1
Introduction

Written by Ray Fix

In 2010, Chris Lattner typed `mkdir shiny` on his laptop, and what would ultimately become the Swift language was born. Shiny started as a personal project he worked on during evenings and weekends.

The project had many ambitious goals. Lattner has mentioned some of them in interviews and podcasts, including:

- Adopting modern language features that allow new programming paradigms.
- Using automatic memory management to avoid the overhead of garbage collection.
- Defining as much of the language as possible in the library instead of the compiler.
- Making the language defaults safe to avoid costly undefined behavior.
- Making it easy to learn for beginners.

But perhaps most of all, he wanted this new language to be “real”. Using it should feel like using a scripting language but be suitable for everything from app development to system-level programming. Half-jokingly, he has called this, “world domination”.

Swift released

Shiny transitioned to Swift and officially became real in the summer of 2014 at Apple’s Worldwide Developer Conference (WWDC). It was real in the sense that, out of the box, it had full interoperability with Objective-C and could leverage battle-tested frameworks such as UIKit for building apps. The iOS community embraced it quickly and wholeheartedly, with little exception.

The following year’s WWDC came with another seismic announcement. Swift would become an open-source project and expand beyond Apple platforms. Apple made good on that promise by the end of the year, setting up https://swift.org/, a website to find toolchain downloads for platforms such as

Linux and ultimately Windows as well as discussion and links to complete source code with history hosted at GitHub (https://github.com/apple/swift). Here, Apple and the community would continue to develop Swift in the open.

Swift.org established a language evolution process in which the community could propose and implement changes to the language. To date, 311 change proposals have been made, all of which (accepted or not) have pushed the language forward.

Easy onboarding

It was imperative to the design of Swift that this one-liner compiled and run as a valid program:

```
print("Hello, world!")
```

COPY

Despite being a strongly typed language that can catch all kinds of errors at compile-time, you can write programs like this. Robust type inference and lack of boilerplate make it pleasant to use and feel almost like a scripting language.

Swift embraces the philosophy of progressive disclosure, which means you're exposed to only the language complexity you need. You can learn about things like modules, access control, objects, static methods, protocols, generics and escaped characters as they become necessary to what you're trying to do.

Additionally, Swift allows this to work out of the box:

```
let pointer = malloc(100)
defer {
  free(pointer)
}
// work with raw memory
```

COPY

Swift strives to be a "language without borders" or "infinitely hackable" so it can support seamless interoperability between low-level C and scripting environments, such as Python. Interoperability with other environments continues to be a theme as the language evolves. You'll learn about interoperability with Objective-C in Chapter 12, "Objective-C Interoperability".

Multi-paradigm

Apple demonstrated Swift's modern language features at that first WWDC. In the first live demo, they functionally mapped strings to SpriteKit images. In addition to handling the intricacies of giant, legacy object-oriented frameworks, such as UIKit, CoreData and SpriteKit, Swift can also speak

functionally with sequences of values using `map`, `reduce` and `filter`. Chapter 10, “Higher-Order Functions”, covers these in detail.

Consider the problem of a sequence of numbers you want to add so you can compare using imperative and functional styles.

You can follow along with the starter playground **Multi-paradigm.playground**. This includes an `example()` function that scopes the example so you can use the same variable names repeatedly in the same playground.

Start by adding and running this:

```
let numbers = [1, 2, 4, 10, -1, 2, -10]

example("imperative") {
  var total = 0
  for value in numbers {
    total += value
  }
  print(total)
}
```

As the example’s title suggests, this adds the numbers and stores the result in the mutable variable `total` in an imperative style.

Now, add a functional version:

```
example("functional") {
  let total = numbers.reduce(0, +)
  print(total)
}
```

This functional version is intoxicatingly simple. The method is descriptive, and it allows `total` to be an immutable value that you don’t need to worry about changing later in your code. Experience suggests going functional can be beneficial for your code. You’ll learn more about these techniques in Chapter 10, “Higher-Order Functions”.

