Testing Swift

This document describes how we test the Swift compiler, the Swift runtime, and the Swift standard library.

Testing approaches

We use multiple approaches to test the Swift toolchain.

- LLVM lit-based testsuites for the compiler, runtime and the standard library.
- Unit tests for sub-tools.
- A selection of open source projects written in Swift.

The LLVM lit-based testsuite

Purpose: primary testsuites for the Swift toolchain.

Contents: Functional and regression tests for all toolchain components.

Run by:

- Engineers and contributors are expected to run tests from these testsuites locally before committing. (Usually on a single platform, and not necessarily all tests.)
- Buildbots run all tests, on all supported platforms. Smoke testing skips the iOS, tvOS, and watchOS platforms.

Testsuite subsets

The testsuite is split into five subsets:

- Primary testsuite, located under `swift/test`.
- Validation testsuite, located under `swift/validation-test`.
- Unit tests, located under `swift/unittests`.
- Long tests, which are marked with `REQUIRES: long_test`.
- Stress tests, which are marked with `REQUIRES: stress_test`.

Running the LLVM lit-based testsuite

The simplest way to run the Swift test suite is with the `--test` switch to `utils/build-script`. This will run the primary test suite. The buildbot runs validation tests, so if those are accidentally broken, it should not go unnoticed.

Before committing a large change to a compiler (especially a language change), or API changes to the standard library, it is recommended to run validation test suite, via `utils/build-script --validation-test`.

Using `utils/build-script` will rebuild all targets which can add substantial time to a debug cycle.

Using utils/run-test Using `utils/run-test` allows the user to run a single test or tests in a specific directory. This can significantly speed up the debug cycle. One can use this tool instead of invoking `lit.py` directly as described in the next section.

Here is an example of running the `test/Parse` tests:

```
% ${swift_SOURCE_ROOT}/utils/run-test --build-dir ${SWIFT_BUILD_DIR} ${swift_SOURCE_ROOT
```

Note that one example of a valid `${SWIFT_BUILD_DIR}` is `{swift_SOURCE_ROOT}/../build/Ninja-DebugAsser` It differs based on your build options and on which directory you invoke the script from.

For full help options, pass `-h` to `utils/run-test` utility.

Using lit.py Using `lit.py` directly can provide more control and faster feedback to your development cycle. To invoke LLVM's `lit.py` script directly, it must be configured to use your local build directory. For example:

```
% ${LLVM_SOURCE_ROOT}/utils/lit/lit.py -sv ${SWIFT_BUILD_DIR}/test-macosx-x86_64/Parse/
```

This runs the tests in the 'test/Parse/' directory targeting 64-bit macOS. The `-sv` options give you a nice progress bar and only show you output from the tests that fail.

One downside of using this form is that you're appending relative paths from the source directory to the test directory in your build directory. (That is, there may not actually be a directory named 'Parse' in 'test-macosx-x86_64/'; the invocation works because there is one in the source 'test/' directory.) There is a more verbose form that specifies the testing configuration explicitly, which then allows you to test files regardless of location.

```
% ${LLVM_SOURCE_ROOT}/utils/lit/lit.py -sv --param swift_site_config=${SWIFT_BUILD_DIR}/
```

For more complicated configuration, copy the invocation from one of the build targets mentioned above and modify it as necessary. lit.py also has several useful features, like timing tests and providing a timeout. Check these features out with `lit.py -h`. We document some of the more useful ones below:

Standard lit.py invocation options

- `-s` reduces the amount of output that lit shows.
- `-v` causes a test's commandline and output to be printed if the test fails.
- `-vv` causes a test's commandline and output to be printed if the test fails, showing the exact command in the test execution script where progress stopped; this can be useful for finding a single silently-failing RUN line, amid a sequence.
- `-a` causes a test's commandline and output to always be printed.
- `--filter=<pattern>` causes only tests with paths matching the given regular expression to be run. Alternately, you can use the

`LIT_FILTER='<pattern>'` environment variable, in case you're invoking `lit.py` through some other tool such as `build-script`.

- `-i` causes tests that have a newer modification date and failing tests to be run first. This is implemented by updating the mtimes of the tests.
- `--no-execute` causes a dry run to be performed. *NOTE* This means that all tests are assumed to PASS.
- `--time-tests` will cause elapsed wall time to be tracked for each test.
- `--timeout=<MAXINDIVIDUALTESTTIME>` sets a maximum time that can be spent running a single test (in seconds). 0 (the default means no time limit.
- `--max-failures=<MAXFAILURES>` stops execution after `MAXFAILURES` number of failures.

