Learning Swift Language on Ubuntu:
Methods
Subscripts
Inheritance
Initialization

https://github.com/apple/swift
この Git リポジトリで Swift 言語は開発されている。このリポジトリの docs ディレクトリに資料がある。

AST と SIL と IR の取得方法

.swift のコードは、AST (Abstract Syntax Tree) と SIL (Swift Intermediate Language) と LLVM IR (Low Level Virtual Machine Intermediate Representation) をこの順に経てアセンブリになる。それらを得る方法は次のものである。

- `swiftc -dump-parse example.swift` `# AST（構文木）を出力する`
- `swiftc -dump-ast example.swift` `# 型チェック済みの AST を出力する`
- `swiftc -emit-silgen example.swift` `# sil_stage raw を出力する`
- `swiftc -emit-sil example.swift` `# sil_stage canonical を出力する`
- `swiftc -emit-ir example.swift` `# LLVM IR を出力する`
- `swiftc -emit-assembly example.swift` `# アセンブリを出力する`

implementation

```
// methods メソッド
// https://docs.swift.org/swift-book/LanguageGuide/Methods.html

// メソッドは型に属する関数だ。クラス、構造体、列挙型について存在する。
// メソッドには、インスタンスメソッドと型メソッドがある。

// instance method インスタンスメソッド
// 3 個のインスタンスメソッドを持つクラスを定義する。
```

```
class Counter {
    var count = 0  // ストアド変数プロパティ
    /// 1 増加する
    func increment() {
        count += 1
    }
    /// by 増加する
    func increment(by amount: Int) {
        count += amount
    }
    /// 0 にする
    func reset() {
        count = 0
    }
}
let counter = Counter()
print(counter.count)  // 0
counter.increment()
print(counter.count)  // 1
counter.increment(by: 5)
print(counter.count)  // 6
counter.reset()
print(counter.count)  // 0

// In practice, you don't need to write self in your code very often.
// ...
// The main exception to this rule occurs when a parameter name for an
// instance method has the same name as a property of that instance.

// self. が有用な例:
struct Point {
    var x = 0.0, y = 0.0
    /// 点 self が x=x よりも右にあれば真を返す。
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOf(x: 1.0) {
    print("This point is to the right of the line where x == 1.0")
```

```
}

// mutating
struct Point1 {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint1 = Point1(x: 1.0, y: 1.0)  // var にしとかないと mutating member が使えない。
somePoint1.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint1.x), \(somePoint1.y))")  // 3.0 4.0

// mutating methods for enumerations
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
print(ovenLight)  // .low
ovenLight.next()
print(ovenLight)  // .high
ovenLight.next()
print(ovenLight)  // .off

// type method 型メソッド
struct LevelTracker {
    static var highestUnlockedLevel = 1  // このデバイスでアンロックされたレベル
    var currentLevel = 1                 // 各プレイヤーのレベル
    static func unlock(_ level: Int) {
```

```
        if level > highestUnlockedLevel { highestUnlockedLevel = level }
    }
    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
    @discardableResult  // 返り値は無視していい
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
class Player {
    var tracker = LevelTracker()
    let playerName: String
    /// プレイヤーがあるレベルを完了したときに呼ばれる
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)  // 返り値は無視する
    }
    init(name: String) {
        playerName = name
    }
}
var player = Player(name: "Argyrios")
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")  // 1
player.complete(level: 1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")  // 2
player = Player(name: "Beto")       // 同じデバイスを使う別のプレイヤー
if player.tracker.advance(to: 6) {  // レベル 6 に進もうとする
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")  // 実行される
}
```

implementation

```
// subscripts 添字
// https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html

struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")  // 18

for multiplier in 1...9 {
    let table = TimesTable(multiplier: multiplier)
    for index in 1...9 {
        print("\(table[index]) ", terminator: "")
    }
    print()
}

var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
print(numberOfLegs)
numberOfLegs["bird"] = 2
print(numberOfLegs)

var array0 = [1, 2, 3]
print(array0[1])
// print(array0[10])  // runtime error

struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
```

```
    /// row または column が不正なら偽を返す。
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
    func printSelf() {
        for row in 0..<rows {
            for column in 0..<columns {
                print("\(self[row, column]) ", terminator: "")
            }
            print()
        }
    }
}
var matrix = Matrix(rows: 2, columns: 2)
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
matrix.printSelf()
```