Suppose the task is to add a number sequence from left to right but to stop when a negative number appears. One possible, functional version might look like this:

```
example("functional, early-exit") {
  let total = numbers.reduce((accumulating: true, total: 0))
    { (state, value) in
      if state.accumulating && value >= 0 {
        return (accumulating: true, state.total + value)
      }
      else {
        return (accumulating: false, state.total)
      }
```

```
    }.total
  print(total)
}
```

This code is more complex but calls the same `reduce` function and uses a tuple to control whether values accumulate and keep the running total. It works, although the compiler needs to work hard to figure out that an early exit is possible.

Although you can solve this problem functionally in other ways (such as finding the sub-sequence and then summing), you can do it in a straightforward way imperatively:

```
example("imperative, early-exit") {
  var total = 0
  for value in numbers {
    guard value >= 0 else { break }
    total += value
  }
  print(total)
}
```

COPY

This code is easy to follow because it more directly describes the problem statement. It also maps directly to actual computer hardware so the optimizer doesn't have to work nearly as hard.

One downside of the above code is that `total` leaks out as a mutable variable. However, thanks to Swift's mutation model, you can fix that:

```
example("imperative, early-exit with just-in-time mutability") {
  let total: Int = {
    // same-old imperative code
    var total = 0
    for value in numbers {
      guard value >= 0 else { break }
      total += value
    }
    return total
  }()
  print(total)
}
```

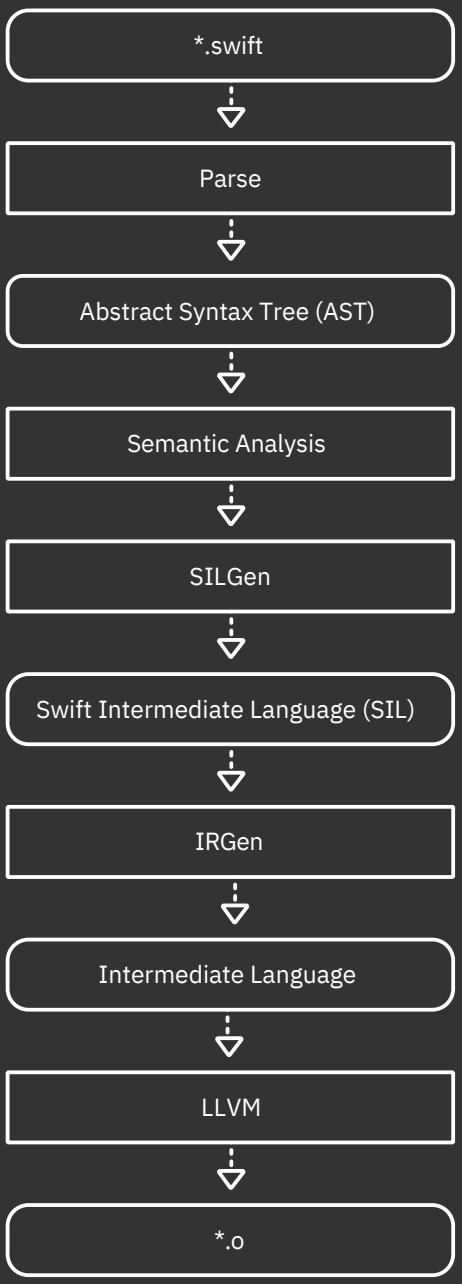
COPY

This code wraps the imperative code in a closure and calls it to assign the outer immutable `total`. In this way, Swift gives you "mutation when you need it." It lets you express algorithms with local mutation when that's the most natural solution.

The Swift compiler

Central to the Swift toolchain is the Swift compiler. It's responsible for turning source code into object code you can link into an executable. It runs on the

LLVM compiler infrastructure, and the data flow looks like this:



The Swift toolchain pipeline

The process of taking a high-level language such as Swift and transforming it into machine code that can run efficiently on actual hardware is called **lowering**. The rounded rectangles shown above are data that are inputs or outputs of the phases represented by rectangles. It's worth understanding each one of the steps from a high level:

1. **Parse**: Swift source code is first parsed into tokens and put into an **abstract syntax tree** or **AST**. You can think of this being a tree in which each expression is a node. The nodes also hold source location information so, if an error is detected, the node can tell you exactly where the problem occurred.
2. **Semantic Analysis (Sema)**: In this step, the compiler uses the AST to analyze your program's meaning. This is where **type checking** occurs. It passes the type-checked AST to the **SILGen** phase.