Swift-specific testing options

- `--param gmalloc` will run all tests under Guard Malloc (macOS only). See `man libgmalloc` for more information.
- `--param swift-version=<MAJOR>` overrides the default Swift language version used by swift/swiftc and swift-ide-test.
- `--param interpret` is an experimental option for running execution tests using Swift's interpreter rather than compiling them first. Note that this does not affect all substitutions.
- `--param swift_test_mode=<MODE>` drives the various suffix variations mentioned above. Again, it's best to get the invocation from the existing build system targets and modify it rather than constructing it yourself.
- `--param use_os_stdlib` will run all tests with the standard libraries coming from the OS.

Remote testing options

- `--param remote_run_host=[USER@]<HOST>[:PORT]` causes execution tests that would normally be run on the host (via the `%target-run` substitutions described below) to be run over SSH on another machine instead, using the `remote-run` tool in the `utils` directory. Requires that `remote_run_tmpdir` also be provided.
- `--param remote_run_tmpdir=<PATH>` specifies the scratch directory to be used on the remote machine when testing with `remote_run_host`.
- `--param remote_run_identity=<FILE>` provides an SSH private key to be used when testing with `remote_run_host`. (`remote-run` does not support passwords.)
- `--param remote_run_extra_args="ARG1 ARG2 ..."` provides a list of extra arguments to pass to `remote-run`. (This can be used with `remote-run`'s `-o` option to pass extra options to SSH.)
- `--param remote_run_skip_upload_stdlib` assumes that the standard library binaries have already been uploaded to `remote_run_tmpdir` and are up to date. This is meant for repeat runs and probably shouldn't be

used in automation.

CMake Although it is not recommended for day-to-day contributions, it is also technically possible to execute the tests directly via CMake. For example, if you have built Swift products at the directory `build/Ninja-ReleaseAssert/swift-macosx-x86_64`, you may run the entire test suite directly using the following command:

```
cmake --build build/Ninja-ReleaseAssert/swift-macosx-x86_64 -- check-swift-macosx-x86_64
```

Note that `check-swift` is suffixed with a target operating system and architecture. Besides `check-swift`, other targets are also available. Here's the full list:

- `check-swift`: Runs tests from the `${SWIFT_SOURCE_ROOT}/test` directory.
- `check-swift-only_validation`: Runs tests from the `${SWIFT_SOURCE_ROOT}/validation-test` directory.
- `check-swift-validation`: Runs the primary and validation tests, without the long tests or stress tests.
- `check-swift-only_long`: Runs long tests only.
- `check-swift-only_stress`: Runs stress tests only.
- `check-swift-all`: Runs all tests (primary, validation, and long).
- `SwiftUnitTests`: Builds all unit tests. Executables are located under `${SWIFT_BUILD_DIR}/unittests` and must be run individually.

For every target above, there are variants for different optimizations:

- the target itself (e.g., `check-swift`) – runs all tests from the primary testsuite. The execution tests are run in `-Onone` mode.
- the target with `-optimize` suffix (e.g., `check-swift-optimize`) – runs execution tests in `-O` mode. This target will only run tests marked as `executable_test`.
- the target with `-optimize_unchecked` suffix (e.g., `check-swift-optimize_unchecked`) – runs execution tests in `-Ounchecked` mode. This target will only run tests marked as `executable_test`.
- the target with `-only_executable` suffix (e.g., `check-swift-only_executable-iphoneos-arm64`) – runs tests marked with `executable_test` in `-Onone` mode.
- the target with `-only_non_executable` suffix (e.g., `check-swift-only_non_executable-iphoneos-arm6` – runs tests not marked with `executable_test` in `-Onone` mode.

Writing tests

General guidelines When adding a new testcase, try to find an existing test file focused on the same topic rather than starting a new test file. There is a fixed runtime cost for every test file. On the other hand, avoid dumping new tests in a file that is only remotely related to the purpose of the new tests.

Don't limit a test to a certain platform or hardware configuration just because this makes the test slightly easier to write. This sometimes means a little bit more work when adding the test, but the payoff from the increased testing is significant. We heavily rely on portable tests to port Swift to other platforms.

Avoid using unstable language features in tests which test something else (for example, avoid using an unstable underscored attribute when another non-underscored attribute would work).

