COMPSCI BOOKS

Complete Programming in QueenScript

L. Mukundwa

QueenScript

1. Introduction

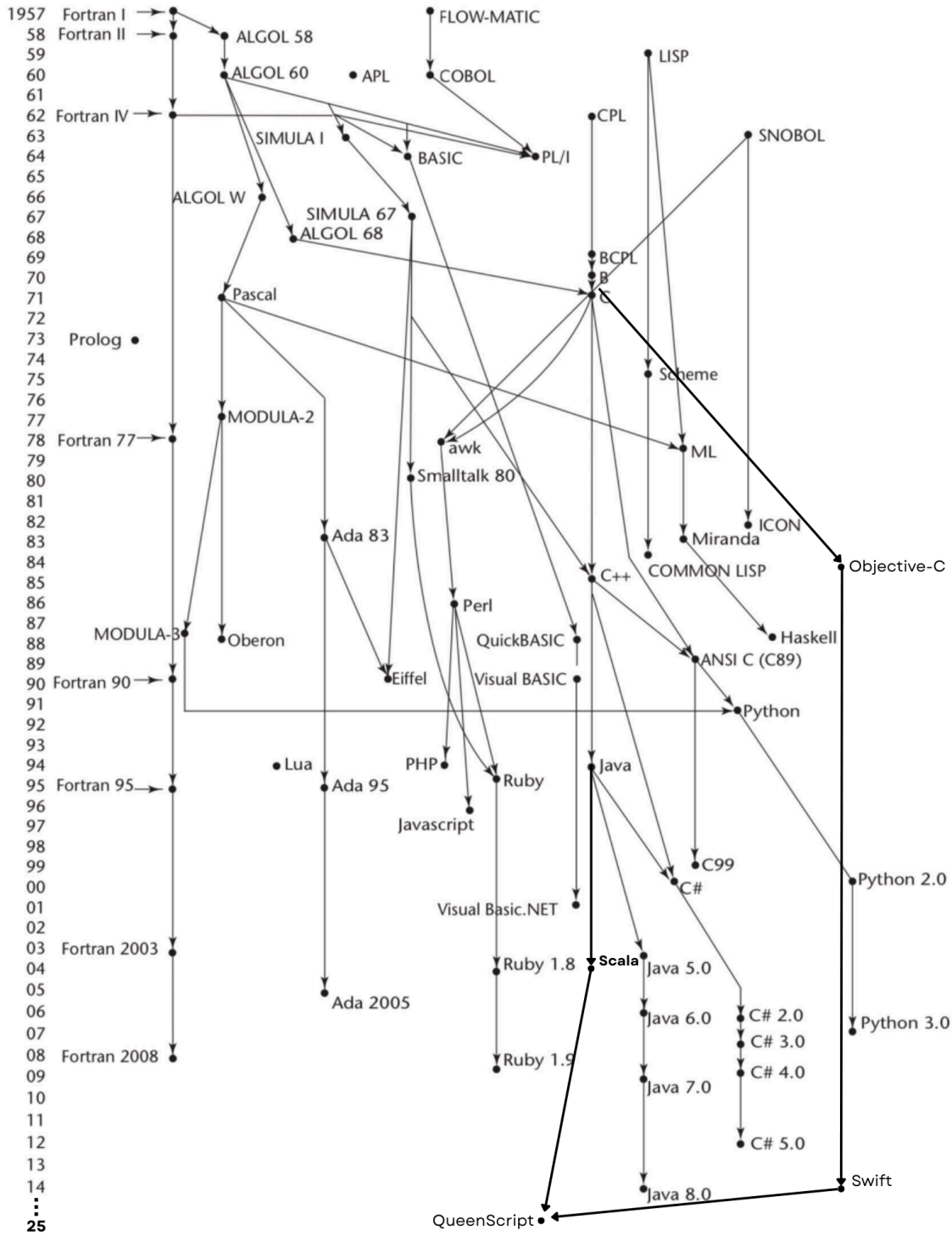
QueenScript is a playful, expressive and typesafe modern programming language that descends from Scala and Swift. It combines the functional aspects of Scala with Swift's safety features and modern syntax, all wrapped in a girlypop, Gen Z meme culture aesthetic. The design philosophy behind this language is "*strongly typed, fabulously dressed*". It brings together the expressive power of Scala and the clean syntax of Swift to create a language that is as powerful as it is stylish, delivering on brains and beauty in one! It is also meant to maintain the aspects that we love about languages such as readability, writability and maintainability while using popular and well structured programming concepts from OOP to fundamental requirements of sequence, alternation and repetition. Although it is based on these two languages, it differs in the following ways:

