cleanup))* command-exit?

command-exit: ('=='|'!=') <exit-status>
```

A command line is a command expression. If it appears directly (as opposed to inside `command-flow`) in a test, then it can be followed by `;` to signal the test continuation or by `:` and the trailing description.

A command expression can combine several command pipes with logical AND and OR operators. Note that the evaluation order is always from left to right (left-associative), both operators have the same precedence, and are short-circuiting. Note, however, that short-circuiting

does not apply to expansions (variable, function calls, evaluation contexts). The logical result of a command expression is the result of the last command pipe executed.

A command pipe can combine several commands with a pipe (`stdout` of the left-hand-side command is connected to `stdin` of the right-hand-side). The logical result of a command pipe is the logical AND of all its commands.

A command begins with a command path followed by options/arguments, redirects, and cleanups, all optional and in any order.

A command may specify an exit code check. If executing a command results in an abnormal process termination, then the whole outer construct (e.g., test, setup/teardown, etc) summarily fails. Otherwise (that is, in case of a normal termination), the exit code is checked. If omitted, then the test is expected to succeed (0 exit code). The logical result of executing a command is therefore a boolean value which is used in the higher-level constructs (pipe and expression).

# 5.14 Command-If

```
command-if:
  ('if'|'if!') command-line
  command-flow-body
  *command-elif
  ?command-else
  'end' (';'|(':' <text>))?

command-elif:
  ('elif'|'elif!') command-line
  command-flow-body

command-else:
  'else'
  command-flow-body

command-flow-body:
  *(variable-line|command-like)
```

A group of commands can be executed conditionally. The condition `command-line` semantics is the same as in `scope-if`. Note that in a compound test, commands inside `command-if` must not end with `;`. Rather, `;` may follow `end`. For example:

```
if ($cxx.target.class == 'windows')
  foo = windows
  setup1
  setup2
else
  foo = posix
end;
test1 $foo
```

# 5.15 Command-For

```
command-for:
  command-for-args|command-for-stream

command-for-args:
  'for' <variable-name> element-attributes? ':' \
  value-attributes? <value>
  command-flow-body
  'end' (';'|(':' <text>))?

command-for-stream:
  (command-pipe '|')? \
  'for' (<opt>|stdin)* <variable-name> element-attributes? (stdin)*
  command-flow-body
  'end' (';'|(':' <text>))?

command-flow-body:
  *(variable-line|command-like)
```

A group of commands can be executed in a loop while iterating over elements of a list and setting the specified variable (called *loop variable*) to the corresponding element on each iteration. At the end of the iteration the loop variable contains the value of the last element, if any. Note that in a compound test, commands inside `command-for` must not end with `;`. Rather, `;` may follow `end`.

The `for` loop has two forms: In the first form the list is specified as arguments. Similar to the `for` loop in the Buildfile language, it can contain variable expansions, function calls, evaluation contexts, and/or literal values. For example:

```
for v: $values
  test1 $v
end;
test2
```

In the second form the list is read from the `stdin` input. The input data is split into elements either at whitespaces (default) or newlines, which can be controlled with the `-n|--newline` and `-w|--whitespace` options. Overall, this form supports the same set of options as the `set` pseudo-builtin. For example:

```
cat values.txt | for -n v
  test1 $v
end
```

Or using the `stdin` redirect:

```
for -n v <=values.txt
  test1 $v
end
```

Both forms can include value attributes enclosed in `[]` to be applied to each element, again similar to the `set` pseudo-builtin.

# 5.16 Command-While

```
command-while:
  'while' command-line
  command-flow-body
  'end' (';'|(':' <text>))?

command-flow-body:
  *(variable-line|command-like)
```

A group of commands can be executed in a loop while a condition evaluates to `true`. The condition `command-line` semantics is the same as in `scope-if`. Note that in a compound test, commands inside `command-while` must not end with `;`. Rather, `;` may follow `end`. For example:

```
i = [uint64] 0;
n = $size($values);
while ($i != $n)
  test1 ($values[$i])
  i += 1
end;
test2
```

Another example:

```
while test -f $file
  test1 $file
end
```

# 5.17 Redirect

```
redirect: stdin|stdout|stderr

stdin:  '0'?(in-redirect)
stdout: '1'?(out-redirect)
stderr: '2'(out-redirect)
```

In redirects the file descriptors must not be separated from the redirect operators with whitespaces. And if leading text is not separated from the redirect operators, then it is expected to be the file descriptor. As an example, the first command below has `2` as an argument (and therefore redirects `stdout`, not `stderr`). While the second is invalid since `a1` is not a valid file descriptor.