3. **SILGen**: This phase departs from previous compiler pipelines such as **Clang**, which didn't have this step. The AST gets lowered into **Swift Intermediate Language** (**SIL**). SIL contains **basic blocks** of computation and understands Swift Types, reference counting and dispatch rules. There are two flavors of SIL: raw and canonical. Canonical SIL results from raw SIL run through a minimum set of optimization passes (even when all optimizations are turned off). SIL also contains source location information so it can produce meaningful errors.
4. **IRGen**: This tool lowers SIL to LLVM's intermediate representation. At this point, the instructions are no longer Swift specific. (Every LLVM-based uses this representation.) IR is still quite abstract. Like SIL, IR is in Static single assignment (SSA) form. It models machines as having an unlimited number of registers, making it easier to find optimizations. It doesn't know anything about Swift types.
5. **LLVM**: This final step optimizes the IR and lowers it to machine instructions for a particular platform. Backends (which output machine instructions) include ARM, x86, Wasm and more.

The diagram above shows how the Swift compiler generates object code. Other tools, such as source code formatters, refactoring tools, documentation generators and syntax highlighters can tap into the intermediate results, such as the AST, making the final results more robust and consistent.

> **Historical note**: Before Apple adopted LLVM and Clang for Xcode's compiler technology, the syntax highlighting, document generation, debugging and compiling all used different parsers. Most of the time, this worked fine. But things could also get weird if they got out of sync.

The magic of SIL

The idea of creating an intermediate language that keeps all the type semantics of the source language was new with the development of Swift. Unlike other LLVM compilers that need to take an extremely circuitous route to show specific diagnostics and perform higher-level optimizations, SILGen can produce them directly in a testable way.

Overflow detection

Check out the power of SIL in action. Consider the following playground error:

```
let x = Int8(15)+115   ⊗ Arithmetic operation '15 + 115' (on type 'Int8') results in an overflow
```

Thanks to a SILGen pass, the compiler statically analyzes (checks at compile-time) your source and sees that the number 130 can't fit in an Int8 that can go

up to only 127.

Definite initialization

Swift is a safe language that by default makes it hard to access uninitialized memory. SILGen provides guarantee through a checking process called definite initialization. Consider this example:

```
final class Printer {
  var value: Int
  init(value: Int) { self.value = value }
  func print() { Swift.print(value) }
}

func printTest() {
  var printer: Printer
  if .random() {
    printer = Printer(value: 1)
  }
  else {
    printer = Printer(value: 2)
  }
  printer.print()
}

printTest()
```

This code compiles and runs fine. But if you comment out the `else` clause, the compiler correctly flags an error (Variable ‘printer’ used before being initialized) thanks to the SIL. This error is possible because SIL understands the semantics of method calls to `Printer`.

Allocation and devirtualization

SILGen helps with the optimization of allocations and method calls. Take the following code and put it in a file called **magic.swift**, using your favorite plain text editor:

```
class Magic {
  func number() -> Int { return 0 }
}

final class SpecialMagic: Magic {
  override func number() -> Int { return 42 }
}

public var number: Int = -1

func magicTest() {
  let specialMagic = SpecialMagic()
  let magic: Magic = specialMagic
  number = magic.number()
}
```

This code is probably the most contrived example you’ll ever see for setting a

number. In the function `magicTest`, you create a `SpecialMagic` type and then assign it to a base class reference and call `number()` to set the global number. Conceptually, it uses the class's virtual table to look up the correct function, which returns the value 42.

Raw SIL

From a terminal window, change to the source directory where **magic.swift** lives and run this command:

```
swiftc -O -emit-silgen magic.swift > magic.rawsil
```

COPY

This runs the Swift compiler with optimization and creates raw SIL, outputting it to the file **magic.rawsil**.