- ★ **It has a rebranded core syntax**: Many of the core constructs found in these languages like class, interface, package, private and more are reimagined with modern, on-theme names.
- ★ **It has a themed type system:** It is strongly-typed, however it uses expressive, slang-based type aliasing. For example, the 🎀 emoji represents a unit return, replacing () or Void.
- ★ **It has stylized access control:** Visibility modifiers are renamed for readability and witty humor.
- ★ **Sassy Pattern Matching:** It enhances pattern matching by allowing expressive matches that read like natural speech, delivering on readability. Overall, the readability makes it easier and more fun for beginners that want to learn a programming language with personality.
- ★ **It has Gen-Z Developer Culture Integration:** The entire language is rooted in Gen-Z culture with meme-references and general slang that makes it uniquely suited for educational tools, creator-focused projects, and more.

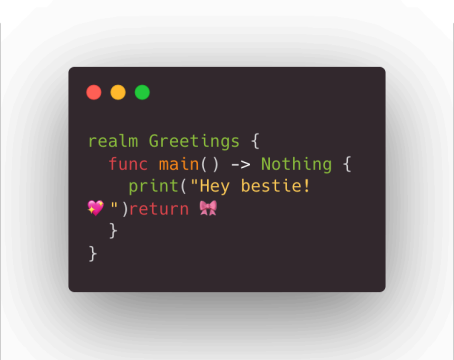
Table of Contents

1.1.Genealogy

The diagram shows where QueenScript fits into the programming language genealogy:



1.2. Hello World



1.3. Program Structure

The key organizational concepts in QueenScript are as follows:

1. Every program must define a **main()** in a **realm**.
2. Any **blueprint** can implement a **vibe**.
3. A **main()** function must return nothing.
4. Any blueprint can have either **core**, **lowkey**, **gatekeep** or **public** variables (which is the default).
5. All source code files should contain a **.queen** file extension.

```
realm App {

  // Interface (vibe) defining user behavior
  vibe Identifiable {
    func displayID() -> Nothing
  }

  // Parent class
  blueprint Account : Identifiable {
    core String username
    gatekeep String passwordHash
    lowkey Int loginAttempts = 0
    public String role = "guest"

    init(username: String, password: String) {
      self.username = username
      self.passwordHash = hash(password)
    }

    func login(password: String) -> Bool {
      if hash(password) == passwordHash {
        loginAttempts = 0
        return true
      } else {
        loginAttempts += 1
        return false
      }
    }

    func displayID() -> Nothing {
      print("User: \(username) | Role: \(role)")
      return 🎀
    }

    // Internal method only for this file's realm
    mutuals func hash(input: String) -> String {
      return "hashed:\(input)" // Placeholder hashing
    }
  }

  // Subclass
  blueprint Admin extends Account {
    core List<String> privileges = ["read", "write",
"delete"]
    func grantPrivilege(p: String) -> Nothing {
      if !privileges.contains(p) {
        privileges.add(p)
      }
      return 🎀
    }

    override func displayID() -> Nothing {
      print("Admin: \(username) 🛡️ ")
      return 🎀
    }
  }

  func main() -> Nothing {
    let user = Account(username: "blair", password: "gossip")
    let admin = Admin(username: "serena", password: "vanD")
    admin.role = "admin"

    user.displayID()
    admin.displayID()

    print("Login success? \(user.login("wrongpass"))")
    print("Login success? \(user.login("gossip"))")

    admin.grantPrivilege("ban")
    print("Admin Privileges: \(admin.privileges)")

    return 🎀
  }
}
```