```
$* 2 >-
$* a1>-
```

# 5.18 Input Redirect

```
in-redirect:  '<-'|\
              '<|'|\
              ('<<<'|'<=') <file>|\
              ('<<'|'<<='){':'?'/'?} <here-end>|\
              ('<'|'<<<='){':'?'/'?} <text>
```

The `stdin` data can come from a pipe, here-string (<), here-document (<<), a file (<<<), or `/dev/null`-equivalent (<-). Specifying both a pipe and a redirect is an error. If no pipe or `stdin` redirect is specified and the test tries to read from `stdin`, it is considered to have failed (unexpected input). However, whether this is detected and diagnosed is implementation-defined. To allow reading from the default `stdin` (for instance, if the test is really an example), the <| redirect is used.

The <=, <<=, and <<<= redirects are a stable syntax across various `build2` scripting language flavors (Testscript, Buildscript, etc). While the <, <<, and <<< redirects are their Testscript aliases with the mapping chosen to be more convenient for this flavor of the scripting language. This mapping is as follows:

```
<    <<<=  here-string
<<   <<=   here-document
<<<  <=    file
```

Here-string and here-document redirects may specify the following redirect modifiers:

The `:` modifier is used to suppress the otherwise automatically-added terminating newline.

The `/` modifier causes all the forward slashes in the here-string or here-document to be translated to the directory separator of the test target platform (as indicated by `test.target`).

A here-document redirect must be specified *literally* on the command line. Specifically, it must not be the result of an expansion (which rarely makes sense anyway since the following here-document fragment itself cannot be the result of an expansion either). See Here Document for details.

# 5.19 Output Redirect

```
out-redirect: '>-'|\
              '>|'|\
              '>!'|\
              '>=' <file>|\
              '>+' <file>|\
              '>&' ('1'|'2')|\
              ('>>>'|'>?') <file>|\
              ('>>'|'>>?'){':'?'/'?}'~'? <here-end>|\
              ('>'|'>>>?'){':'?'/'?}'~'? <text>
```

The `stdout` and `stderr` data can go to a pipe (`stdout` only), file (>= to overwrite and >+ to append), or `/dev/null`-equivalent (>-). It can also be compared to a here-string (>), a here-document (>>), or a file contents (>>>). For `stdout` specifying both a pipe and a redirect is an error. A test that tries to write to an un-redirected stream (either `stdout` or `stderr`) is considered to have failed (unexpected output). To allow writing to the default `stdout` or `stderr` (for instance, if the test is really an example), the >| redirect is used.

The >?, >>?, and >>>? redirects are a stable syntax across various `build2` scripting language flavors (Testscript, Buildscript, etc). While the >, >>, and >>> redirects are their Testscript aliases with the mapping chosen to be more convenient for this flavor of the scripting language. This mapping is as follows:

```
>    >>>?  here-string comparison
>>   >>?   here-document comparison
>>>  >?    file contents comparison
```

The `>!` redirect acts like `>-` if the build system verbosity level is below 2 and as `>|` otherwise. It is normally used to ignore diagnostics (as opposed to data) during normal operation but to still be able to examine it, for example, when debugging a failing test.

It is also possible to merge `stderr` to `stdout` or vice versa with a merge redirect (`>&`). In this case the left-hand-side descriptor (implied or explicit) must not be the same as the right-hand-side. Having both merge redirects at the same time is an error.

The `:` and `/` redirect modifiers have the same semantics as in the input redirects. The `~` modifier is used to indicate that the following here-string/here-document is a regular expression (see Regex) rather than a literal. Note that if present, it must be specified last.

Similar to the input redirects, an output here-document redirect must be specified literally on the command line. See Here Document for details.

## 5.20 Here-Document

```
here-document:
  *<text>
  <here-end>
```

A here-document can be used to supply data to `stdin` or to compare output to the expected result for `stdout` and `stderr`. The order of here-document fragments must match the order of redirects, for example:

```
: select-no-table-error
$* --interactive >>EOO <<EOI 2>>EOE
enter query:
EOO
SELECT * FROM no_such_table
EOI
error: no such table 'no_such_table'
EOE
```

Two or more here-document redirects can use the same end marker. In this case all the redirects must have the same modifiers, if any. Only the here-document fragment corresponding to the first occurrence of the end marker must be present (called *shared* here-document) with the subsequent redirects reusing the same data. This mechanism is primarily useful for round-trip testing, for example:

```
: xml-round-trip
$* <<EOD >>EOD
<hello>Hello, World!</hello>
EOD
```

Here-strings can be single-quoted literals or double-quoted with expansion. This semantics is extended to here-documents as follows: If the end marker on the command line is single-quoted, then the here-document lines are parsed as if they were single-quoted except

that the single quote itself is not treated as special. In this mode there are no expansions, escape sequences, not even line continuations – each line is taken literally.

If the end marker on the command line is double-quoted, then the here-document lines are parsed as if they were double-quoted except that the double quote itself is not treated as special. In this mode we can use variable expansions, function calls, and evaluation contexts. However, we have to escape the `$ ( \` character set.

If the end marker is not quoted then it is treated as if it were single-quoted. Note also that quoted end markers must be quoted entirely, that is, from the beginning and until the end and without any interruptions.

Here-document fragments can be indented. The leading whitespaces of the end marker line (called *strip prefix*) determine the indentation. Every other line in the here-document should start with this prefix which is then automatically stripped. The only exception is a blank line. For example, the following two testscripts are equivalent:

```
{
  $* <<EOI
  foo
    bar
  EOI
}

{
  $* <<EOI
foo
  bar
EOI
}
```

Note, however, that the leading whitespace stripping does not apply to line continuations.

# 5.21 Output Regex

Instead of literal text the expected result in output here-strings and here-documents can be specified as ECMAScript regular expressions (more specifically, ECMA-262-based C++11 regular expressions). To signal the use of regular expressions the redirect must end with the `~` modifier, for example:

```
$* >~'/fo+/' 2>>~/EOE/
/ba+r/
baz
EOE
```

The regular expression used for output matching is *two-level*. At the outer level the expression is over lines with each line treated as a single character. We will refer to this outer expression as *line-regex* and to its characters as *line-char*.

A line-char can be a literal line (like `baz` in the example above) in which case it will only be equal to an identical line in the output. Alternatively, a line-char can be an inner level regex (like `ba+r` above) in which case it will be equal to any line in the output that matches this regex. Where not clear from context we will refer to this inner expression as *char-regex* and its characters as *char*.

A line is treated as literal unless it starts with the *regex introducer character* (`/` in the above example). In contrast, the line-regex is always in effect (in a sense, the `~` modifier is its introducer). Note that the here-string regex naturally (since there is only one line) must start with an introducer.

A char-regex line that starts with an introducer must also end with one optionally followed by *match flags*, for example:

```
$* >>~/EOO/
/ba+r/i
/ba+z/i
EOO
```

The following match flags are recognized:

`i`
: Perform case-insensitive match.

`d`
: Invert the dot character (`.`) escaping. With this flag unescaped dots are treated as literal characters while the escaped ones (`\.`) – as matching any character. Note that dots specified within character classes (`[.]`) are not affected.

Any character can act as a regex introducer. For here-strings it is the first character in the string. For here-documents the introducer is specified as part of the end marker. In this case the first character is the introducer, everything after that and until the second occurrence of the introducer is the actual end marker, and everything after that are global match flags. Global match flags apply to every char-regex (but not literal lines or the line-regex itself) in this here-document. Note that there is no way to escape the introducer character inside the regex.

As an example, here is a shorter version of the previous example that also uses a different introducer character.

```
$* >>~%EOO%i
%ba+r%
%ba+z%
EOO
```

A line-char is treated as an ordinary, non-syntax character with regards to the outer-level line-regex. Lines that start with a regex introducer but do not end with one are used to specify syntax line-chars. Such syntax line-chars can also be specified after (or instead of) match flags. For example:

```
$* >>~/EOO/
/(
/fo+x/|
/ba+r/|
/ba+z/
/)+
EOO
```

As an illustration, if we call the `/fo+x/` expression `A`, `/ba+r/` – `B`, and `/ba+z/` – C, then we can represent the above line-regex in the following more traditional form:

```
(A|B|C)+
```

Only characters from the `.()|*+?{}\0123456789,=!` set are allowed as syntax line-chars with the presence of any other characters being an error.

A blank line as well as the `//` sequence (assuming `/` is the introducer) are treated as an empty line-char. For the purpose of matching, newlines are viewed as separators rather than being part of a line. In particular, in this model, the customary trailing newline at the end of the output introduces a trailing empty line-char. As a result, unless the `:` (no newline) redirect modifier is used, an empty line-char is implicitly added at the end of line-regex.

# 5.22 Cleanup

```
cleanup: ('&'|'&?'|'&!') (<file>|<dir>)
```

If a command creates extra files or directories, then they can be registered for automatic cleanup at the end of the scope (test or group). Files mentioned in redirects are registered automatically. Additionally, certain builtins (for example `touch` and `mkdir`) also register their output files/directories automatically (as described in each builtin's documentation).

If the path ends with a directory separator (slash), then it is assumed to be a directory. Otherwise – a file. A directory about to be removed must be empty (no unexpected output).

The `&` syntax registers a normal or *always* cleanup: the test fails if the file/directory does not exist. The `&?` syntax is a *maybe* cleanup: the file/directory is removed if it exists. Finally, `&!` is a *never* cleanup: it disables a previously registered cleanup for this file/directory (primarily used to disable automatic cleanups registered by builtins).

The path components may contain the `*` and `?` wildcard characters with the following semantics:

```
?    - any single character
*    - all immediate files
*/   - all immediate sub-directories (which must be empty)
**   - all files recursively
**/  - all sub-directories recursively (which must be empty)
***/ - all sub-directories recursively (which must be empty)
       as well as the start directory itself
```

In addition, if the last component in the path is `***` (without trailing directory separator), then it matches all files and sub-directories recursively as well as the start directory itself. For example, the following cleanup will remove `dir/` and its content recursively.