implementation

```
// inheritance 継承
// https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html

// クラス B が A を継承したら B は A のサブクラスで A は B のスーパークラスだ。
// サブクラスではスーパークラスのメソッドやプロパティが override されうる。
// 他のどのクラスからも継承していないクラスを base class と呼ぶ。

class Vehicle {
    var currentSpeed = 0.0
```

```
    /// Vehicle オブジェクトを描写する。
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    /// override 用
    func makeNoise() {}
}
let someVehicle = Vehicle()  // initializer syntax
print("Vehicle: \(someVehicle.description)")

class Bicycle: Vehicle {
    var hasBasket = false  // このサブクラスで追加するストアドプロパティ
}
let bicycle = Bicycle()
bicycle.hasBasket = true
bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")

class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")

// overriding
// override キーワードがなく override したときと、あって override してないときはコンパイルエラー。

class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
let train = Train()
train.makeNoise()

// overriding property getters and setters
class Car: Vehicle {
```

```
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")

// overriding property observers
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1  // 速度に応じたギアを選ぶ。
        }
    }
}
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")

// final キーワードで修飾することでメンバないしクラスを継承不可能にできる。
// クラスを final にすると dynamic dispatch から static dispatch になる。
// static dispatch にするためには、サブクラスで final にしたのでは駄目だろうと思う。

class Class0 {
    func f() { print("Class0#f()") }
}
Class0().f()  // -> "Class0#f()"
final class Class1: Class0 {
    final override func f() { print("Class1#f()") }
}
Class1().f()  // -> "Class1#f()"
let class1: Class1 = Class1()
class1.f()    // -> "Class1#f()"
let class0: Class0 = class1
class0.f()    // -> "Class1#f()"
```

implementation

```
// initialization 初期化
// https://docs.swift.org/swift-book/LanguageGuide/Initialization.html

// Swift のイニシャライザ init() は値を返さない。

struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")

// initialization parameters

struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
print(boilingPointOfWater.temperatureInCelsius)
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
print(freezingPointOfWater.temperatureInCelsius)
let bodyTemperature = Celsius(37.0)
print(bodyTemperature.temperatureInCelsius)

struct Color {
// struct Color: CustomStringConvertible {
```

```
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
    /* var description: String {
        let s: String = [red, green, blue].map{ String($0) }.joined(separator: " ")
        return "(\(s))"
    } */
}
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
print(magenta)
let halfGray = Color(white: 0.5)
print(halfGray)

// optional property types
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
cheeseQuestion.response = "Yes, I do like cheese."

let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

```
// default initializer
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
print(item)

class Class0 {
    init(x: Int) { print(x) }
}
// Class0()  // compile-time error; default initializer が作られない。
let _ = Class0(x: 1)

// memberwise initializers for structure types
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)  // implicit memberwise initializer
print(twoByTwo, Size())                       // implicit default initializer もある

// initializer delegation イニシャライザ移譲とは init() の別の実装を応用すること。
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        // イニシャライザ移譲
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
```

```
}
let basicRect = Rect()
print(basicRect)
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
                      size: Size(width: 5.0, height: 5.0))
print(originRect)
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
print(centerRect)

// class inheritance and initialization
// クラスのイニシャライザには、designated initializer と convenience initializer がある。
// designated initializer は、すべてのプロパティを初期化する。
// convenience initializer は convenience 修飾を持ち、designated initializer に移譲する。
// designated initializer は、多くの場合 1 つだが、複数あってもいい。

// two-phase initialization
// クラスの init() はまず自らのプロパティを初期化し、スーパークラスの init() で初期化する。
// 次に、任意のプロパティを任意に変更する。

