How to Write a Swift Macro

without going (too much) down the rabbit hole

Vincent Pradeilles 🇫🇷 (@v_pradeilles) – PhotoRoom

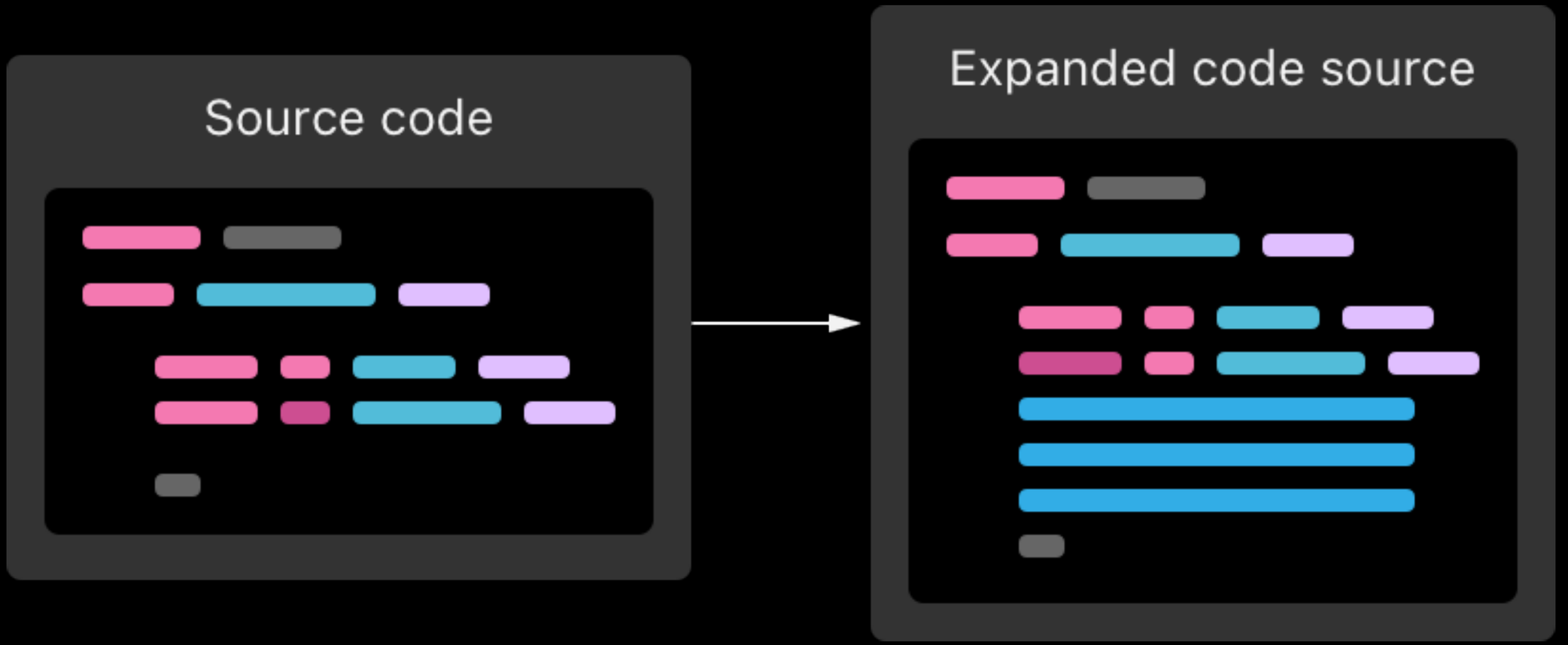
Let’s start with the bad news…

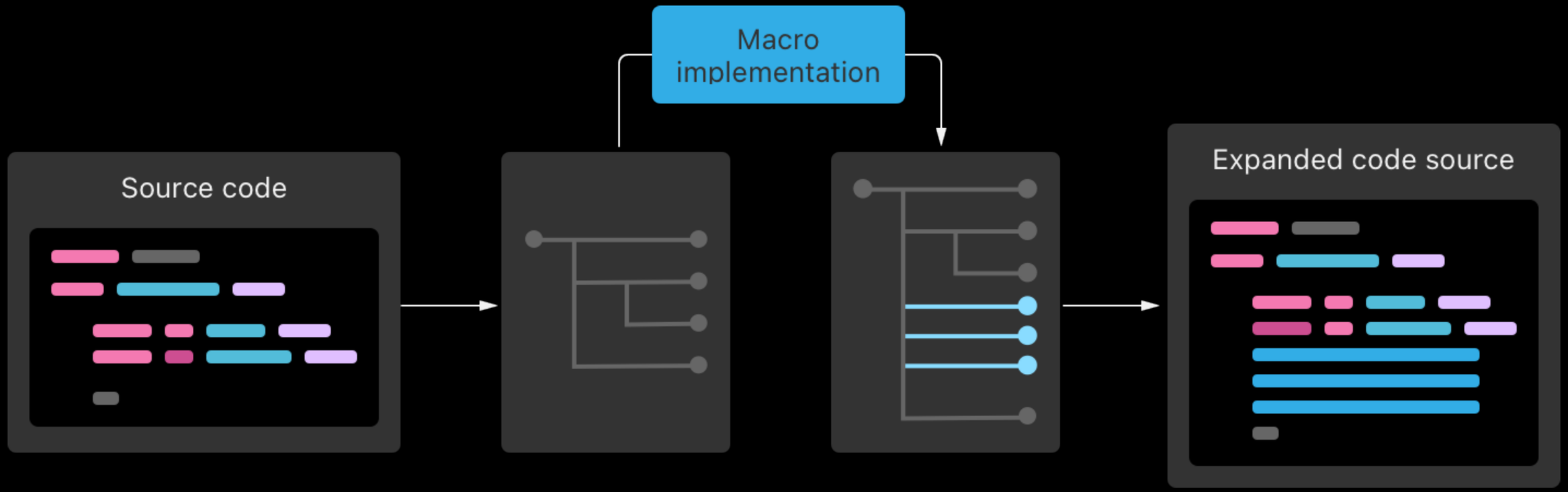
…Swift macros are hard!