```
$* &dir/***
```

Registering a path for cleanup that is outside the script working directory is an error. You can, however, clean them up manually with `rm/rmdir -f`.

# 5.23 Description

```
description:
  +(':' <text>)
```

Description lines start with a colon (`:`) and are used to document tests and test groups. In a sense they are formalized comments.

A description can be *leading*, that is, specified before the test or group. For tests it can also be *trailing* – specified as a single line after the (last) command of the test. It is an error to specify both leading and trailing descriptions.

By convention the leading description has the following format with all three components being optional.

```
: <id>
: <summary>
:
: <details>
```

If the first line in the description does not contain any whitespaces, then it is assumed to be the test or test group id. If the next line is followed by a blank line, then it is assumed to be the test or test group summary. After the blank line come optional details which are free-form.

The trailing description can only be used to specify the id or summary (but not both).

If an id is not specified then it is automatically derived from the test or test group location. If the test or test group is contained directly in the top-level testscript file, then just its start line number is used as an id. Otherwise, if the test or test group resides in an included file, then the start line number (inside the included file) is prefixed with the line number of the `include` directive followed by the included file name (without the extension) in the form `<line>-<file>-`. This process is repeated recursively in case of nested inclusions.

The start line for a scope (either test or group) is the line containing its opening brace (`{`) and for a test – the first test line.

# 6 Builtins

The Testscript language provides a portable subset of POSIX utilities as builtins. Each utility normally implements the commonly used subset of the corresponding POSIX specification, though there are deviations (for example, in option handling) and extensions, as described in this chapter. Note also that the builtins are implemented in-process with some of the simple ones such as `true/false`, `mkdir`, etc., being just function calls.

To run a system utility instead of a builtin prefix its name with `^`, for example:

```
^cat --squeeze-blank <file>
```

## 6.1 cat

```
cat <file>...
```

Read files in order and write their contents to `stdout`. Read from `stdin` if no file is specified or − is specified as a file name.

## 6.2 cp

```
cp [-p] [--no-cleanup]        <src-file>     <dst-file>
cp [-p] [--no-cleanup] -R|-r  <src-dir>      <dst-dir>
cp [-p] [--no-cleanup]        <src-file>...  <dst-dir>/
cp [-p] [--no-cleanup] -R|-r  <src-path>...  <dst-dir>/
```

Copy files and/or directories. The first two forms make a copy of a single entity at the specified path. The last two copy one or more entities into the specified directory.

If the last argument does not end with a directory separator and the `−R` or `−r` option is not specified, then the first synopsis is assumed where `cp` copies *src-file* as *dst-file* failing if the *src-file* filesystem entry does not exist or if either filesystem entry is a directory.

If the last argument does not end with a directory separator and the `−R` or `−r` option is specified, then the second synopsis is assumed where `cp` copies *src-dir* as *dst-dir* failing if the *src-dir* filesystem entry does not exist or is not a directory or if the *dst-dir* filesystem entry already exists.

In both these cases `cp` also fails if more than two arguments are specified.

If the last argument ends with a directory separator and the `−R` or `−r` option is not specified, then the third synopsis is assumed where `cp` copies one or more *src-file* files into the *dst-dir* directory as if by executing the following command for each file:

```
cp src-file dst-dir/src-name
```

Where *src-name* is the last path component in *src-file*.

In this case `cp` fails if a filesystem entry for any of the *src-file* files does not exist or is a directory or if the *dst-dir* filesystem entry does not exist or is not a directory.

Finally, if the last argument ends with a directory separator and the `-R` or `-r` option is specified, then the last synopsis is assumed where `cp` copies one or more *src-path* files or directories into the *dst-dir* directory as if by executing the following command for each file:

```
cp src-path dst-dir/src-name
```

And the following command for each directory:

```
cp -R src-path dst-dir/src-name
```

Where *src-name* is the last path component in *src-path*. The determination of whether *src-path* is a file or directory is done by querying the filesystem entry type.

In this case `cp` fails if a filesystem entry for any of the *src-path* files/directories does not exist or if the *dst-dir* filesystem entry does not exist or is not a directory. For a *src-path* directory `cp` also fails if the *dst-dir/src-name* filesystem entry already exists.

`-R`|`-r`|`--recursive`
: Copy directories recursively.

`-p`|`--preserve`
: Copy permissions as well as modification and access times.

Unless the `--no-cleanup` option is specified, newly created files and directories that are inside the script working directory are automatically registered for cleanup.

## 6.3 `date`

