# Swift implemented in Swift

This is the part of the Swift compiler which is implemented in Swift itself.

## Building

Although this directory is organized like a Swift package, building is done with CMake. Beside building, the advantage of a Swift package structure is that it is easy to edit the sources with Xcode.

The Swift modules are compiled into object files and linked as a static library. The build-script option `--bootstrapping` controls how the Swift modules are built.

Currently, the `swift-frontend` and `sil-opt` tools include the Swift modules. Tools, which don't use any parts from the Swift code don't need to link the Swift library. And also `swift-frontend` does not strictly require to include the Swift modules. If the compiler is built without the Swift modules, the corresponding features, e.g. optimizations, are simply not available.

### Build modes

There are four build modes, which can be selected with the `--bootstrapping=<mode>` build-script option:

- `off`: the Swift code is not included in the tools.
- `hosttools`: the Swift code is built with a pre-installed Swift toolchain, using a `swiftc` which is expected to be in the command search path. This mode is the preferred way to build for local development, because it is the fastest way to build. It requires a 5.5 (or newer) swift toolchain to be installed on the host system.
- `bootstrapping`: The compiler is built with a two-stage bootstrapping process. This is the preferred mode if no swift toolchain is available on the system or if the build should not depend on any pre-installed toolchain.
- `bootstrapping-with-hostlibs`: This mode is only available on macOS. It's similar to `bootstrapping`, but links the compiler against the host system swift libraries instead of the built libraries. The build is faster than `bootstrapping` because only the swiftmodule files of the bootstrapping libraries have to be built.

The bootstrapping mode is cached in the `CMakeCache.txt` file in the Swift build directory. When building with a different mode, the `CMakeCache.txt` has to be deleted.

### Bootstrapping

The bootstrapping process is completely implemented with CMake dependencies. The build-script is not required to build the whole bootstrapping chain. For exam-

ple, a `ninja swift-frontend` invocation builds all the required bootstrapping steps required for the final `swift-frontend`.

Bootstrapping involves the following steps:

**1. The level-0 compiler** The build outputs of level-0 are stored in the `bootstrapping0` directory under the main build directory.

In this first step `swift-frontend` is built, but without the Swift modules. When more optimizations are migrated from C++ to Swift, this compiler will produce worse code than the final compiler.

**2. The level-0 library** With the compiler from step 1, a minimal subset of the standard library is built in `bootstrapping0/lib/swift`. The subset contains the swift core library `libswiftCore` and some other required libraries, e.g. `libswiftOnoneSupport` in case of a debug build.

These libraries will be less optimized than the final library build.

This step is skipped when the build mode is `bootstrapping-with-hostlibs`.

**3. The level-1 Swift modules** The build outputs of level-1 are stored in the `bootstrapping1` directory under the main build directory.

The Swift modules are built using the level-0 compiler and standard library from step 2 - or the OS libraries in case of `bootstrapping-with-hostlibs`.

**4. The level-1 compiler** In this step, the level-1 `swift-frontend` is built which includes the Swift modules from step 3. This compiler already produces the exact same code as the final compiler, but might run slower, because its Swift modules are not optimized as good as in the final build.

Unless the build mode is `bootstrapping-with-hostlibs`, the level-1 compiler dynamically links against the level-1 libraries. This is specified with the `RPATH` setting in the executable.

**5. The level-1 library** Like in step 2, a minimal subset of the standard library is built, using the level-1 compiler from step 4.

In this step, the build-system redirects the compiler's dynamic library path to the level-0 library (by setting `DY/LD_LIBRARY_PATH` in `SwiftSource.cmake:_compile_swift_files`). This is needed because the level-1 libraries are not built, yet.

This step is skipped when the build mode is `bootstrapping-with-hostlibs`.

**6. The final Swift modules** The final Swift modules are built with the level-1 compiler and standard library from step 5 - or the OS libraries in case of `bootstrapping-with-hostlibs`.

**7. The final compiler** Now the final `swift-frontend` can be built, linking the final Swift modules from step 6.

**8. The final standard library** With the final compiler from step 7, the final and full standard library is built. This library should be binary equivalent to the level-1 library.

Again, unless the build mode is `bootstrapping-with-hostlibs`, for building the standard library, the build system redirects the compiler's dynamic library path to the level-1 library.

## Using Swift in the compiler

To include Swift modules in a tool, `initializeSwiftModules()` must be called at the start of the tool. This must be done before any Swift code is called. ideally, it's done at the start of the tool, e.g. at the place where `INITIALIZE_LLVM()` is called.

## SIL

The design of the Swift SIL matches very closely the design on the C++ side. For example, there are functions, basic blocks, instructions, SIL values, etc.

Though, there are some small deviations from the C++ SIL design. Either due to the nature of the Swift language (e.g. the SIL `Value` is a protocol, not a class), or improvements, which could be done in C++ as well.