Take a deep breath, don't panic and open **magic.rawsil** in your text editor. If you scroll down to the bottom, you find this definition of the function `magicTest()`:

```
// magicTest()
sil hidden [ossa] @$s5magic0A4TestyyF : $@convention(thin) () -> ()
{
bb0:
  %0 = global_addr @$s5magic6numberSivp : $*Int   // user: %14
  %1 = metatype $@thick SpecialMagic.Type         // user: %3
  // function_ref SpecialMagic.__allocating_init()
  %2 = function_ref @$s5magic12SpecialMagicCACycfC :
$@convention(method) (@thick SpecialMagic.Type) -> @owned
SpecialMagic // user: %3
  %3 = apply %2(%1) : $@convention(method) (@thick
SpecialMagic.Type) -> @owned SpecialMagic // users: %18, %5, %4
  debug_value %3 : $SpecialMagic, let, name "specialMagic" // id:
%4
  %5 = begin_borrow %3 : $SpecialMagic            // users: %9, %6
  %6 = copy_value %5 : $SpecialMagic              // user: %7
  %7 = upcast %6 : $SpecialMagic to $Magic        // users: %17,
%10, %8
  debug_value %7 : $Magic, let, name "magic"      // id: %8
  end_borrow %5 : $SpecialMagic                   // id: %9
  %10 = begin_borrow %7 : $Magic                  // users: %13,
%12, %11
  %11 = class_method %10 : $Magic, #Magic.number : (Magic) -> () ->
Int, $@convention(method) (@guaranteed Magic) -> Int // user: %12
  %12 = apply %11(%10) : $@convention(method) (@guaranteed Magic) -
> Int // user: %15
  end_borrow %10 : $Magic                         // id: %13
  %14 = begin_access [modify] [dynamic] %0 : $*Int // users: %16,
%15
  assign %12 to %14 : $*Int                       // id: %15
  end_access %14 : $*Int                          // id: %16
  destroy_value %7 : $Magic                       // id: %17
  destroy_value %3 : $SpecialMagic                // id: %18
  %19 = tuple ()                                  // user: %20
  return %19 : $()                                // id: %20
} // end sil function '$s5magic0A4TestyyF'
```

COPY

This excerpt is the SIL definition of the three-line function `magicTest()`. The label `bb0` stands for basic block 0 and is a unit of computation. (If you had an `if`/`else` statement, there would be two basic blocks, `bb1` and `bb2` created for each possible path.) The `%1`, `%2`, etc. values are virtual registers. SIL is in Single Static Assignment form so registers are unlimited and never reused. There are many more small details that aren't important to the discussion here. Reading through it, you should roughly see how it's allocating, assigning, calling and deallocating the objects. This expresses the full semantics of the Swift language.

Canonical SIL

Canonical SIL includes optimizations, including a minimum set of optimization passes when optimization is turned off with `-Onone`. Run this Terminal command:

```
swiftc -O -emit-sil magic.swift > magic.sil
```

COPY

This command creates the file **magic.sil**, which contains canonical SIL. Scroll toward the end of the file to find `magicTest()`:

```
// magicTest()
sil hidden @$s5magic0A4TestyyF : $@convention(thin) () -> () {
bb0:
  %0 = global_addr @$s5magic6numberSivp : $*Int   // user: %3
  %1 = integer_literal $Builtin.Int64, 42         // user: %2
  %2 = struct $Int (%1 : $Builtin.Int64)          // user: %4
  %3 = begin_access [modify] [dynamic] [no_nested_conflict] %0 :
$*Int // users: %4, %5
  store %2 to %3 : $*Int                          // id: %4
  end_access %3 : $*Int                           // id: %5
  %6 = tuple ()                                   // user: %7
  return %6 : $()                                 // id: %7
} // end sil function '$s5magic0A4TestyyF'
```

COPY

This excerpt is a lot more concise than the raw SIL, even though it represents the same thing. The main work is to store the integer literal 42 into a global address location `store %2 to %3 : $*Int`. No classes are being initialized or de-initialized, nor are any virtual methods being called. When you hear that `structures` use the stack and `classes` use the heap, keep in mind this is a generalization.

In Swift, everything starts off being initialized on the heap, and a SIL analysis can move the allocation to the stack or even get rid of it altogether. Virtual function calls can also be devirtualized through the optimization process and called directly or even inlined.