// safety check 1
// super.init() の前にサブクラスのプロパティがすべて初期化されてなければコンパイルエラー。
// safety check 2
// super.init() の前にスーパークラスのプロパティをどれか初期化したらコンパイルエラー。
// ∵ super.init() で上書きされてしまうかもしれないので。
// safety check 3
// convenicence init() が init() する前にプロパティを初期化したらコンパイルエラー。
// ∵ init() が上書きしてしまうかもしれないので。
// safety check 4
// self と super のプロパティが初期化される第 1 段階以降にのみプロパティやメソッドを使える。
// ∵ まだ初期化されていないものにアクセスしかねないので。

// super.init() より前にサブクラスのプロパティが初期化されてなければならない理由:
// クラス Sub のプロパティ x にアクセスする override func f() がスーパークラスから呼ばれうるから。



// initializer inheritance and overriding
// （暗黙で designated な）default initializer を持つクラス。
class Vehicle {
    var numberOfWheels = 0
```

```
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}
let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")

class Bicycle: Vehicle {
    override init() {  // custom designated initializer
        super.init()
        numberOfWheels = 2
    }
}
let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")

class Hoverboard: Vehicle {
    var color: String
    init(color: String) {
        self.color = color
        // super.init() implicitly called here
    }
    override var description: String {
        return "\(super.description) in a beautiful \(color)"
    }
}
let hoverboard = Hoverboard(color: "silver")
print("Hoverboard: \(hoverboard.description)")

// Note
// You always write the override modifier when overriding a superclass
// designated initializer, even if your subclass' s implementation of the
// initializer is a convenience initializer.
class Bicycle1: Vehicle {
    var color: String
    init(color: String) {
        self.color = color
        super.init()
        numberOfWheels = 2
    }
```

```
    // スーパークラスの init() を間接的に呼ぶので override である。
    // （間接的に呼ばずともスーパークラスの designated イニシャライザなので override。）
    override convenience init() { self.init(color: "silver") }
}
let bicycle1 = Bicycle1()
print("Bicycle1: \(bicycle1.description)")

// As a result, you do not write the override modifier when providing a
// matching implementation of a superclass convenience initializer.
class Bicycle2: Bicycle1 {
    var bgColor: String
    init(color: String, bgColor: String) {
        self.bgColor = bgColor
        super.init(color: color)
    }
    // スーパークラスの init() を間接的にすら呼ばないので override ではない。
    //（というかスーパークラスの convenience イニシャライザなので override ではない。）
    convenience init() { self.init(color: "silver", bgColor: "silver") }
}
let bicycle2 = Bicycle2()
print("Bicycle2: \(bicycle2.description)")

// 間接的に呼ばずともスーパークラスの designated イニシャライザなら override せねばならない:
class Super {
    init(x: Int) { print("Super#init(x: Int)") }
    init(x: String) {}  // 間接的にも呼ばれないスーパークラスの designated イニシャライザ。
}
class Sub: Super {
    init(x: Double) { super.init(x: 0) }
    override convenience init(x: String) { self.init(x: 1.2) }
}
let _ = Sub(x: "")
// override かどうかを考える際には、祖先クラスの convenience イニシャライザは忘れてよさそうだ。
// こういう仕組みである必要性は不明。

// https://stackoverflow.com/questions/24192253/overriding-convenience-init
// スーパークラスのすべての designated イニシャライザを override するとすべての convenience
// イニシャライザは自動的に継承される、らしい。2014 年。
// https://stackoverflow.com/questions/25322421/convenience-init-override
// 2014 年。
```

```
// https://forums.swift.org/t/overriding-convenience-initializers-possible-bug/11274/2
// Remember that convenience initializers are inherited if you implement all
// of the designated initializers from your superclass, or if you don't
// provide any designated initializers (in which case you inherit all
// initializers from your superclass).
// https://teamtreehouse.com/community/help-with-override-in-swift

// すべての designated イニシャライザを override すると convenience イニシャライザは継承される。
class Super1 {
    init()       { print("Super1#init()") }          // designated
    init(x: Int) { print("Super1#init(x: Int)") }    // designated
    convenience init(x: String) { self.init(x: 0) }  // convenience
}
class Sub1: Super1 {
    override init() { super.init() }                 // designated
    override init(x: Int) { super.init(x: x) }       // designated
    // convenience init(x: String) { self.init() }
}
let _ = Sub1(x: "")