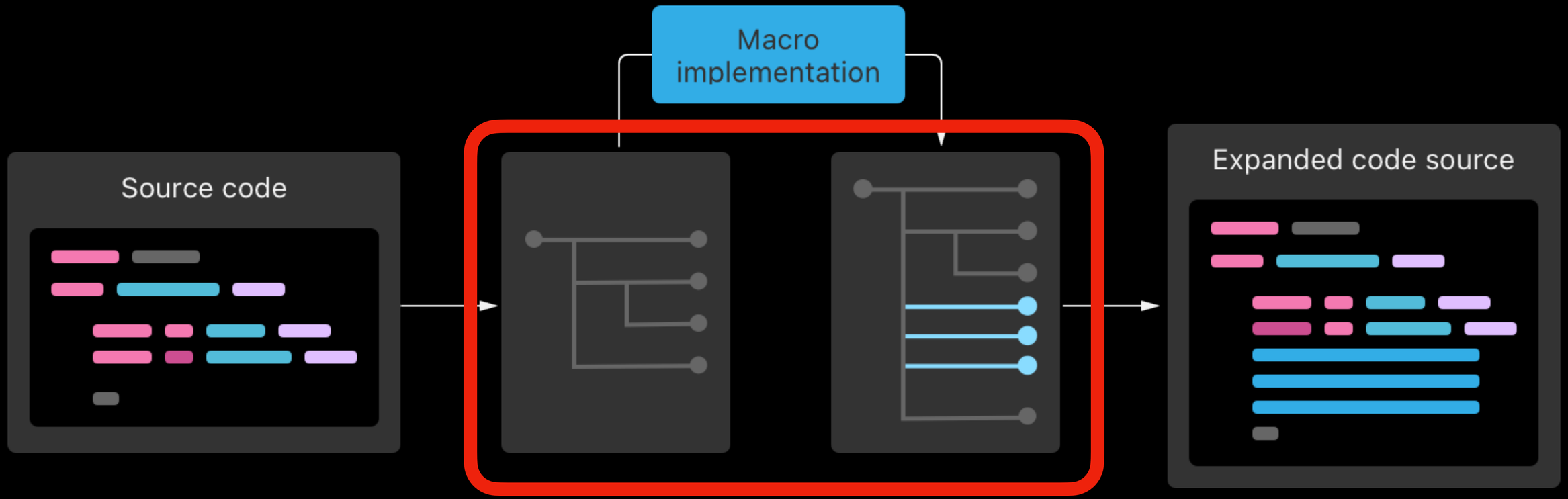
And there’s no way around it 😔

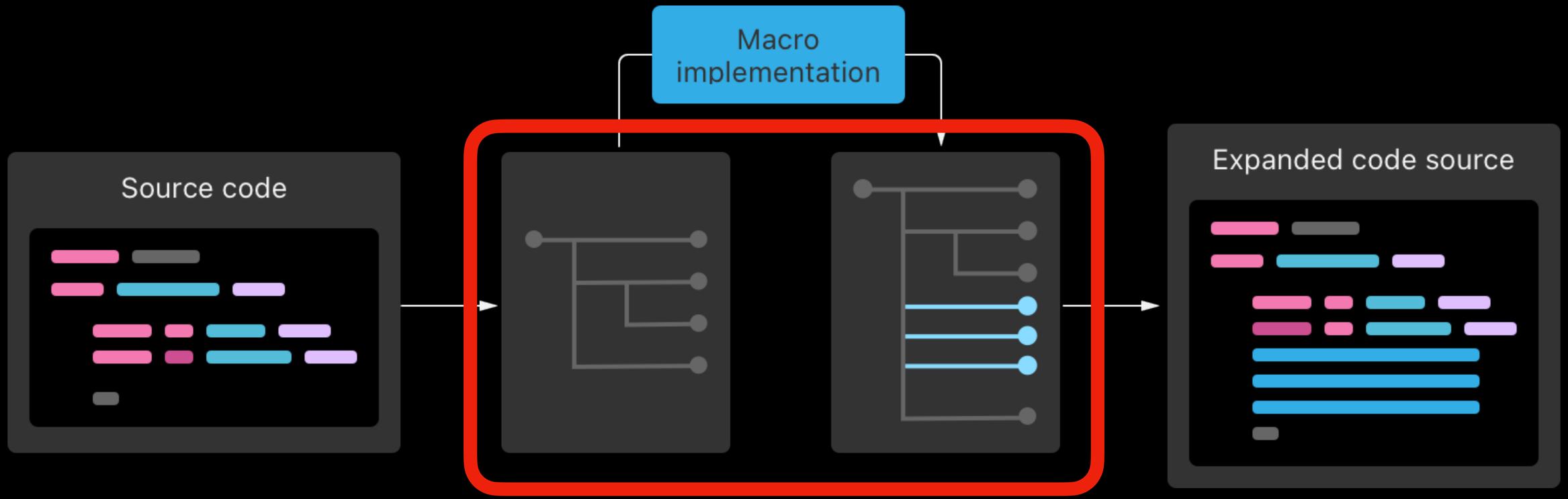
Why are macros that hard? 🤨

Let’s deep dive on how macros work!