The **realm** App encapsulates everything within an App module. Within that, there is the **blueprint** for an account and an admin. Account also implements the **vibe** Identifiable interface, this language's trait/protocol. Admin then inherits Account and overrides displayID(). The app has

multiple members such as core variables like username, **lowkey**, **public**, **mutual and gatekeep** variables. **Main()** is the entry point, where every program must define a **main()**in a **realm**. It's fine to be inside whichever **realm** as long as it can still be reached by the compiler. Additionally, it has returned nothing, which is a keyword in this language. **Vibes** are interfaces and define shared behavior.

1.4. Types and Variables

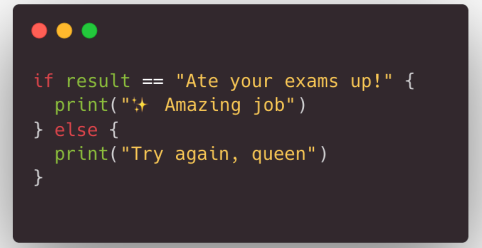
There are two kinds of types in QueenScript: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. See Section 3 for details 🌟.

1.5. Visibility

QueenScript has four choices for access modifiers which control the visibility and accessibility of classes, methods and members within a program. Since Queenscript follows OOP, they help encapsulate the code and manage how different parts of the program interact.

```
// Public access (default)
main vibe Profile {
 // ...
}

// Private access (only within the vibe)
gatekeep Integer password = 1234

// Protected access (within vibe and sub-vibes)
lowkey String nickname = "silverback"

// Package-private (only within the same realm)
mutuals String sharedInfo = "only for the group"
```

1.6. Statements differing from Scala and Swift

STATEMENT	EXAMPLE
For loop →	
While loop →	
If-else →	

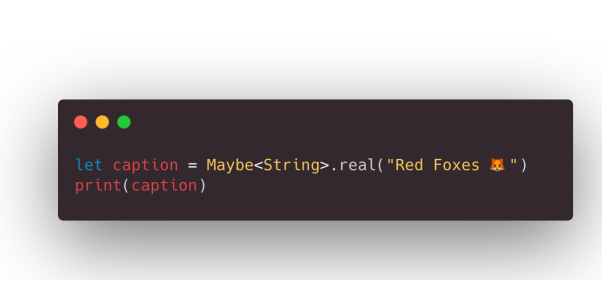
```
for i in 1..<5 {
  print("Looping: \(i)
💅")
}
```

```
var count = 0
while count < 3 {
  print("Counting: \(count) 🛸")
  count += 1
}
```

```
if result == "Ate your exams up!" {
  print("✨ Amazing job")
} else {
  print("Try again, queen")
}
```

Match case →

Function call + Optional →



2.Lexical Structure

2.1. Programs

QueenScript source code is written in files with an extension of "**[filename].queen**". A source file is an ordered sequence of Unicode characters. In each file, there can be more than one **realm**, which are comparable to Swift modules or Scala packages. Since this is a language for programmers, a structured organization could look like

```
Project/
 |
├── main.queen          <- Entry point with main() function
├── Auth.queen          <- Contains blueprint account, admin, and more
├── Util.queen          <- Helpers and reusable vibes
├── DSA.queen           <- Any helpful data structure and algorithms
├── Metrics.queen       <- Libraries (such as GirlMath (internet joke)
```

From this structure, compilation would occur in three phases:

1. **Front-End: .queen** files would be validated using a lexer and then parsed using our BNF grammar. Then, an Abstract Syntax Tree would be created as long as there were no type errors, undefined functions or references, access violations and also respects visibility modifiers.