```
date [-u] [+<format>]
```

Print the local time or, if the `-u` option is specified, the Coordinated Universal Time (UTC) in the specified format.

The optional *format* argument is the `std::put_time()` C++11 manipulator's format string that in addition supports the nanoseconds specifier in the form **`%[`**<*d*>**`N]`** where <*d*> is the optional single delimiter character, for example `.`. If the nanoseconds part is 0, then it is not printed (nor the delimiter character). Otherwise, the nanoseconds part is padded to 9 characters with leading zeros.

Note that this builtin's format specifier set is a superset of the POSIX `date` utility.

If the *format* argument is not specified, then `%a %b %e %H:%M:%S %Z %Y` is used by default.

`-u`|`--utc`
: Print Coordinated Universal Time (UTC).

## 6.4 diff

```
diff [-u|-U <num>] <file1> <file2>
```

Compare the contents of *file1* and *file2*.

The `diff` utility is not a builtin. Instead, the test platform is expected to provide a (reasonably) POSIX-compatible implementation. It should at least supports the `-u` and `-U` options and recognize the `-` file name as an instruction to read from `stdin`. On Windows, GNU `diff` can be assumed (provided as part of the `build2` toolchain).

`-u`
: Produce output in the unified format.

`-U <num>`
: Produce output in the unified output format with *num* lines of context.

## 6.5 echo

```
echo <string>...
```

Write strings to `stdout` separating them with a single space and ending with a newline.

## 6.6 env

```
env [-t <sec>] [-c <dir>] [-u <name>]... [-] [<name>=<value>]... -- \
    <cmd>
```

Run a command limiting its execution time, changing its working directory, and/or adding/removing the variables to/from the environment.

Note that `env` is a *pseudo-builtin*. In particular, its name and the `--` separator must be specified *literally* on the command line. Specifically, they must not be the result of an expansion. Also note that the `--` separator must always be present.

To avoid ambiguity, the variable assignments can be separated from the options with the explicit `-` separator. In the example below the `--unset` variable is added to the environment:

```
env - --unset=FOO -- $*
```

`-t|--timeout <sec>`
: Terminate the command if it fails to complete within the specified number of seconds. See also the `timeout` builtin.

`-s|--timeout-success`
: Assume the command terminated due to the timeout specified with the `-t|--timeout` option to have succeeded.

`-c|--cwd <dir>`
: Change the command's working directory.

`-u|--unset <name>`
: Remove the specified variable from the environment.

See also the `export` builtin.

## 6.7 `exit`

```
exit [<diagnostics>]
```

Exit the current group or test scope skipping any remaining commands.

Note that `exit` is a *pseudo-builtin*. In particular, it must be the only command in the pipe expression and its standard streams cannot be redirected.

Without any arguments `exit` exits the current scope successfully. In this case, if exiting a group scope, teardown commands and cleanups are executed normally.

If an argument is specified, then `exit` exits the current scope and all the outer scopes unsuccessfully, as if the `exit` command failed. In this case the argument must be the diagnostics string describing the error.

## 6.8 `export`

```
export [-c <name>]... [-u <name>]... [<name>=<value>]...
```

Add/remove the variables to/from the current scope commands execution environment and/or clear the previous additions/removals.

Note that `export` is a *pseudo-builtin*. In particular, it must be the only command in the pipe expression, it either succeeds or terminates abnormally, and its standard streams cannot be redirected.

The environment variables can be added and removed on multiple levels: with the `export` builtin in the nested test group scopes and the test scope and with the `env` builtin for individual commands. Before executing a command, all the variable additions and removals from its environment hierarchy are merged so that those specified in the inner levels override those specified in the outer levels.

`-c|--clear <name>`
: Clear the previous variable addition/removal to/from the environment, if exists.

`-u|--unset <name>`
: Remove the specified variable from the environment.

## 6.9 `false`

```
false
```

Do nothing and terminate normally with the 1 exit code (indicating failure).

## 6.10 `find`

```
find <start-path>... [<expression>]
```

Search for filesystem entries in a filesystem hierarchy. Traverse filesystem hierarchies from each *start-path* specified on the command line, evaluate for each filesystem entry the boolean *expression* consisting of the options-like arguments called *primaries*, and print the filesystem entry path if it evaluates to `true`, one path per line. The primaries are combined into the expression with an implicit logical AND operator. The empty expression always evaluates to `true`.

Note that the implementation deviates from POSIX in a number of ways. It only supports a small subset of primaries and doesn't support compound expressions, negations, logical OR and (explicit) AND operators, and the `-type` primary values other than `f`, `d`, and `l`. It, however, supports the `-mindepth` and `-maxdepth` primaries which are not specified by POSIX but are supported by the major `find` utility implementations.

The following primaries are supported:

`-name <pattern>`
: Evaluates to `true` if a filesystem entry base name matches the specified wildcard pattern.

`-type <type>`
: Evaluates to `true` if a filesystem entry type matches the specified type: `f` for a regular file, `d` for a directory, and `l` for a symbolic link.

`-mindepth <depth>`
: Evaluates to `true` if a filesystem entry directory level is not less than the specified depth. The level of the *start-path* entries specified on the command line is 0.

`-maxdepth <depth>`
: Evaluates to `true` if a filesystem entry directory level is not greater than the specified depth. The level of the *start-path* entries specified on the command line is 0. Note that the implementation is smart enough not to traverse a directory when the maximum depth is reached.

## 6.11 `ln`