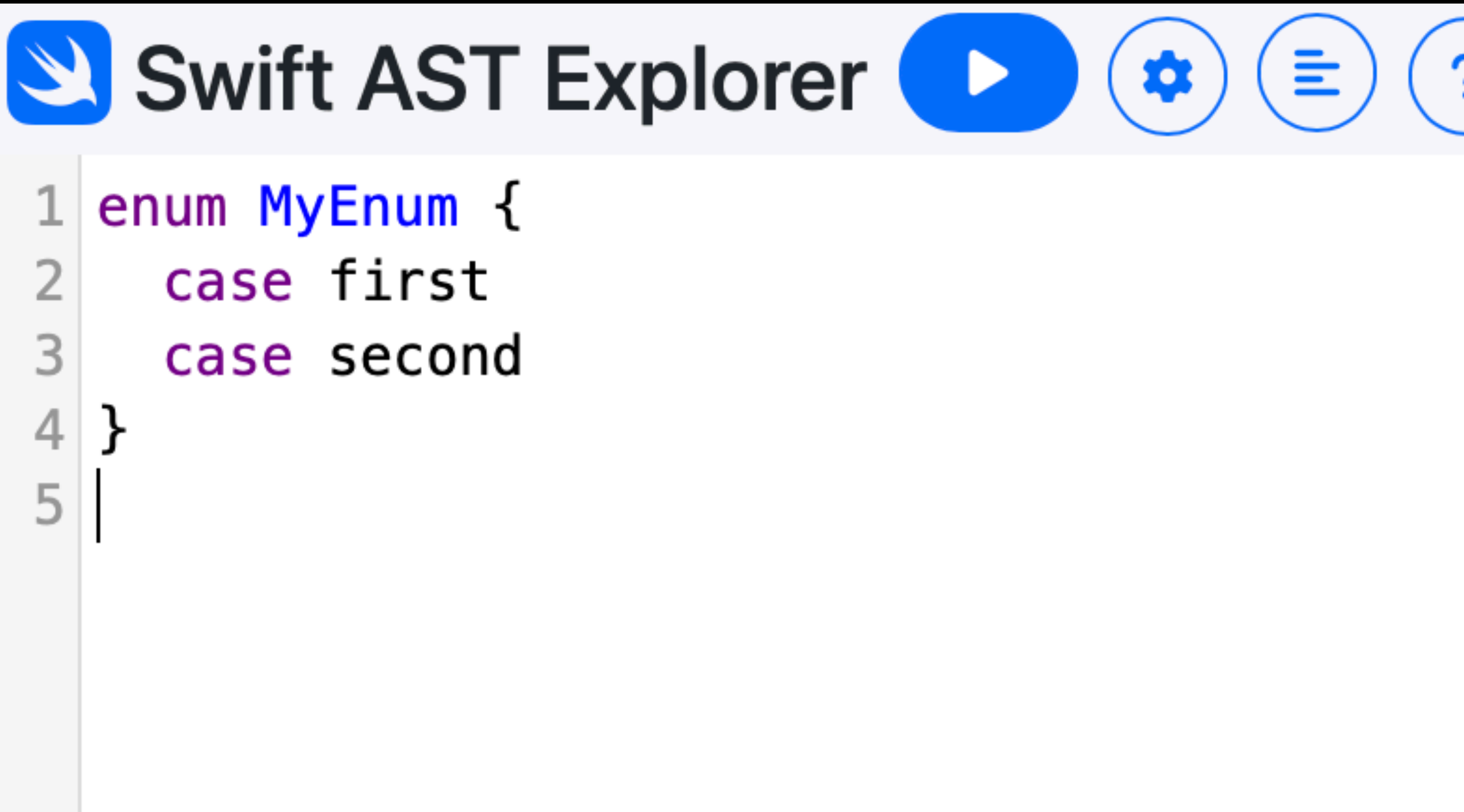

Source code
Expanded code source

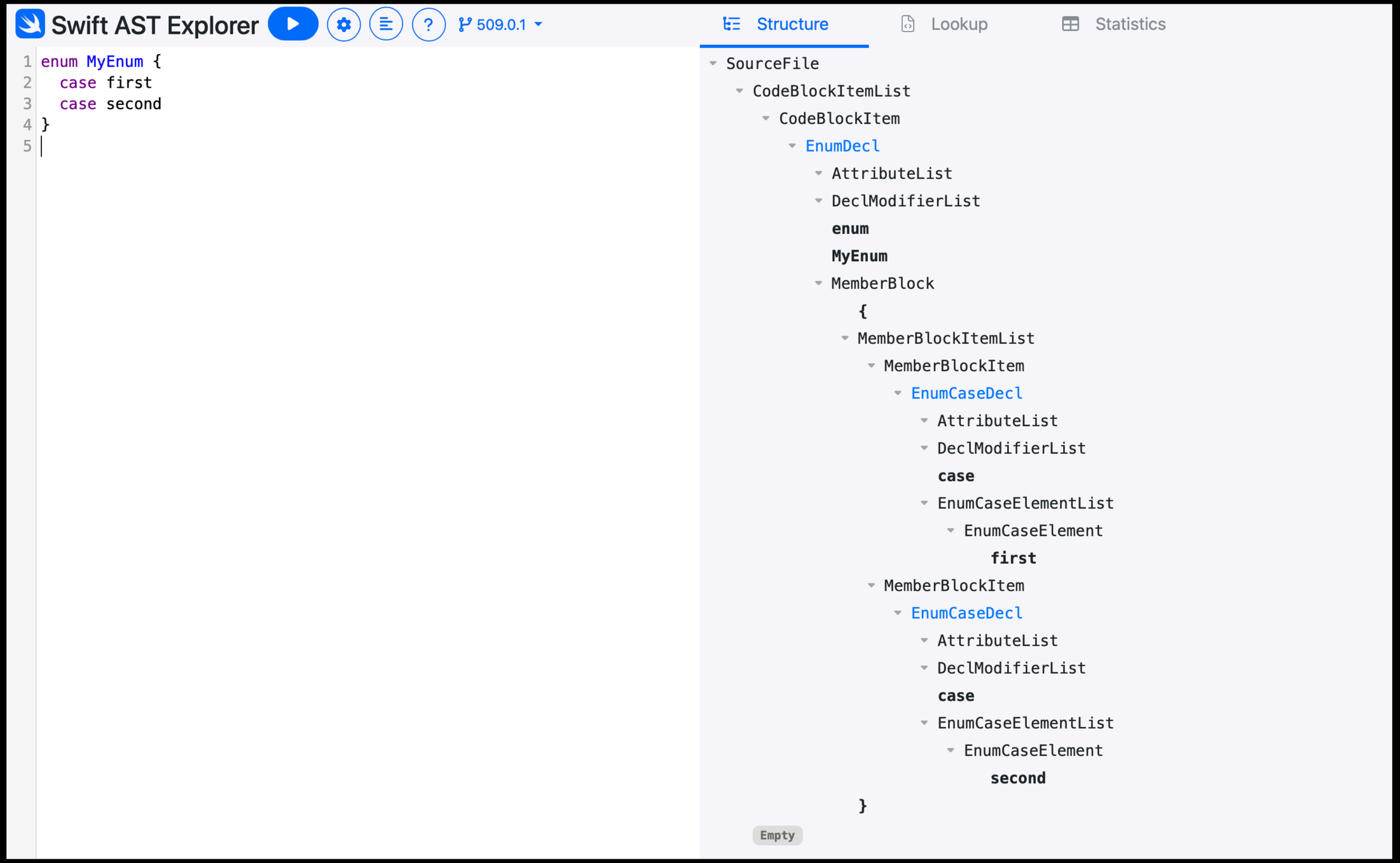

Macro implementation
Source code
Expanded code source


Macro
implementation
Source code
Expanded code source



Abstract Syntax Tree (or AST)


Swift AST Explorer
1 enum MyEnum {
2 case first
3 case second
4 }
5


Swift AST Explorer
509.0.1
Structure
Lookup
Statistics
1 enum MyEnum {
2 case first
3 case second
4 }
5
SourceFile
CodeBlockItemList
CodeBlockItem
EnumDecl
AttributeList
DeclModifierList
enum
MyEnum
MemberBlock
{
MemberBlockItemList
MemberBlockItem
EnumCaseDecl
AttributeList
DeclModifierList
case
EnumCaseElementList
EnumCaseElement
first
MemberBlockItem
EnumCaseDecl
AttributeList
DeclModifierList
case
EnumCaseElementList
EnumCaseElement
second
}
Empty

How macros work

How macros work

- Big picture: a macro takes a portion of the AST as its input…

How macros work

- Big picture: a macro takes a portion of the AST as its input…
- …and expands it into a new AST

How macros work

- Big picture: a macro takes a portion of the AST as its input…
- …and expands it into a new AST
- The AST is a complex structure…