Implementing a language feature

Swift pushes the implementation of as many features as possible from the compiler into the library. You might be aware, for example, that `Optional` is just a generic enumeration. The truth is that most of the fundamental types are part of the standard library and not baked into the compiler. This includes `Bool`, `Int`, `Double`, `String`, `Array`, `Set`, `Dictionary`, `Range` and many more. In an October 2020 Lex Fridman interview, Lattner said he regards this kind of expressive library design as *the most beautiful* feature of a programming language.

A great way to learn about some of the more esoteric features of Swift or gain a better appreciation of some of the basic ones is to do this yourself — to build a language-like feature. You'll do that now.

Building `ifelse`

For this coding experiment, you'll implement an `ifelse()` statement like the statistical programming language **R** uses. The function looks like this:

```
ifelse(condition, valueTrue, valueFalse)
```

COPY

It does the same thing as the Swift ternary operator `condition ? valueTrue : valueFalse`, which some don't like because of aesthetic objections.

Start by typing this into a playground:

```
func ifelse(condition: Bool,
            valueTrue: Int,
            valueFalse: Int) -> Int {
  if condition {
    return valueTrue
  } else {
    return valueFalse
  }
}
let value = ifelse(condition: Bool.random(),
                   valueTrue: 100,
                   valueFalse: 0)
```

COPY

What's wrong with this solution? Maybe nothing. If it solves your problem and you're working with only `Int`, this might even be a good place to stop. But because you're trying to make a general-purpose language feature for everyone, you can make several improvements. First, refine the interface a little:

```
func ifelse(_ condition: Bool,
            _ valueTrue: Int,
            _ valueFalse: Int) -> Int {
  condition ? valueTrue : valueFalse
}
```

COPY

```
let value = ifelse(.random(), 100, 0)
```

For a language construct that's going to be used often, removing the argument labels makes sense. The wildcard label `_` gives you the ability to remove them. For brevity, implement the feature in terms of the less verbose ternary operator. (You might wonder why you shouldn't use the camel-case name `ifElse`. There's precedent for keywords being simple concatenations, such as `typealias` and `associatedtype`, so stay with the original **R** language naming.)

The next obvious problem is that this works only for `Int` types. You could replace it with a lot of overloads for the important types you want:

COPY

```
func ifelse(_ condition: Bool,
            _ valueTrue: Int,
            _ valueFalse: Int) -> Int {
  condition ? valueTrue : valueFalse
}
func ifelse(_ condition: Bool,
            _ valueTrue: String,
            _ valueFalse: String) -> String {
  condition ? valueTrue : valueFalse
}
func ifelse(_ condition: Bool,
            _ valueTrue: Double,
            _ valueFalse: Double) -> Double {
  condition ? valueTrue : valueFalse
}
func ifelse(_ condition: Bool,
            _ valueTrue: [Int],
            _ valueFalse: [Int]) -> [Int] {
  condition ? valueTrue : valueFalse
}
```

It's easy to see this doesn't scale. As soon as you think you're done, there's another type your users want support for. And each overload repeats the implementation, which is not great.

As an alternative, you could use the type `Any`, a type-erased, stand-in for any Swift type:

COPY

```
func ifelse(_ condition: Bool,
            _ valueTrue: Any,
            _ valueFalse: Any) -> Any {
  condition ? valueTrue : valueFalse
}

let value = ifelse(.random(), 100, 0) as! Int
```

This code works for any type, but there's an important caveat you have to cast back to the original type you want. Using the `Any` type doesn't protect you from a situation like this where you mix types:

```
let value = ifelse(.random(), "100", 0) as! Int
```

This statement might work in testing but crash in production if the random number comes up true. `Any` is super versatile but also error-prone to use.

A better answer, as you might have guessed, is to use generics. Change the code to this:

```
func ifelse<V>(_ condition: Bool,
               _ valueTrue: V,
               _ valueFalse: V) -> V {
  condition ? valueTrue : valueFalse
}

// let value = ifelse(.random(), "100", 0)  // doesn't compile
anymore
let value = ifelse(.random(), 100, 0)
```

This design both preserves type information and constrains the arguments to be the same type as the return type. Generics are such an essential part of the Swift language that Chapter 4, "Generics" is dedicated to them. You'll use generics throughout the book.