2. **Intermediate Representation (IR):** The AST would be compiled into an IR layer respecting modifiers, simplifying type-sage instructions, and performing optimizations such as dead code elimination.
3. **Back-End:** The IR would be converted to bytecode, or native code for the target platform, provide any error messages and output a **.qtc** bytecode file or run the **main() vibe.**

2.2. Grammars

This specification presents the syntax of QueenScript where it differs from Scala and Swift.

2.2.1 Lexical grammar (tokens) where different from Scala and Swift

```
IDENTIFIER          ::= [a-zA-Z_] [a-zA-Z0-9_]*

KEYWORD             ::=
                      "blueprint" | "vibe" | "func" | "realm" | "core" | "gatekeep" | "lowkey" | "mutuals" |
"viral" |   "public"
                     | "init" | "main" | "match" | "case" | "else" | "if" | "for" | "in" | "while" | "return"
                     | "try" | "catch" | "finally" | "override" | "Nothing" | "Ghost" | "Maybe"

OPERATOR            ::= "==" | "!=" | "<=" | ">=" | "&&" | "||" | "+" | "-" | "*" | "/" | "=" | "." | ":" | "->" | "::"

DELIMITER           ::= "{" | "}" | "(" | ")" | "[" | "]" | "," | ";" | "<" | ">"

INT_LITERAL         ::= [0-9]+
FLOAT_LITERAL        ::= [0-9]+ "." [0-9]+
STRING_LITERAL      ::= "\"" ( ~["\\] | "\\" . )* "\""
BOOLEAN_LITERAL     ::= "true" | "false"
EMOJI_LITERAL       ::= "🎀"
OPTIONAL_LITERAL    ::= ".ghost" | ".real"

LINE_COMMENT        ::= "//" ~[\n\r]*       // single line comment
BLOCK_COMMENT       ::= "/*" .*? "*/"       // multiline comment

WHITESPACE          ::= [ \t\r\n]+          // (ignored outside of string literals)
```

2.2.2. Syntactic (“parse”) grammar where different from Scala and Swift

```
<program>          ::= { <realm_decl> }

<realm_decl>       ::= "realm" <identifier> "{" { <decl> } "}"

<decl>             ::= <blueprint_decl>
                     | <vibe_decl>
                     | <func_decl>
                     | <var_decl>

<blueprint_decl>   ::= [ "main" ] "blueprint" <identifier> [ ":" <type> ] "{" { <member_decl> } "}"

<vibe_decl>        ::= "vibe" <identifier> "{" { <func_decl> } "}"

<func_decl>        ::= "func" <identifier> "(" [ <params> ] ")" "->" <type> "{" <block> "}"

<params>           ::= <param> { "," <param> }
<param>            ::= <identifier> ":" <type>

<var_decl>         ::= <visibility> <type> <identifier> [ "=" <expr> ]

<visibility>       ::= "core"
                     | "viral"
                     | "lowkey"
                     | "gatekeep"
                     | "mutuals"
                     | "public"  // default

<type>             ::= "Int" | "Float" | "Bool" | "String"
                     | "Nothing" | "Ghost"
                     | "Maybe" "<" <type> ">"
                     | <identifier>

<block>            ::= { <stmt> }

<stmt>             ::= <var_decl> ";"
                     | <assign_stmt> ";"
                     | <if_stmt>
                     | <while_stmt>
                     | <for_stmt>
                     | <match_stmt>
                     | <return_stmt>
                     | <try_stmt>
                     | <expr_stmt> ";"
```

```
<assign_stmt>       ::= <identifier> "=" <expr>

<if_stmt>           ::= "if" <expr> <block> [ "else" <block> ]
<while_stmt>        ::= "while" <expr> <block>
<for_stmt>          ::= "for" <identifier> "in" <expr> <block>

<match_stmt>        ::= "match" <expr> "{" { "case" <pattern> "=>" <block> } [ "case" "_" "=>"
<block> ] "}"

<try_stmt>          ::= "try" <block> "catch" <identifier> "." <identifier> <block> [ "finally"
<block> ]

<return_stmt>       ::= "return" <expr> ";"
<expr_stmt>         ::= <expr>

<expr>              ::= <literal>
                      | <identifier>
                      | <expr> "." <identifier>
                      | <expr> "." <identifier> "(" [ <args> ] ")"
                      | <expr> "(" [ <args> ] ")"
                      | <expr> <binary_op> <expr>
                      | "(" <expr> ")"

<args>              ::= <expr> { "," <expr> }

<binary_op>         ::= "+" | "-" | "*" | "/" | "==" | "!=" | "<" | ">" | "<=" | ">=" | "&&" | "||"

<literal>           ::= <int_literal>
                      | <float_literal>
                      | <string_literal>
                      | "true" | "false"
                      | ".ghost" | ".real" "(" <expr> ")"
                      | "🎀"

<pattern>           ::= <literal> | <identifier>

<identifier>        ::= letter { letter | digit | "_" }
<int_literal>       ::= digit { digit }
<float_literal>     ::= digit { digit } "." digit { digit }
<string_literal>    ::= "\"" { character } "\""
```