```
ln [--no-cleanup] -s <target-path>    <link-path>
ln [--no-cleanup] -s <target-path>... <dir>/
```

Create symbolic links to files and/or directories. The first form creates a single target link at the specified path. The second form creates links to one or more targets inside the specified directory.

If the last argument does not end with a directory separator, then the first synopsis is assumed where `ln` creates the symbolic link to *target-path* at *link-path* failing if the *target-path* filesystem entry does not exist, *link-path* filesystem entry already exists or more than two arguments are specified. If *target-path* is relative, then it is assumed to be relative to the *link-path*'s directory.

If the last argument ends with a directory separator, then the second synopsis is assumed where `ln` creates one or more symbolic links to *target-path* files or directories inside the *dir* directory as if by executing the following command for each target:

```
ln -s target-path dir/target-name
```

Where *target-name* is the last path component in *target-path*.

For both cases `ln` falls back to creating a hard link if symbolic link creation is not supported. If hard link creation is not supported either, then `ln` falls back to copying the content, recursively in case of a directory target.

`-s|--symbolic`
: Create symbolic links. Note that creation of hard links is currently not supported, so this option is always required.

Unless the `--no-cleanup` option is specified, created filesystem entries that are inside the script working directory are automatically registered for cleanup.

## 6.12 mkdir

```
mkdir [--no-cleanup] [-p] <dir>...
```

Create directories. Unless the `-p` option is specified, all the leading directories must exist and the directory itself must not exist.

`-p|--parents`
: Create missing leading directories and ignore directories that already exist.

Unless the `--no-cleanup` option is specified, newly created directories (including the leading ones) that are inside the script working directory are automatically registered for cleanup.

## 6.13 mv

```
mv [--no-cleanup] [-f] <src-path>    <dst-path>
mv [--no-cleanup] [-f] <src-path>... <dst-dir>/
```

Rename or move files and/or directories.

The first form moves an entity to the specified path. The parent directory of the destination path must exist. An existing destination entity is replaced with the source if they are both either directories or non-directories (files, symlinks, etc). In the former case the destination

directory must be empty. The source and destination paths must not be the same nor be the test working directory or its parent directory. The source path must also not be outside the script working directory unless the `-f` option is specified.

The second form moves one or more entities into the specified directory as if by executing the following command for each entity:

```
mv src-path dst-dir/src-name
```

Where *src-name* is the last path component in *src-path*.

`-f`|`--force`
: Do not fail if a source path is outside the script working directory.

Unless the `--no-cleanup` option is specified, the cleanups registered for the source entities are adjusted according to their new names and/or locations. If the destination entity already exists or is outside the test working directory then the source entity cleanup is canceled. Otherwise the source entity cleanup path is replaced with the destination path. If the source entity is a directory, then, in addition, cleanups that are sub-paths of this directory are made sub-paths of the destination directory.

Note that the implementation deviates from POSIX in a number of ways. It never interacts with the user and fails immediately if unable to act on an argument. It does not check for dot containment in the path nor considers filesystem permissions. In essence, it simply tries to move the filesystem entry.

## 6.14 `rm`

```
rm [-r] [-f] <path>...
```

Remove filesystem entries. To remove a directory (even empty) the `-r` option must be specified.

The path must not be the test working directory or its parent directory. It also must not be outside the script working directory unless the `-f` option is specified.

`-r`|`--recursive`
: Remove directories and their contents recursively.

`-f`|`--force`
: Do not fail if no path is specified, the path does not exist, or is outside the script working directory.

Note that the implementation deviates from POSIX in a number of ways. It never interacts with the user and fails immediately if unable to act on an argument. It does not check for dot containment in the path nor considers filesystem permissions. In essence, it simply tries to remove the filesystem entry.

## 6.15 `rmdir`

```
rmdir [-f] <dir>...
```

Remove directories. The directory must be empty and not be the test working directory or its parent directory. It also must not be outside the script working directory unless the `-f` option is specified.

`-f`|`--force`
: Do not fail if no directory is specified, the directory does not exist, or is outside the script working directory.

## 6.16 `sed`