Avoid using arbitrary implementation details of the standard library. Always prefer to define types locally in the test, if feasible.

Avoid purposefully shadowing names from the standard library, this makes the test extremely confusing (if nothing else, to understand the intent — was the compiler bug triggered by this shadowing?) When reducing a compiler testcase from the standard library source, rename the types and APIs in the testcase to differ from the standard library APIs.

In IRGen, SILGen and SIL tests, avoid using platform-dependent implementation details of the standard library (unless doing so is point of the test). Platform-dependent details include:

- `Int` (use integer types with explicit types instead).
- Layout of `String`, `Array`, `Dictionary`, `Set`. These differ between platforms that have Objective-C interop and those that don't.

Unless testing the standard library, avoid using arbitrary standard library types and APIs, even if it is very convenient for you to do so in your tests. Using the more common APIs like `Array` subscript or `+` on `IntXX` is acceptable. This is important because you can't rely on the full standard library being available. The long-term plan is to introduce a mock, minimal standard library that only has a very basic set of APIs.

If you write an executable test please add `REQUIRES: executable_test` to the test.

Every long test must also include `REQUIRES: nonexecutable_test` or `REQUIRES: executable_test`.

Substitutions in lit tests Substitutions that start with `%target` configure the compiler for building code for the target that is not the build machine:

- `%target-typecheck-verify-swift`: parse and type check the current Swift file for the target platform and verify diagnostics, like `swift -frontend -typecheck -verify %s`. For further explanation of `-verify` mode, see Diagnostics.md.

  Use this substitution for testing semantic analysis in the compiler.

- `%target-swift-frontend`: run `swift -frontend` for the target.

Use this substitution (with extra arguments) for tests that don't fit any other pattern.

- `%target-swift-frontend(mock-sdk:` *mock sdk arguments* `)` *other arguments*: like `%target-swift-frontend`, but allows to specify command line parameters (typically `-sdk` and `-I`) to use a mock SDK and SDK overlay that would take precedence over the target SDK.
- `%target-build-swift`: compile and link a Swift program for the target.

  Use this substitution only when you intend to run the program later in the test.
- `%target-run-simple-swift`: build a one-file Swift program and run it on the target machine.

  Use this substitution for executable tests that don't require special compiler arguments.

  Add `REQUIRES: executable_test` to the test.
- `%target-run-simple-swift(` *compiler arguments* `)`: like `%target-run-simple-swift`, but enables specifying compiler arguments when compiling the Swift program.

  Add `REQUIRES: executable_test` to the test.
- `%target-run-simple-swiftgyb`: build a one-file Swift `.gyb` program and run it on the target machine.

  Use this substitution for executable tests that don't require special compiler arguments.

  Add `REQUIRES: executable_test` to the test.
- `%target-run-simple-swiftgyb(` *compiler arguments* `)`: like `%target-run-simple-swiftgyb`, but enables specifying compiler arguments when compiling the Swift program.

  Add `REQUIRES: executable_test` to the test.
- `%target-run-simple-swift`: build a one-file Swift program and run it on the target machine.

  Use this substitution for executable tests that don't require special compiler arguments.

  Add `REQUIRES: executable_test` to the test.
- `%target-run-stdlib-swift`: like `%target-run-simple-swift` with `-parse-stdlib -Xfrontend -disable-access-control`.

  This is sometimes useful for testing the Swift standard library.

  Add `REQUIRES: executable_test` to the test.

- `%target-repl-run-simple-swift`: run a Swift program in a REPL on the target machine.
- `%target-run`: run a command on the target machine.

  Add `REQUIRES: executable_test` to the test.
- `%target-jit-run`: run a Swift program on the target machine using a JIT compiler.
- `%target-swiftc_driver`: run `swiftc` for the target.
- `%target-sil-opt`: run `sil-opt` for the target.
- `%target-sil-func-extractor`: run `sil-func-extractor` for the target.
- `%target-swift-ide-test`: run `swift-ide-test` for the target.
- `%target-swift-ide-test(mock-sdk:` *mock sdk arguments* `)` *other arguments*: like `%target-swift-ide-test`, but allows to specify command line parameters to use a mock SDK.
- `%target-swift-autolink-extract`: run `swift-autolink-extract` for the target to extract its autolink flags on platforms that support them (when the autolink-extract feature flag is set)
- `%target-clang`: run the system's `clang++` for the target.