// スーパークラスの convenience イニシャライザがサブクラスから呼ばれる可能性はない。
// ∴ そのときは override とはしない。
// スーパークラスの designated イニシャライザはサブクラスから呼ばれるかもしれない。
// ∴ そのときは override とする。

// 普通のメソッドとは異なり、初期化子は継承されない。
// よって、親クラスにある初期化子が子クラスにないことは普通である。
// よって、依存する特命初期化子がないかもしれないから親クラスの利便初期化子は呼べない。
// また同じ理由で、親クラスの特命初期化子をすべて継承すれば利便初期化子もすべて使える。

// オブジェクトには多態性があるから上書きしているメンバが暗黙に適用される。
// 特命初期化子であれ利便初期化子であれ特命初期化子を上書きするならば明示する必要がある。
// 一方で、利便初期化子については、子クラスでは存在してないようなものだから上書きにはならない。
// つまり特命初期化子には常に上書きが発生し、利便初期化子には決して上書きが発生しない。
// 利便初期化子は上書きの対象に値しない、特命初期化子の付属的な構文糖衣にすぎない。

// automatic initializer inheritance
/// ベースクラス
class Food {
    var name: String
```

```
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
let namedMeat = Food(name: "Bacon")
let mysteryMeat = Food()
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
class ShoppingListItem1: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ☒" : " ☒"
        return output
    }
}
var breakfastList = [
    ShoppingListItem1(),
    ShoppingListItem1(name: "Bacon"),
    ShoppingListItem1(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
```

```
// failable initializer
// 失敗可能な初期化子は init?() と書かれる。
let wholeNumber = 12345.0, pi = 3.14159  // これらを使うとなぜか実行時エラーになった。
if let valueMaintained = Int(exactly: 12345.0) {
    print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")
}
let valueChanged = Int(exactly: 3.14159)
if valueChanged == nil {
    print("\(pi) conversion to Int does not maintain value")
}

struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
let someCreature = Animal(species: "Giraffe")
if let giraffe = someCreature {
    print("An animal was initialized with a species of \(giraffe.species)")
}
let anonymousCreature = Animal(species: "")
if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}

enum TemperatureUnit {
    case kelvin, celsius, fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .kelvin
        case "C":
            self = .celsius
        case "F":
            self = .fahrenheit
        default:
            return nil
```

```
        }
    }
}
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}

// rawValue を持つ列挙型は暗黙に init?(rawValue:) を持つ。
enum TemperatureUnit1: Character {
    case kelvin = "K", celsius = "C", fahrenheit = "F"
}
let fahrenheitUnit1 = TemperatureUnit1(rawValue: "F")
if fahrenheitUnit1 != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
let unknownUnit1 = TemperatureUnit1(rawValue: "X")
if unknownUnit1 == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}

// propagation of initialization failure
class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
```

```
}
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    print("Unable to initialize zero shirts")
}
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
} else {
    print("Unable to initialize one unnamed product")
}

// overriding a failable initializer
// 失敗可能な初期化子を継承して失敗不可能な初期化子にできる。
// 失敗不可能な初期化子を継承して失敗可能な初期化子にはできない。
class Document {
    var name: String?
    init() {}
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {  // init(name: String) で init?(name: String) を上書き。

        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
```

```
}
class UntitledDocument: Document {
    override init() {
        super.init(name: "[Untitled]")!  // force-unwrap
    }
}

// setting a default property value with a closure or function
struct Chessboard {
    let boardColors: [Bool] = {
        var temporaryBoard = [Bool]()
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    /// row 行 column 列のセルが黒ければ真を返す。
    func squareIsBlackAt(row: Int, column: Int) -> Bool {
        return boardColors[(row * 8) + column]
    }
}
let board = Chessboard()
print(board.squareIsBlackAt(row: 0, column: 1))  // true
print(board.squareIsBlackAt(row: 7, column: 7))  // false
```