> **Note**: The Swift standard library uses generics extensively to eliminate code duplication, as in the example above. In some cases where the generic system is not *yet* strong enough, the library uses a python script, gyb (or generate-your-boilerplate), to generate the code for a family of types.

Deferring execution

The feature is looking good, but it's still not done. Consider this usage:

```
func expensiveValue1() -> Int {
  print("side-effect-1")
  return 2
}

func expensiveValue2() -> Int {
  print("side-effect-2")
  return 1729
}

let taxicab = ifelse(.random(),
                     expensiveValue1(),
                     expensiveValue2())
```

If you run this, you see *both* functions are always called. As a language feature, you would hope that only the expression you use gets evaluated. You can fix this by passing a closure that defers execution:

```
func ifelse<V>(_ condition: Bool,
               _ valueTrue: () -> V,
               _ valueFalse: () -> V) -> V {
  condition ? valueTrue() : valueFalse()
}
```

This code defers the execution but changes how you need to call the function. Now, you have to call it like this:

```
let value = ifelse(.random(), { 100 }, { 0 })

let taxicab = ifelse(.random(),
                     { expensiveValue1() },
                     { expensiveValue2() })
```

Only one function gets called, but having to wrap your arguments in a closure is pretty annoying. Fortunately, Swift has a way to fix it:

```
func ifelse<V>(_ condition: Bool,
               _ valueTrue: @autoclosure () -> V,
               _ valueFalse: @autoclosure () -> V) -> V {
  condition ? valueTrue() : valueFalse()
}

let value = ifelse(.random(), 100, 0 )

let taxicab = ifelse(.random(),
                     expensiveValue1(),
                     expensiveValue2())
```

Decorating a parameter type with `@autoclosure` causes the compiler to wrap arguments in a closure automatically. This change restores the call sites to what they used to be and still defers execution so only the used argument evaluates.

Using expressions that can fail

Things are going well, but there's still one more small problem. What if you want to use expressions that can fail?

Consider the following example:

```
func expensiveFailingValue1() throws -> Int {
  print("side-effect-1")
  return 2
}

func expensiveFailingValue2() throws -> Int {
  print("side-effect-2")
  return 1729
}

let failableTaxicab = ifelse(.random(),
                             try expensiveFailingValue1(),
```

```
                                try expensiveFailingValue2())
```

This fails to compile because the autoclosures aren't expecting a throwing closure. Without any special help from the compiler, you might think to solve it by creating another function version like this:

```
func ifelseThrows<V>(_ condition: Bool,
                     _ valueTrue: @autoclosure () throws -> V,
                     _ valueFalse: @autoclosure () throws -> V) throws ->
V {
  condition ? try valueTrue() : try valueFalse()
}

let taxicab2 = try ifelseThrows(.random(),
                                try expensiveFailingValue1(),
                                try expensiveFailingValue2())
```

This code works, but the situation is worse than initially described. Suppose only the first expression throws or suppose only the second throws. Do you need to make four versions of the same function to handle all the cases? Because the keyword `throws` does not figure into the signature of a Swift function, you would need to have four flavors of `ifelse`, all with slightly different names.

Fortunately, there's a better way. You can write one version of the function that handles all these cases:

```
func ifelse<V>(_ condition: Bool,
               _ valueTrue: @autoclosure () throws -> V,
               _ valueFalse: @autoclosure () throws -> V) rethrows
-> V {
  condition ? try valueTrue() : try valueFalse()
}
```

The key is using `rethrows`. Rethrows propagates the error of any failing closure to the caller. If none of the closure parameters throw, it deduces the function is non-throwing and doesn't need to be marked with `try`.