```
sed [-n] [-i] (-e <script>)... [<file>]
```

Read text from *file*, make editing changes according to *script*, and write the result to `stdout`. If multiple *scripts* are present, then they are applied in the order specified. If *file* is not specified or is `-`, read from `stdin`. If both *file* and the `-i` option are specified then edit the *file* in place. Specifying `-i` when reading from `stdin` is illegal.

Note that this builtin implementation deviates substantially from POSIX `sed` (as described next). Most significantly, the regular expression flavor is ECMAScript (more specifically, ECMA-262-based C++11 regular expressions).

`-n`|`--quiet`
: Suppress automatic printing of the pattern space at the end of the script execution.

`-i`|`--in-place`
: Edit *file* in place.

`-e`|`--expression <script>`
: Editing commands to be executed. At least one script must be specified.

To perform the transformation `sed` reads each line of input (without the newline) into the pattern space. It then executes the script commands on the pattern space. At the end of the script execution, unless the `-n` option is specified, `sed` writes the pattern space to output followed by a newline.

Currently, only single-command scripts using the following editing commands are supported.

`s/<regex>/<replacement>/<flags>`
: Match *regex* against the pattern space. If successful, replace the part of the pattern space that matched with *replacement*. If the `g` flag is present in *flags* then continue substituting subsequent matches of *regex* in the same pattern space. If the `p` flag is present in *flags* and the replacement has been made, then write the pattern space to `stdout` and start the next cycle by proceeding to read the next line of input. If both `g` and `p` were specified, then write the pattern space out only after the last substitution.

Any character other than `\` (backslash) or newline can be used instead of `/` (slash) to delimit *regex*, *replacement*, and *flags*. Note that no escaping of the delimiter character is supported.

If *regex* starts with `^`, then it only matches at the beginning of the pattern space. Similarly, if it ends with `$`, then it only matches at the end of the pattern space. If the `i` flag is present in *flags*, then the match is performed in a case-insensitive manner.

In *replacement*, besides the standard ECMAScript escape sequences (`$1`, `$2`, `$&`, etc), the following additional sequences are recognized:

```
\N - Nth capture, where N is in the 1-9 range.

\u - Convert next character to the upper case.
\l - Convert next character to the lower case.

\U - Convert next characters until \E to the upper case.
\L - Convert next characters until \E to the lower case.

\n - Newline.
\\ - Literal backslash.
```

Note that unlike POSIX semantics, just `&` does not have a special meaning in *replacement*.

## 6.17 `set`

```
set [-e] [-n|-w] <var> [<attr>]
```

Set variable from the `stdin` input.

Note that `set` is a *pseudo-builtin*. In particular, it must be the last command in the pipe expression, it either succeeds or terminates abnormally, and its `stderr` cannot be redirected. Note also that all the variables on the command line are expanded before any `set` commands are executed, for example:

```
foo = foo
echo 'bar' | set foo && echo $foo  # foo
echo $foo                          # bar
```

Unless the `-e` option is specified, a single final newline is ignored in the input.

If the `-n` option is specified, then the input is split into a list of elements at newlines, including a final blank element in case of `-e`. Multiple consecutive newlines are not collapsed.

If the `-w` option is specified, then the input is split into a list of elements at whitespaces, including a final blank element in case of `-e`. In this mode if `-e` is not specified, then all (and not just newline) trailing whitespaces are ignored. Multiple consecutive whitespaces (including newlines) are collapsed.

If neither `-n` nor `-w` is specified, then the entire input is used as a single element, including a final newline in case of `-e`.

If the *attr* argument is specified, then it must contain a list of value attributes enclosed in `[]`, for example:

```
sed -e 's/foo/bar/' input | set x [string]
```

Note that this is also the only way to set a variable with a computed name, for example:

```
foo = FOO
set $foo [null] <-
```

`-e`|`--exact`
: Do not ignore the final newline.

`-n`|`--newline`
: Split the input into a list of elements at newlines.

`-w`|`--whitespace`
: Split the input into a list of elements at whitespaces.

## 6.18 `sleep`

```
sleep <seconds>
```

Suspend the current test or test group execution for at least the specified number of seconds. Note that in order to improve resource utilization, the implementation may sleep longer than requested, potentially significantly.

## 6.19 `test`

```
test -f|-d <path>
```

Test the specified *path* according to one of the following options. Succeed (0 exit code) if the test passes and fail (non-0 exit code) otherwise.

`-f`|`--file`
: Path exists and is to a regular file.

`-d`|`--directory`
: Path exists and is to a directory.

Note that tests dereference symbolic links.

## 6.20 `timeout`

```
timeout [-s] [<group-timeout>]/[<test-timeout>]
timeout [-s] <timeout>
```

Specify test and/or test group timeout.

The first form sets the test group and/or individual test timeouts and can only be used as a setup command. Either of the timeouts (but not both) can be omitted.

The second form sets the test group timeout if used as a setup or teardown command and the remaining test fragment timeout if used as a test command.

In both forms the timeouts are specified in seconds with the zero value clearing the previously set timeout.

Note that `timeout` is a *pseudo-builtin*. In particular, it must be the only command in the pipe expression, it either succeeds or terminates abnormally, and its standard streams cannot be redirected.

The timeouts can be set on multiple levels: via the `config.test.timeout` variable on the (potentially nested) project root scopes (see `test` module for details), with the `timeout` builtin in the nested test group scopes and the test scope, and with the `env` builtin for individual commands. Each command must complete before the nearest timeout from its timeout hierarchy. Failed that, a command is terminated forcibly causing the entire `test` operation to fail unless the expired timeout was specified with the `--success` option, in which case the timed out command is assumed to have succeeded.

`-s|--success`
: Assume a command terminated due to this timeout to have succeeded.

## 6.21 `touch`