Bridging SIL between C++ and Swift is toll-free, i.e. does not involve any "conversion" between C++ and Swift SIL.

### The bridging layer

The bridging layer is a small interface layer which enables calling into the SIL C++ API from the Swift side. Currently the bridging layer is implemented in C using C interop. In future it can be replaced by a C++ implementation by using C++ interop, which will further simplify the bridging layer or make it completely obsolete. But this is an implementation detail and does not affect the API of Swift SIL.

The bridging layer consists of the C header file `SILBridging.h` and its implementation file `SILBridging.cpp`. The header file contains all the bridging functions and some C data structures like `BridgedStringRef` (once we use C++ interop, those C data structures are not required anymore and can be removed).

### SIL C++ objects in Swift

The core SIL C++ classes have corresponding classes in Swift, for example `Function`, `BasicBlock` or all the instruction classes.

This makes the SIL API easy to use and it allows to program in a "Swifty" style. For example one can write

```
for inst in block.instructions {
  if let cfi = inst as? CondFailInst {
    // ...
  }
}
```

Bridging SIL classes is implemented by including a two word Swift object header (`SwiftObjectHeader`) in the C++ definition of a class, like in `SILFunction`, `SILBasicBlock` or `SILNode`. This enables to use SIL objects on both, the C++ and the Swift, side.

The Swift class metatypes are "registered" by `registerClass()`, called from `initializeSwiftModules()`. On the C++ side, they are stored in static global variables (see `registerBridgedClass()`) and then used to initialize the object headers in the class constructors.

The reference counts in the object header are initialized to "immortal", which let's all ARC operations on SIL objects be no-ops.

The Swift classes don't define any stored properties, because those would overlap data fields in the C++ classes. Instead, data fields are accessed via computed properties, which call bridging functions to retrieve the actual data values.

### Lazy implementation

In the current state the SIL functionality and API is not completely implemented, yet. For example, not all instruction classes have a corresponding class in Swift. Whenever a new Swift optimization needs a specific SIL feature, like an instruction, a Builder-function or an accessor to a data field, it's easy to add the missing parts.

For example, to add a new instruction class:

- replace the macro which defines the instruction in `SILNodes.def` with a `BRIDGED-*` macro
- add the instruction class in `Instruction.swift`
- register the class in `registerSILClasses()`
- if needed, add bridging functions to access the instruction's data fields.

No yet implemented instruction classes are mapped to a "placeholder" instruction, e.g `UnimplementedInstruction`. This ensures that optimizations can process any kind of SIL, even if some instructions don't have a representation in Swift yet.

## The Optimizer

Similar to SIL, the optimizer also uses a small bridging layer (`OptimizerBridging.h`). Passes are registered in `registerSwiftPasses()`, called from `initializeSwiftModules()`. The C++ `PassManager` can then call a Swift pass like any other `SILFunctionTransform` pass.

To add a new function pass:

- add a `SWIFT_FUNCTION_PASS` entry in `Passes.def`
- create a new Swift file in `SwiftCompilerSources/Optimizer/FunctionPasses`
- add a `FunctionPass` global
- register the pass in `registerSwiftPasses()`

All SIL modifications, which a pass can do, are going through the `PassContext` - the second parameter of the pass run-function. In other words, the context is the central place to make modifications. This enables automatic change notifications to the pass manager. Also, it makes it easier to build a concurrent pass manager in future.

Currently it is not possible to implement mandatory Swift passes, because this would break tools which compile Swift code but don't link the Swift modules, like the bootstrapping level-0 compiler.

### Instruction Passes

In addition to function passes, it's possible to define Instruction passes. Instruction passes are invoked from `SILCombine` (in the C++ SILOptimizer) and correspond to a visit-function in `SILCombine`.

With instruction passes it's possible to implement small peephole optimizations for certain instruction classes.

To add a new instruction pass:

- add a `SWIFT_INSTRUCTION_PASS` entry in `Passes.def`
- create a new Swift file in `SwiftCompilerSources/Optimizer/InstructionPasses`
- add an `InstructionPass` global
- register the pass in `registerSwiftPasses()`
- if this pass replaces an existing `SILCombiner` visit function, remove the old visit function

## Performance

Performance is very important, because compile time is critical. Some performance considerations:

- Memory is managed on the C++ side. On the Swift side, SIL objects are treated as "immortal" objects, which avoids (most of) ARC overhead. ARC runtime functions are still being called, but no atomic reference counting

operations are done. In future we could add a compiler feature to mark classes as immortal to avoid the runtime calls at all.

- Minimizing memory allocations by using data structures which are malloc-free, e.g. `Stack`
- The Swift modules are compiled with `-cross-module-optimization`. This enables the compiler to optimize the Swift code across module boundaries.