With this single version, all these variants work:

```
let value = ifelse(.random(), 100, 0 )
let taxicab = ifelse(.random(),
                     expensiveValue1(),
                     expensiveValue2())
let taxicab2 = try ifelse(.random(),
                          try expensiveFailingValue1(),
                          try expensiveFailingValue2())
let taxicab3 = try ifelse(.random(),
                           expensiveValue1(),
                           try expensiveFailingValue2())
let taxicab4 = try ifelse(.random(),
                          try expensiveFailingValue1(),
                          expensiveValue2())
```

You're getting close to finishing `ifelse`. You don't want to pay the cost of an extra layer of abstraction, and the implementation will never change, so it makes sense to mark the function `@inlinable`. This added keyword hints to the compiler that the body of the method should be directly included in the client code without the overhead of calling a function.

```
@inlinable
func ifelse<V>(_ condition: Bool,
               _ valueTrue: @autoclosure () throws -> V,
               _ valueFalse: @autoclosure () throws -> V) rethrows
-> V {
  condition ? try valueTrue() : try valueFalse()
}
```

Note: There are stronger forms of @inlinable available privately. You'll see these if you browse the Swift source. One such attribute is @_transparent, which always "sees through" to the underlying implementation. It will inline even with `-Onone` and not include a stack frame when debugging. Check out the details here: https://github.com/apple/swift/blob/main/docs/TransparentAttr.md

Performance

One of the cool things about writing programs with an optimizing compiler is that the abstraction cost of making code clear and maintainable is often nothing or close to nothing.

To look at how you did here, put this code into a text file called **ifelse.swift**:

```
@inlinable
func ifelse<V>(_ condition: Bool,
               _ valueTrue: @autoclosure () throws -> V,
               _ valueFalse: @autoclosure () throws -> V) rethrows
-> V {
  condition ? try valueTrue() : try valueFalse()
}

func ifelseTest1() -> Int {
  if .random() {
      return 100
  } else {
      return 200
  }
}

func ifelseTest2() -> Int {
  Bool.random() ? 300 : 400
}

func ifelseTest3() -> Int {
  ifelse(.random(), 500, 600)
}
```

Take this code and run the compiler on it directly with this command:

```
swiftc -O -emit-assembly ifelse.swift > ifelse.asm
```

COPY

Take another deep breath and open the assembly file. Keep in mind these assembly files contain a ton of boilerplate ceremony around calling conventions and entry points. Don't let that discourage you from looking. Trimming off the unneeded stuff, here are the good parts:

```
_$s6ifelse0A5Test1SiyF:
    :
    callq   _swift_stdlib_random
    testl   $131072, -8(%rbp)
    movl    $100, %ecx
    movl    $200, %eax
    cmoveq  %rcx, %rax
    :

_$s6ifelse0A5Test2SiyF:
    :
    callq   _swift_stdlib_random
    testl   $131072, -8(%rbp)
    movl    $300, %ecx
    movl    $400, %eax
    cmoveq  %rcx, %rax
    :

_$s6ifelse0A5Test3SiyF:
    :
    callq   _swift_stdlib_random
    testl   $131072, -8(%rbp)
    movl    $500, %ecx
    movl    $600, %eax
    cmoveq  %rcx, %rax
    :
```

COPY

These are the assembly instructions for your three test functions. It might look like gibberish to you. The important thing is that this is the *same* gibberish for `ifelseTest1()`, `ifelseTest2()` and `ifelseTest3()`. In other words, there's zero abstraction penalty for the three ways of writing the code. Choose what looks most beautiful to you.

Now, demystifying the above assembly, the `callq` instruction calls the function to get a random number. Next, the `testl` instruction gets the random number return value (located at the address pointed to by the 64-bit base pointer - 8). It checks this against 131072, which is 0x20000 or the 17th bit. If you look at the Swift source for `Bool.random`, you find this:

```
@inlinable
public static func random<T: RandomNumberGenerator>(
  using generator: inout T
) -> Bool {
  return (generator.next() >> 17) & 1 == 0
```

COPY

```
}
```

That explains the 131072 mystery: It's shifting over the 17th bit, masking it and testing it all in one instruction. Next, the two possible outcome values of the function are moved (using the `movl` instruction) into the registers cx and ax. The prefix "e" stands for extended 32-bit versions of the registers. The rest of the bits are zero-extended to fill all 64-bits. Finally, the "conditional move if equal" or `cmoveq` instruction uses the result of the earlier test instruction to move the cx register to the ax register. The prefix r on rcx and rax indicates you use the full 64-bits of the registers.