2.3. Lexical Analysis

2.3.1. Comments

Similarly to most programming languages, two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters // and extend to the end of the source line. ***Delimited comments*** start with the characters /* and end with the characters */. Delimited comments may span multiple lines. Comments do not nest.

2.4. Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, although they act as separators for tokens where needed.

Tokens:

- ★ identifier
- ★ keyword
- ★ integer-literal
- ★ real-literal
- ★ character-literal
- ★ string-literal
- ★ operator-or-punctuator

2.4.1. Keywords different from Scala and Swift

New keywords	Removed / Replaced keywords
★ link ★ core ★ lowkey ★ viral ★ Maybe<T> ★ real ★ ghost ★ realm ★ blueprint ★ vibe ★ ghost ★ nothing	★ ref ★ const ★ private ★ Static ★ Optional<T> ★ some ★ none ★ package ★ class ★ Interface ★ null ★ void

3. Type System

Just like its predecessors, QueenScript is a statically, strongly typed language and it uses type inference in order to determine what the initial type of variables and cores (constants) are. This means that it performs early binding compile-type checking. Essentially, it compiles to machine code and uses an LLVM-based compiler. Therefore, the compiler will analyze each of the types being used in the code and make sure that the code is correctly using those types, otherwise, one may get this CompilationError:

"Expected a Float, got String instead. Fix it. 🚫💅"

(The compiler is a little sassy...)

There are also runtime errors that can be caught such as the one every developer encounters once in a while, an index out of bounds or off by one.

"Index out of bounds? So is your audacity. 📏"

3.1. Type Rules

The type rules for QueenScript are as follows:

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is a primitive type}}{S \vdash e1 = e2: T}$$

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is a primitive type}}{S \vdash e1 == e2: bool}$$

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is a primitive type}}{S \vdash e1 != e2: bool}$$

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is a primitive type}}{S \vdash e1 < e2: bool}$$

$$\frac{S \vdash e1: T \quad S \vdash e2: T \quad T \text{ is a primitive type}}{S \vdash e1 > e2: bool}$$

$$\frac{S \vdash e1: String \quad S \vdash e2: String}{S \vdash e1 + e2: String}$$

$$\frac{S \vdash e1: Int \quad S \vdash e2: Int}{S \vdash e1 + e2: Int}$$

$$\frac{S \vdash e1: Int \quad S \vdash e2: Int}{S \vdash e1 * e2: Int}$$

$$\frac{S \vdash e1: Int \quad S \vdash e2: Int}{S \vdash e1 - e2: Int}$$

$$\frac{S \vdash e1: String \quad S \vdash e2: Int}{}$$

$$\frac{S \vdash e1: T \quad S \vdash e2: T}{}$$

----------	----------	T is a value type
S ⊢ e1 / e2: Float	S ⊢ e1 + e2: String	----------
		S ⊢ e1 ?: e2: T

QueenScript has two categories for its types, value types and reference types. The important distinction between these two is that value types hold actual data whereas reference types contain a reference to data's location within memory. This language borrows similar types between Scala and Swift but with its own unique flair.