  If you want to run the `clang` executable that was built alongside Swift, use `%clang` instead.
- `%target-ld`: run `ld` configured with flags pointing to the standard library directory for the target.
- `%target-cc-options`: the clang flags to setup the target with the right architecture and platform version.
- `%target-sanitizer-opt`: if sanitizers are enabled for the build, the corresponding `-fsanitize=` option.
- `%target-triple`: a triple composed of the `%target-cpu`, the vendor, the `%target-os`, and the operating system version number. Possible values include `i386-apple-ios7.0` or `armv7k-apple-watchos2.0`.
- `%target-cpu`: the target CPU instruction set (`i386`, `x86_64`, `armv7`, `armv7k`, `arm64`).
- `%target-os`: the target operating system (`macosx`, `darwin`, `linux`, `freebsd`, `windows-cygnus`, `windows-gnu`).
- `%target-is-simulator`: `true` if the target is a simulator (iOS, watchOS, tvOS), otherwise `false`.
- `%target-object-format`: the platform's object format (`elf`, `macho`, `coff`).

- `%target-runtime`: the platform's Swift runtime (objc, native).
- `%target-ptrsize`: the pointer size of the target (32, 64).
- `%target-swiftmodule-name` and `%target-swiftdoc-name`: the basename of swiftmodule and swiftdoc files for a framework compiled for the target (for example, `arm64.swiftmodule` and `arm64.swiftdoc`).
- `%target-sdk-name`: only for Apple platforms: `xcrun`-style SDK name (`macosx`, `iphoneos`, `iphonesimulator`).
- `%target-static-stdlib-path`: the path to the static standard library.

  Add `REQUIRES: static_stdlib` to the test.
- `%target-rtti-opt`: the `-frtti` or `-fno-rtti` option required to link with the Swift libraries on the target platform.
- `%target-cxx-lib`: the argument to add to the command line when using `swiftc` and linking in a C++ object file. Typically `-lc++` or `-lstdc++` depending on platform.
- `%target-msvc-runtime-opt`: for Windows, the MSVC runtime option, e.g. `-MD`, to use when building C/C++ code to link with Swift.

Always use `%target-*` substitutions unless you have a good reason. For example, an exception would be a test that checks how the compiler handles mixing module files for incompatible platforms (that test would need to compile Swift code for two different platforms that are known to be incompatible).

When you can't use `%target-*` substitutions, you can use:

- `%swift_driver_plain`: run `swift` for the build machine.
- `%swift_driver`: like `%swift_driver_plain` with `-module-cache-path` set to a temporary directory used by the test suite, and using the `SWIFT_TEST_OPTIONS` environment variable if available.
- `%swiftc_driver`: like `%target-swiftc_driver` for the build machine.
- `%swiftc_driver_plain`: like `%swiftc_driver`, but does not set the `-module-cache-path` to a temporary directory used by the test suite, and does not respect the `SWIFT_TEST_OPTIONS` environment variable.
- `%sil-opt`: like `%target-sil-opt` for the build machine.
- `%sil-func-extractor`: run `%target-sil-func-extractor` for the build machine.
- `%lldb-moduleimport-test`: run `lldb-moduleimport-test` for the build machine in order simulate importing LLDB importing modules from the `__apple_ast` section in Mach-O files. See `tools/lldb-moduleimport-test/` for details.
- `%swift-ide-test`: like `%target-swift-ide-test` for the build machine.

- `%swift-ide-test_plain`: like `%swift-ide-test`, but does not set the `-module-cache-path` or `-completion-cache-path` to temporary directories used by the test suite.
- `%swift`: like `%target-swift-frontend` for the build machine.
- `%clang`: run the locally-built `clang`. To run `clang++` for the target, use `%target-clang`.

Other substitutions:

- `%clang-include-dir`: absolute path of the directory where the Clang include headers are stored on Linux build machines.
- `%clang-importer-sdk`: FIXME.
- `%clang_apinotes`: run `clang -cc1apinotes` using the locally-built clang.
- `%sdk`: only for Apple platforms: the `SWIFT_HOST_VARIANT_SDK` specified by tools/build-script. Possible values include `IOS` or `TVOS_SIMULATOR`.
- `%gyb`: run `gyb`, a boilerplate generation script. For details see `utils/gyb`.
- `%platform-module-dir`: absolute path of the directory where the standard library module file for the target platform is stored. For example, `/.../lib/swift/macosx`.
- `%platform-sdk-overlay-dir`: absolute path of the directory where the SDK overlay module files for the target platform are stored.
- `%swift_src_root`: absolute path of the directory where the Swift source code is stored.
- `%{python}`: run the same Python interpreter that's being used to run the current `lit` test.
- `%FileCheck`: like the LLVM `FileCheck` utility, but occurrences of full paths to the source and build directories in the input text are replaced with path-independent constants.
- `%raw-FileCheck`: the LLVM `FileCheck` utility.
- `%empty-directory(` *directory-name* `)`: ensures that the given directory exists and is empty. Equivalent to `rm -rf directory-name && mkdir -p directory-name`.