> **Note**: The **mangled** symbol **_$s6ifelse0A5Test1SiyF:** is the unique symbol name for the `ifelse.ifelseTest1() -> Int`. (The leading "ifelse." is the module name, or in this case, the filename.) The linker needs short, guaranteed unique names for all the external symbols in your program. You can find the specification for mangling here: https://github.com/apple/swift/blob/main/docs/ABI/Mangling.rst. You can also run the command line tool `swift-demangle` found in `/Library/Developer/CommandLineTools/usr/bin/`. For example, `swift-demangle _\$s6ifelseAAyxSb_xyKXKxyKXKtKlF` corresponds to the symbol `ifelse.ifelse<A>(Swift.Bool, @autoclosure () throws -> A, @autoclosure () throws -> A) throws -> A`.

That completes the discussion and implementation of `ifelse`. It would be best if you asked the question posed by Swift core team member John McCall: "Is this an abstraction that pays for itself?" In this case, probably not. The ternary operator already exists, which does essentially the same thing. Nevertheless, going through this example hopefully reminded you of some of the capabilities Swift offers when building language-like features as part of a library.

Key points

This chapter talked about some of the motivations for building the Swift language and how Swift's libraries and compiler work together to make powerful abstractions. Here are some key takeaways:

- Swift is a multi-paradigm language that supports many programming styles, including imperative, functional, object-oriented, protocol-oriented and generic paradigms.
- Swift aims to pick reasonable defaults, making undefined behavior hard to trigger.
- Swift embraces the idea of progressive disclosure. You only need to learn about more advanced language features when you need them.

- Swift is a general-purpose programming language that features a powerful type system and type inference.
- Much of Swift is defined in its expressive, standard library and not as part of the compiler.
- The Swift compiler phases are parse, semantic analysis, SILGen, IRGen and LLVM.
- Source location information resides in AST and SIL, making better error reporting possible.
- SIL is a low-level description using basic blocks of instructions written in SSA form. It understands the semantics of Swift types, which enables many optimizations not possible with pure LLVM IR.
- SIL helps support definite initialization, memory allocation optimizations and devirtualization.
- `Any` is the ultimate type-erasure in Swift, but it can be error-prone to use. Generics are usually a better alternative.
- Pass a closure as a parameter that returns a value to defer evaluation of the argument until within the body of a function.
- `@autoclosure` is a way to implement short-circuit behavior because it defers execution of expression arguments.
- `rethrow` is a way to propagate errors from closures that might or might not be marked `throws`.
- `@inlinable` hints to the compiler that the instructions for a function should be emitted into the call site.
- Compilers remove much if not all the abstraction costs of your source code. If different source code has the same semantics, the compiler likely will emit identical machine instructions.
- Abstractions should pay for themselves. Think hard before creating new language features.

Where to go from here?

Much of this chapter discussed how the Swift compiler lowers high-level types and statements to efficient machine representation through SIL. Lowering is an intense (and niche) topic. If you're interested in knowing more, check out this slightly old but still quite relevant blog post by compiler engineer Slava Pestov http://bit.ly/slava-types. It's an *extremely* deep dive into Swift types and lowering, so you might want to read the rest of this section on types, protocols and generics before tackling it.

Members of the Swift compiler team, including Chris Lattner, Slava Pestov, Joseph Groff, John McCall and Doug Gregor, appear in LLVM conference talks about compiler implementation. A good one to start with is http://bit.ly/swift-

sil.

Finally, check out the online tool https://godbolt.org, which lets you edit code in the web browser for many different languages (including Swift) and see how they lower. You might want to experiment around with compiler flags `-O`, `-Onone`, `-Ounchecked`. Under **Output settings** in the web interface, you might want to uncheck "Intel asm syntax" to get assembly output like in this chapter.

Mark Complete

2. Types & Mutation v. Introduction

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

Have feedback to share about the online reading experience? If you have feedback about the UI, UX, highlighting, or other features of our online readers, you can send them to the design team with the form below:

Feedback about the UI, UX, or other features of the online reader? Leave them here!

Send Feedback