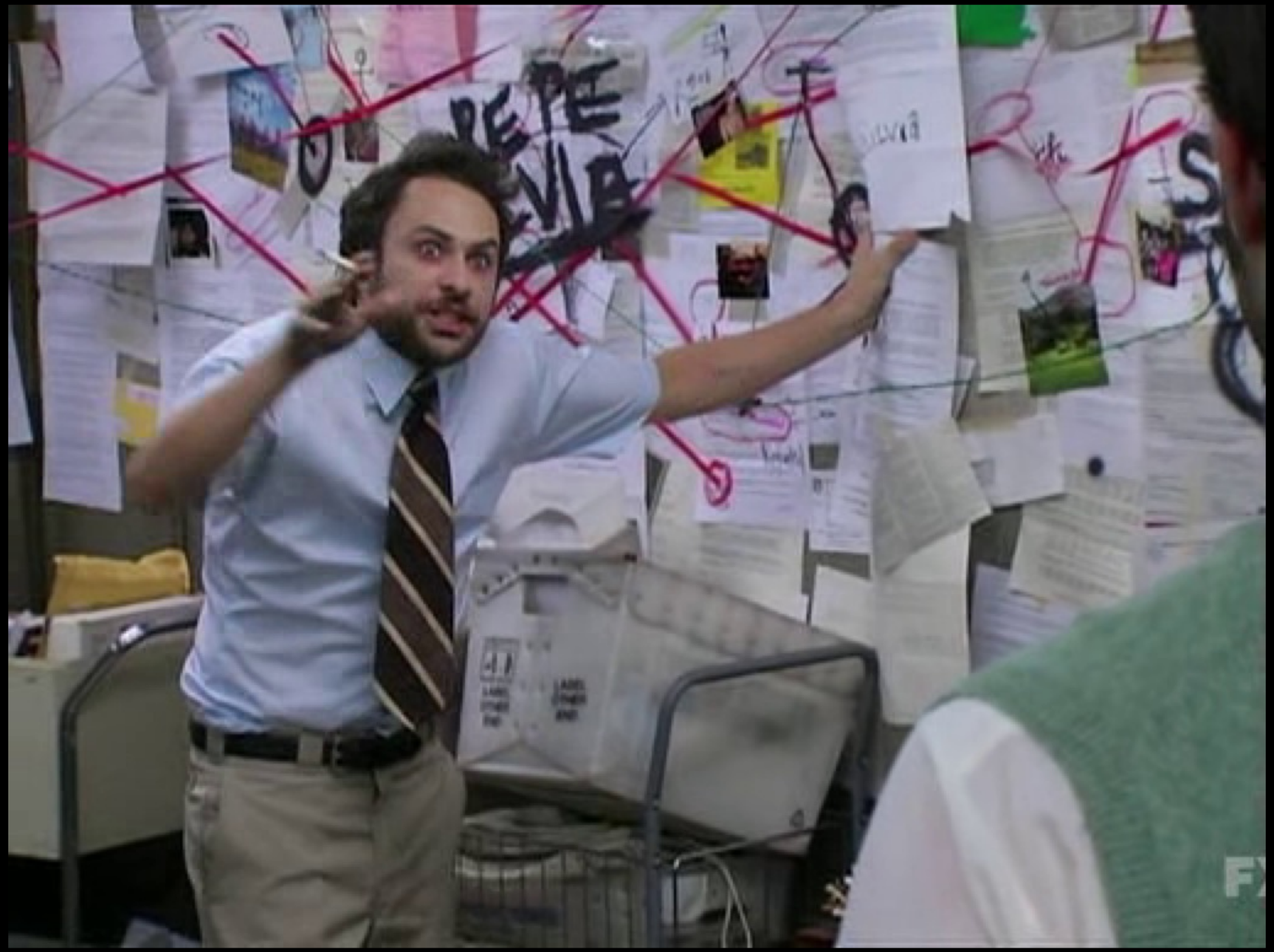
How macros work

- Big picture: a macro takes a portion of the AST as its input…
- …and expands it into a new AST
- The AST is a complex structure…
- …and handling it is fundamentally different than writing app-level code

How macros work

- Big picture: a macro takes a portion of the AST as its input…
- …and expands it into a new AST
- The AST is a complex structure…
- …and handling it is fundamentally different than writing app-level code
- And that’s the main reason why macros are hard!

This means that a talk about building macros is likely to feel like this:



So here’s what we’re going to do

So here’s what we’re going to do

So here’s what we’re going to do

- I made a selection of macros that have been implemented by the community

So here’s what we’re going to do

- I made a selection of macros that have been implemented by the community
- We’ll go over how to use them and how they’ve been implemented

So here’s what we’re going to do

- I made a selection of macros that have been implemented by the community
- We’ll go over how to use them and how they’ve been implemented
- This way you’re going to discover macros that are ready-to-use…

So here’s what we’re going to do

- I made a selection of macros that have been implemented by the community
- We’ll go over how to use them and how they’ve been implemented
- This way you’re going to discover macros that are ready-to-use…
- …and you’ll also have some pointers on how to implement or customize macros, should you need to!

#01 – #URL(_:)

https://github.com/apple/swift-syntax/tree/main/Examples/Sources/MacroExamples

```
import Foundation

let goodUrl = #URL("https://www.apple.com")

let badUrl = #URL("https://www.ap   ple.com")
```

```
import Foundation

let goodUrl = #URL("https://www.apple.com")
URL(string: "https://www.apple.com")!                    #URL

let badUrl = #URL("https://www.ap    ple.com")   M...
```

Malformed url: "https://www.ap ple.com"