When writing a test where output (or IR, SIL) depends on the bitness of the target CPU, use this pattern::

```
// RUN: %target-swift-frontend ... | %FileCheck --check-prefix=CHECK --check-prefix=CHECK-

// CHECK: common line
// CHECK-32: only for 32-bit
// CHECK-64: only for 64-bit
// FileCheck does a single pass for a combined set of CHECK lines, so you can
// do this:
//
// CHECK: define @foo() {
// CHECK-32: integer_literal $Builtin.Int32, 0
// CHECK-64: integer_literal $Builtin.Int64, 0
```

When writing a test where output (or IR, SIL) depends on the target CPU itself, use this pattern::

```
// RUN: %target-swift-frontend ... | %FileCheck --check-prefix=CHECK --check-prefix=CHECK-

// CHECK: common line
// CHECK-i386:        only for i386
// CHECK-x86_64:      only for x86_64
// CHECK-armv7:       only for armv7
// CHECK-arm64:       only for arm64
// CHECK-powerpc64:   only for powerpc64
// CHECK-powerpc64le: only for powerpc64le
```

Features for `REQUIRES` and `XFAIL` FIXME: full list.

- `swift_ast_verifier`: present if the AST verifier is enabled in this build.
- When writing a test specific to x86, if possible, prefer `REQUIRES: CPU=i386 || CPU=x86_64` to `REQUIRES: CPU=x86_64`.
- `swift_test_mode_optimize[_unchecked|none]` and `swift_test_mode_optimize[_unchecked|none]_` specify a test mode plus cpu configuration.
- `optimized_stdlib_<CPUNAME>`: an optimized stdlib plus cpu configuration.
- `SWIFT_VERSION=<MAJOR>`: restricts a test to Swift 3, Swift 4, Swift 5. If you need to use this, make sure to add a test for the other version as well unless you are specifically testing `-swift-version`-related functionality.
- `XFAIL: linux`: tests that need to be adapted for Linux, for example parts that depend on Objective-C interop need to be split out.

Feature `REQUIRES: executable_test` This feature marks an executable test. The test harness makes this feature generally available. It can be used to restrict the set of tests to run.

StdlibUnittest Tests accept command line parameters, run StdlibUnittest-based test binary with `--help` for more information.

Testing memory management in execution tests In execution tests, memory management testing should be performed using local variables enclosed in a closure passed to the standard library `autoreleasepool` function. For example::

```
// A counter that's decremented by Canary's deinitializer.
var CanaryCount = 0

// A class whose instances increase a counter when they're destroyed.
class Canary {
  deinit { ++CanaryCount }
}

// Test that a local variable is correctly released before it goes out of
// scope.
CanaryCount = 0
autoreleasepool {
  let canary = Canary()
}
assert(CanaryCount == 1, "canary was not released")
```

Memory management tests should be performed in a local scope because Swift does not guarantee the destruction of global variables. Code that needs to interoperate with Objective-C may put references in the autorelease pool, so code that uses an `if true {}` or similar no-op scope instead of `autoreleasepool` may falsely report leaks or fail to catch overrelease bugs. If you're specifically testing the autoreleasing behavior of code, or do not expect code to interact with the Objective-C runtime, it may be OK to use `if true {}`, but those assumptions should be commented in the test.

Enabling/disabling the lldb test allowlist It's possible to enable a allowlist of swift-specific lldb tests to run during PR smoke testing. Note that the default set of tests which run (which includes tests not in the allowlist) already only includes swift-specific tests.

Enabling the allowlist is an option of last-resort to unblock swift PR testing in the event that lldb test failures cannot be resolved in a timely way. If this becomes necessary, be sure to double-check that enabling the allowlist actually unblocks PR testing by running the smoke test build preset locally.

To enable the lldb test allowlist, add `-G swiftpr` to the `LLDB_TEST_CATEGORIES` variable in `utils/build-script-impl`. Disable it by removing that option.