3.2. Value Types

The value types and especially primitive types (Int, Float, Bool), String, Char are all value types that this language are all inherited from their descendents. The primary different ones are:

- ➔ "**Ghost**" (as in ghosting) is similar to "Null" which is of the literal type Null.
- ➔ "**Maybe <T>**" is similar to "Optional <T>" in Swift which represents either a wrapped value or the absence of a value.

3.3. Reference Types

These are some different reference types in this language:

- ➔ '**blueprint**' is similar to a 'class' in both Scala and Swift. It is quite literally a blueprint that you can use to build objects, and conforms to inheritance found in OOP.
- ➔ '**vibe**' is similar to an 'interface' which enforces certain properties on an object (blueprint).

4. Example Programs

4.1 & 4.2 Caesar Cipher:

```
realm CipherKit {

  blueprint CaesarCipher {
    core Int shift

    init(shift: Int) {
      self.shift = shift % 26
    }

    // Encrypt a message
    func encrypt(message: String) -> String {
      var result = ""
      for char in message {
        let code = char.unicode
        if code >= 65 && code <= 90 { // Uppercase A-Z
          result += String.fromUnicode(((code - 65 + shift) % 26) + 65)
        } else if code >= 97 && code <= 122 { // Lowercase a-z
          result += String.fromUnicode(((code - 97 + shift) % 26) + 97)
        } else {
          result += String(char) // Non-letters remain the same
        }
      }
      return result
    }

    // Decrypt a message
    func decrypt(ciphertext: String) -> String {
      var result = ""
      for char in ciphertext {
        let code = char.unicode
        if code >= 65 && code <= 90 {
          result += String.fromUnicode(((code - 65 - shift + 26) % 26) +
65)     } else if code >= 97 && code <= 122 {
          result += String.fromUnicode(((code - 97 - shift + 26) % 26) +
97)     } else {
          result += String(char)
        }
      }
      return result
    }
  }

  func main() -> Nothing {
    let cipher = CaesarCipher(shift: 3)
    let encrypted = cipher.encrypt("Hello, World!")
    print("Encrypted: \(encrypted)")

    let decrypted = cipher.decrypt(encrypted)
    print("Decrypted: \(decrypted)")

    return 🎀
  }
}
```

4.3. Factorial:

```
realm MathProgram {
  func factorial(n: Int) -> Int {
    if n <= 1 {
      return 1
    } else {
      return n * factorial(n - 1)
    }
  }


  func main() -> Nothing {
    let result = factorial(5)
    print("Factorial: \(result)")
    return 🎀
  }
}
```