Now let’s see the implementation

```
/// Check if provided string literal is a valid URL and
produce a non-optional
/// URL value. Emit error otherwise.
@freestanding(expression)
public macro URL(
    _ stringLiteral: String
) -> URL = #externalMacro(
    module: "MyMacroMacros",
    type: "URLMacro"
)
```

```
/// Check if provided string literal is a valid URL and
produce a non-optional
/// URL value. Emit error otherwise.
@freestanding(expression)
public macro URL(
    _ stringLiteral: String
) -> URL = #externalMacro(
    module: "MyMacroMacros",
    type: "URLMacro"
)
```

```
/// Check if provided string literal is a valid URL and
produce a non-optional
/// URL value. Emit error otherwise.
@freestanding(expression)
public macro URL(
    _ stringLiteral: String
) -> URL = #externalMacro(
    module: "MyMacroMacros",
    type: "URLMacro"
)
```

```
/// Check if provided string literal is a valid URL and
produce a non-optional
/// URL value. Emit error otherwise.
@freestanding(expression)
public macro URL(
    _ stringLiteral: String
) -> URL = #externalMacro(
    module: "MyMacroMacros",
    type: "URLMacro"
)
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }

    // 2. Running the macro's logic
    guard let _ = URL(string: literalSegment.content.text) else {
      throw CustomError.message("malformed url: \(argument)")
    }

    // 3. Returning a new AST
    return "URL(string: \(argument))!"
  }
}
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

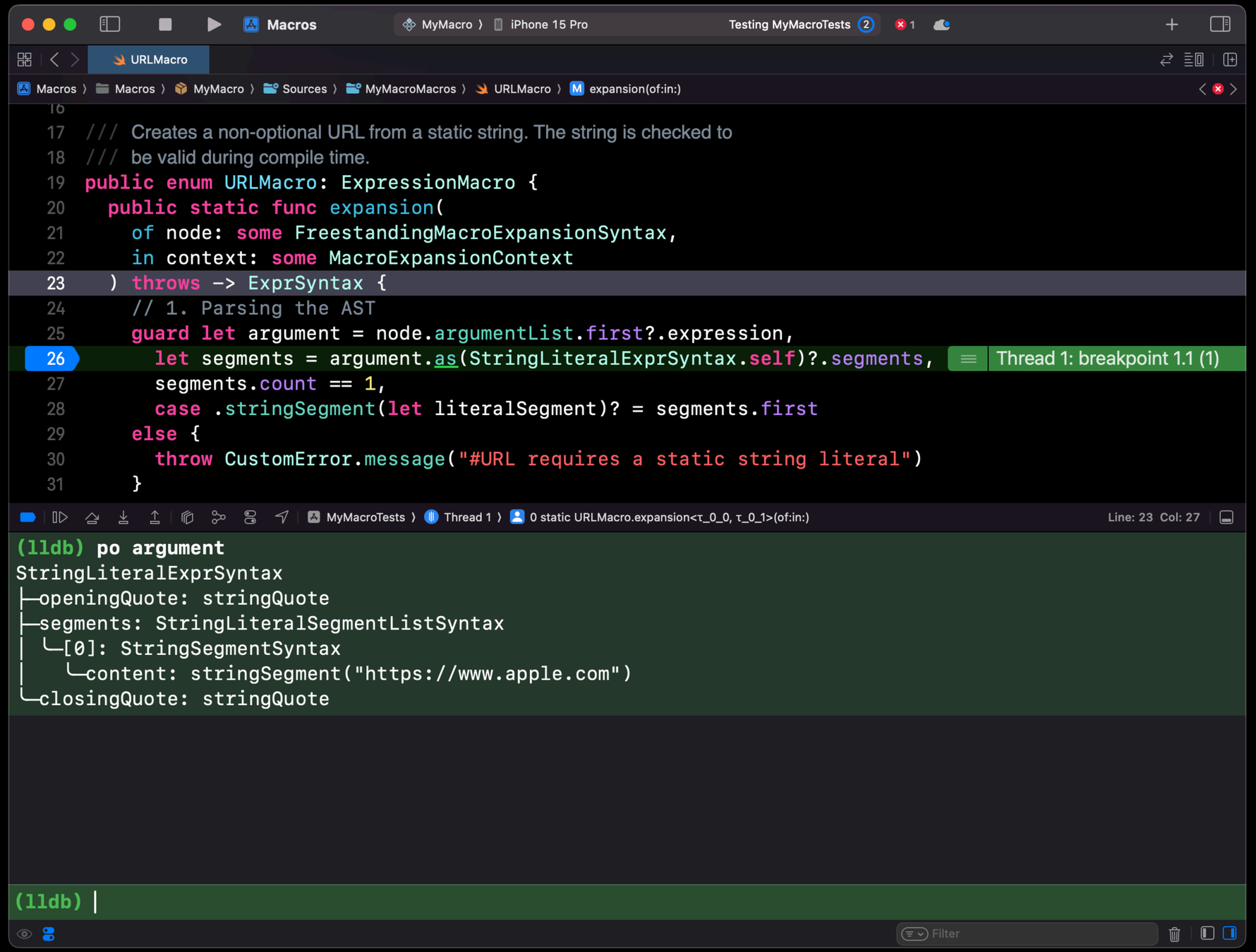
```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```