```
touch [--no-cleanup] [--after <ref-file>] <file>...
```

Change file access and modification times to the current time. Create files that do not exist. Fail if a filesystem entry other than the file exists for the specified name.

`--after <ref-file>`
: Keep touching the file until its modification time becomes after that of the specified reference file.

Unless the `--no-cleanup` option is specified, newly created files that are inside the script working directory are automatically registered for cleanup.

## 6.22 `true`

```
true
```

Do nothing and terminate normally with the 0 exit code (indicating success).

# 7 Style Guide

This chapter describes the testing guidelines and the Testscript style that is used in the `build2` project.

The primary goal of testing in `build2` is not to exhaustively test every possible situation. Rather, it is to keep tests comprehensible and maintainable in the long run.

To this effect, don't try to test every possible combination; this striving will quickly lead to everyone drowning in hundreds of tests that are only slight variations of each other. Sometimes combination tests are useful but generally keep things simple and test one thing at a time. The belief is that real-world usage will uncover much more interesting interactions (which must become regression tests) that you would never have thought of yourself. To quote a famous physicist, "*... the imagination of nature is far, far greater than the imagination of man.*"

To expand on combination tests, don't confuse them with corner case tests. As an example, say you have tests for feature A and B. Now you wonder what if for some reason they don't work together. Note that you don't have a clear understanding let alone evidence of why they might not work together; you just want to add one more test, *for good measure*. We don't do that. To put it another way, for each test you should have a clear understanding of what logic in the code you are testing.

One approach that we found works well is to look at the diff of changes you would like to commit and make sure you at least have a test that exercises each *happy* (non-error) *logic branch*. For important code you may also want to do so for *unhappy logic branches*.

It is also a good idea to keep testing in mind as you implement things. When tempted to add a small special case just to make the result a little bit *nicer*, remember that you will also have to test this special case.

If the functionality is well exposed in the program, prefer functional to unit tests since the former test the end result rather than something intermediate and possibly mocked. If unit-testing a complex piece of functionality, consider designing a concise, textual *mini-format* for input (either via command line or `stdin`) and output rather than constructing the test data and expected results programmatically.

Documentation-wise, each test should at least include an explicit id that adequately summarizes what it tests. Add a summary or even details for more complex tests. Failure tests usually fall into this category.

Use the leading description for multi-line tests, for example:

```
: multi-name
:
$* 'John' 'Jane' >>EOO
Hello, John!
Hello, Jane!
EOO
```

Here is an example of a description that includes all three components:

```
: multi-name
: Test multiple name arguments
:
: This test makes sure we properly handle multiple names passed as
: separate command line arguments.
:
$* 'John' 'Jane' >>EOO
Hello, John!
Hello, Jane!
EOO
```

Separate multi-line tests with blank lines. You may want to place larger tests into explicit test scopes for better visual separation (this is especially helpful if the test contains blank lines, for example, in here-document fragments). In this case the description should come before the scope. Note that here-documents are indented as well. For example:

```
: multi-name
:
{
  $* 'John' 'Jane' >>EOO
  Hello, John!

  Hello, Jane!

  EOO
}
```

One-line tests may use the trailing description (which must always be the test id). Within a test block (one-liners without a blank between them), the ids should be aligned, for example:

```
$* John >'Hi, John!'       : custom-john
$* World >'Hello, World!'  : custom-world
```

Note that you are free to put multiple spaces between the end of the command line and the trailing description. But don't try to align ids between blocks – this is a maintenance pain.

If multiple tests belong to the same group, consider placing them into an explicit group scope. A good indication that tests form a group is if their ids start with the same prefix, as in the above example. If placing tests into a group scope, use the prefix as the group's id and don't repeat it in the tests. It is also a good idea to give the summary of the group, for example:

```
: custom
: Test custom greetings
:
{
  $* John >'Hi, John!'       : john
  $* World >'Hello, World!'  : world
}
```

In the same vein, don't repeat the testscript id in group or test ids. For example, if the above tests were in `greeting.testscript`, then using `custom-greeting` as the group id would be unnecessarily repetitive since the id path would then become `greeting/custom-greeting/john`, etc.

We quote values that are *strings* as opposed to options, file names, paths (unless they contain spaces), integers, or boolean. When quoting, use single quotes unless you need expansions (or single quotes) inside. Note that unlike Bash you do not need to quote variable expansions in order to preserve whitespaces. For example:

```
arg = 'Hello   Spaces'
echo $arg              # Hello   Spaces
```

For further reading on testing that we (mostly) agree with, see:

How I Write Tests by Nelson Elhage
: The only part we don't agree on is the (somewhat implied) suggestion to write as many tests as possible.