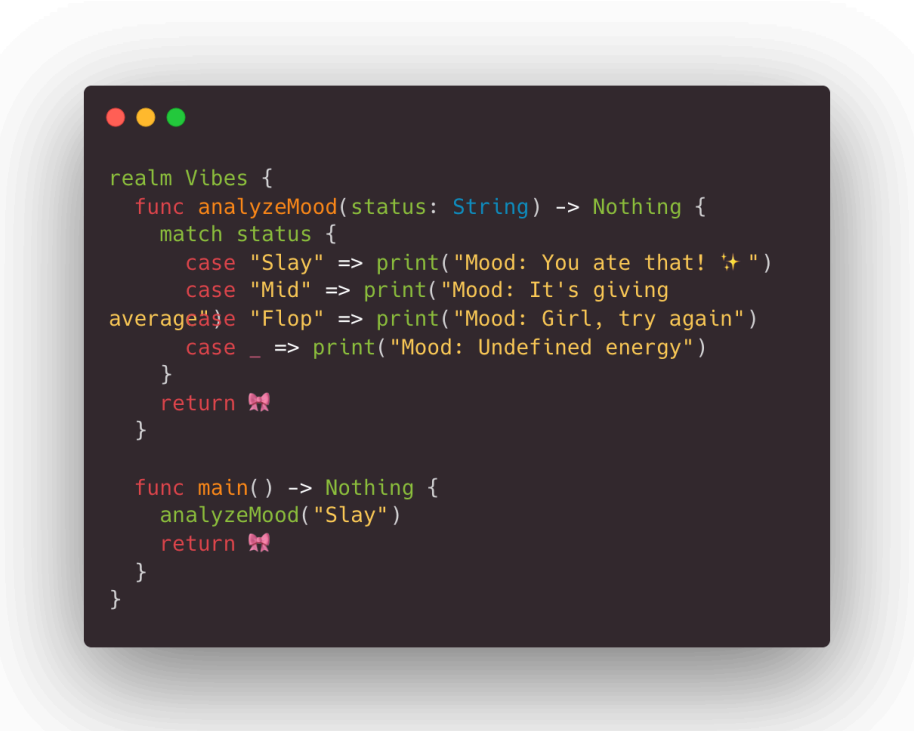
4.4. Merge Sort

```
realm SortProgram {
  func mergeSort(list: List<Int>) -> List<Int> {
    if list.size <= 1 {
      return list
    }


    let middle = list.size / 2
    let left = mergeSort(list.slice(0, middle))
    let right = mergeSort(list.slice(mid, list.size))


    return merge(left, right)
  }


  func merge(left: List<Int>, right: List<Int>) -> List<Int>
{   var result: List<Int> = []
    var i = 0
    var j = 0


    while i < left.size && j < right.size {
      if left[i] < right[j] {
        result.add(left[i])
        i += 1
      } else {
        result.add(right[j])
        j += 1
      }
    }


    result.addAll(left.slice(i, left.size))
    result.addAll(right.slice(j, right.size))
    return result
  }


  func main() -> Nothing {
    let unsorted = [5, 3, 8, 1, 2]
    let sorted = mergeSort(unsorted)
    print("Sorted: \(sorted)")
    return 🎀
  }
}
```

4.5 Binary Search Tree:

```
realm Trees {
  blueprint Node {
    core Int value
    Maybe<Node> left = .ghost
    Maybe<Node> right = .ghost

    init(value: Int) {
      self.value = value
    }
  }

  blueprint BST {
    Maybe<Node> root = .ghost

    func insert(value: Int) -> Nothing {
      root = insertNode(root, value)
      return 🎀
    }

    func insertNode(current: Maybe<Node>, value: Int) -> Node
{     if current == .ghost {
        return Node(value: value)
      }
      if value < current.value {
        current.left = insertNode(current.left, value)
      } else {
        current.right = insertNode(current.right, value)
      }
      return current
    }

    func printInOrder(node: Maybe<Node>) -> Nothing {
      if node != .ghost {
        printInOrder(node.left)
        print("\(node.value)")
        printInOrder(node.right)
      }
      return 🎀
    }
  }

  func main() -> Nothing {
    let tree = BST()
    tree.insert(4)
    tree.insert(2)
    tree.insert(6)
    tree.insert(1)
    tree.insert(3)
    tree.printInOrder(tree.root)
    return 🎀
  }
}
```

4.6 Pattern Matching:



```
realm Vibes {
  func analyzeMood(status: String) -> Nothing {
    match status {
      case "Slay" => print("Mood: You ate that! ✨")
      case "Mid" => print("Mood: It's giving
average.")
      case "Flop" => print("Mood: Girl, try again")
      case _ => print("Mood: Undefined energy")
    }
    return 🎀
  }

  func main() -> Nothing {
    analyzeMood("Slay")
    return 🎀
  }
}
```