Macros
MyMacro 〉 iPhone 15 Pro
Testing MyMacroTests
URLMacro
Macros 〉 Macros 〉 MyMacro 〉 Sources 〉 MyMacroMacros 〉 URLMacro 〉 expansion(of:in:)
/// Creates a non-optional URL from a static string. The string is checked to
/// be valid during compile time.
public enum URLMacro: ExpressionMacro {
public static func expansion(
of node: some FreestandingMacroExpansionSyntax,
in context: some MacroExpansionContext
) throws -> ExprSyntax {
// 1. Parsing the AST
guard let argument = node.argumentList.first?.expression,
let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
Thread 1: breakpoint 1.1 (1)
segments.count == 1,
case .stringSegment(let literalSegment)? = segments.first
else {
throw CustomError.message("#URL requires a static string literal")
}
MyMacroTests 〉 Thread 1 〉 0 static URLMacro.expansion<τ_0_0, τ_0_1>(of:in:)
Line: 23 Col: 27
(lldb) po argument
StringLiteralExprSyntax
├─openingQuote: stringQuote
├─segments: StringLiteralSegmentListSyntax
│ └─[0]: StringSegmentSyntax
│ └─content: stringSegment("https://www.apple.com")
└─closingQuote: stringQuote
(lldb)
Filter

```swift
public enum URLMacro: ExpressionMacro {
  public static func expansion(
    of node: some FreestandingMacroExpansionSyntax,
    in context: some MacroExpansionContext
  ) throws -> ExprSyntax {
    // 1. Parsing the AST
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }
```

```
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }

    // 2. Running the macro's logic
    guard let _ = URL(string: literalSegment.content.text) else {
      throw CustomError.message("malformed url: \(argument)")
    }

    // 3. Returning a new AST
    return "URL(string: \(argument))!"
  }
}
```

```
    guard let argument = node.argumentList.first?.expression,
      let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
      segments.count == 1,
      case .stringSegment(let literalSegment)? = segments.first
    else {
      throw CustomError.message("#URL requires a static string literal")
    }

    // 2. Running the macro's logic
    guard let _ = URL(string: literalSegment.content.text) else {
      throw CustomError.message("malformed url: \(argument)")
    }

    // 3. Returning a new AST
    return "URL(string: \(argument))!"
  }
}
```

```
      guard let argument = node.argumentList.first?.expression,
        let segments = argument.as(StringLiteralExprSyntax.self)?.segments,
        segments.count == 1,
        case .stringSegment(let literalSegment)? = segments.first
      else {
        throw CustomError.message("#URL requires a static string literal")
      }

      // 2. Running the macro's logic
      guard let _ = URL(string: literalSegment.content.text) else {
        throw CustomError.message("malformed url: \(argument)")
      }

      // 3. Returning a new AST
      return "URL(string: \(argument))!"
    }
}
```

```
import Foundation

let goodUrl = #URL("https://www.apple.com")
URL(string: "https://www.apple.com")!                    #URL

let badUrl = #URL("https://www.ap    ple.com")   M...
```

Malformed url: "https://www.ap ple.com"

#02 – @ReuseIdentifier

https://github.com/collisionspace/ReuseIdentifierMacro

```
@ReuseIdentifier
class CarouselCollectionViewCell {}

let reuseID = CarouselCollectionViewCell.reuseIdentifier
```

```swift
@ReuseIdentifier
class CarouselCollectionViewCell {}
    static var reuseIdentifier: String {
        "CarouselCollectionViewCell"
    }

let reuseID = CarouselCollectionViewCell.reuseIdentifier
```

```
/// A macro that produces a reuseIdentifier that's usually required
/// for UICollectionViewCells and UITableViewCells
@attached(member, names: named(reuseIdentifier))
public macro ReuseIdentifier() = #externalMacro(
    module: "ReuseIdentifierMacros",
    type: “ReuseIdentifierMacro"
)
```

```
/// A macro that produces a reuseIdentifier that's usually required
/// for UICollectionViewCells and UITableViewCells
@attached(member, names: named(reuseIdentifier))
public macro ReuseIdentifier() = #externalMacro(
    module: "ReuseIdentifierMacros",
    type: “ReuseIdentifierMacro"
)
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
public struct ReuseIdentifierMacro: MemberMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingMembersOf declaration: some DeclGroupSyntax,
        in context: some MacroExpansionContext
    ) throws -> [SwiftSyntax.DeclSyntax] {
        guard let classDecl = declaration.as(ClassDeclSyntax.self) else {
            throw ReuseIdentifierError.onlyApplicableToClass
        }

        let reuseID = try VariableDeclSyntax("static var reuseIdentifier: String") {
            StringLiteralExprSyntax(content: classDecl.identifier.text)
        }

        return [
            DeclSyntax(reuseID)
        ]
    }
}
```

```swift
@ReuseIdentifier
class CarouselCollectionViewCell {}
    static var reuseIdentifier: String {
        "CarouselCollectionViewCell"
    }

let reuseID = CarouselCollectionViewCell.reuseIdentifier
```

#03 – @CaseDetection

https://github.com/apple/swift-syntax/tree/main/Examples/Sources/MacroExamples

```
enum Pet {
   case dog
   case cat(curious: Bool)

}

let pet: Pet = .cat(curious: true)

if case .cat = pet {
    print("Pet is cat")
}
```

```swift
@CaseDetection
enum Pet {
  case dog
  case cat(curious: Bool)

  var isDog: Bool {
    if case .dog = self {
      return true
    }

    return false
  }

  var isCat: Bool {
    if case .cat = self {
      return true
    }

    return false
  }
}
```

@CaseDetection

```
/// Add computed properties named `is<Case>` for each case
element in the enum.
@attached(member, names: arbitrary)
public macro CaseDetection() = #externalMacro(
    module: "MyMacroMacros",
    type: "CaseDetectionMacro"
)
```

```
/// Add computed properties named `is<Case>` for each case
element in the enum.
@attached(member, names: arbitrary)
public macro CaseDetection() = #externalMacro(
    module: "MyMacroMacros",
    type: "CaseDetectionMacro"
)
```

```swift
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
            return true
          }

          return false
        }
        """
      }
  }
}
```

```swift
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
```

```
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
```

```swift
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
```

```swift
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
```

```swift
public enum CaseDetectionMacro: MemberMacro {
  public static func expansion(
    of node: AttributeSyntax,
    providingMembersOf declaration: some DeclGroupSyntax,
    in context: some MacroExpansionContext
  ) throws -> [DeclSyntax] {
    declaration.memberBlock.members
      .compactMap { $0.decl.as(EnumCaseDeclSyntax.self) }
      .map { $0.elements.first!.name }
      .map { ($0, $0.initialUppercased) }
      .map { original, uppercased in
        """
        var is\(raw: uppercased): Bool {
          if case .\(raw: original) = self {
```

```
        .map { $0.elements.first!.name }
        .map { ($0, $0.initialUppercased) }
        .map { original, uppercased in
          """
          var is\(raw: uppercased): Bool {
            if case .\(raw: original) = self {
              return true
            }

            return false
          }
          """
        }
    }
}
```

```
        .map { $0.elements.first!.name }
        .map { ($0, $0.initialUppercased) }
        .map { original, uppercased in
          """
          var is\(raw: uppercased): Bool {
            if case .\(raw: original) = self {
              return true
            }

            return false
          }
          """
        }
    }
}
```

```swift
@CaseDetection
enum Pet {
  case dog
  case cat(curious: Bool)

  var isDog: Bool {
    if case .dog = self {
      return true
    }

    return false
  }

  var isCat: Bool {
    if case .cat = self {
      return true
    }

    return false
  }
}
```

@CaseDetection

#04 – @EnvironmentValue & @EnvironmentStorage

https://github.com/Wouter01/SwiftUI-Macros/

```swift
import SwiftUI

struct SecondaryFontEnvironmentKey: EnvironmentKey {
    static var defaultValue: Font?
}

extension EnvironmentValues {
    var secondaryFont: Font? {
        get {
            self [SecondaryFontEnvironmentKey.self]
        }
        set {
            self [SecondaryFontEnvironmentKey.self] = newValue
        }
    }
}
```

```swift
import SwiftUI

@EnvironmentStorage
extension EnvironmentValues {
    var secondaryFont: Font?
}
```

```
26
27 @EnvironmentStorage
28 extension EnvironmentValues {
       @EnvironmentValue                    @EnvironmentStorage
29     var secondaryFont: Font?
30 }
31
```

```
@EnvironmentStorage
extension EnvironmentValues {
    @EnvironmentValue                                   @EnvironmentStorage
    var secondaryFont: Font?
        internal struct EnvironmentKey_secondaryFont: EnvironmentKey {
            static var defaultValue: Font?
        }
    {                                                   @EnvironmentValue
        get {
            self [EnvironmentKey_secondaryFont.self]
        }
        set {
            self [EnvironmentKey_secondaryFont.self] = newValue
        }
    }
}
```

```
/// Applies the @EnvironmentValue macro to each child in the scope.
/// This should only be applied on an EnvironmentValues extension.
@attached(memberAttribute)
public macro EnvironmentStorage() = #externalMacro(
    module: "SwiftUIMacrosImpl",
    type: “EnvironmentStorage"
)
```

```
/// Applies the @EnvironmentValue macro to each child in the scope.
/// This should only be applied on an EnvironmentValues extension.
@attached(memberAttribute)
public macro EnvironmentStorage() = #externalMacro(
    module: "SwiftUIMacrosImpl",
    type: “EnvironmentStorage"
)
```

```swift
public struct EnvironmentStorage: MemberAttributeMacro {
    public static func expansion(
        of node: AttributeSyntax,
        attachedTo declaration: some DeclGroupSyntax,
        providingAttributesFor member: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AttributeSyntax] {

        // Only attach macro if member is a variable.
        // Otherwise, it will also get attached to the structs generated by @EnvironmentValue
        guard member.is(VariableDeclSyntax.self) else {
            return []
        }

        return [
            AttributeSyntax(
                atSignToken: .atSignToken(),
                attributeName: SimpleTypeIdentifierSyntax(name: .identifier("EnvironmentValue"))
            )
        ]
    }
}
```

```swift
public struct EnvironmentStorage: MemberAttributeMacro {
    public static func expansion(
        of node: AttributeSyntax,
        attachedTo declaration: some DeclGroupSyntax,
        providingAttributesFor member: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AttributeSyntax] {

        // Only attach macro if member is a variable.
        // Otherwise, it will also get attached to the structs generated by @EnvironmentValue
        guard member.is(VariableDeclSyntax.self) else {
            return []
        }

        return [
            AttributeSyntax(
                atSignToken: .atSignToken(),
                attributeName: SimpleTypeIdentifierSyntax(name: .identifier("EnvironmentValue"))
            )
        ]
    }
}
```

```swift
public struct EnvironmentStorage: MemberAttributeMacro {
    public static func expansion(
        of node: AttributeSyntax,
        attachedTo declaration: some DeclGroupSyntax,
        providingAttributesFor member: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AttributeSyntax] {

        // Only attach macro if member is a variable.
        // Otherwise, it will also get attached to the structs generated by @EnvironmentValue
        guard member.is(VariableDeclSyntax.self) else {
            return []
        }

        return [
            AttributeSyntax(
                atSignToken: .atSignToken(),
                attributeName: SimpleTypeIdentifierSyntax(name: .identifier("EnvironmentValue"))
            )
        ]
    }
}
```

```swift
public struct EnvironmentStorage: MemberAttributeMacro {
    public static func expansion(
        of node: AttributeSyntax,
        attachedTo declaration: some DeclGroupSyntax,
        providingAttributesFor member: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AttributeSyntax] {

        // Only attach macro if member is a variable.
        // Otherwise, it will also get attached to the structs generated by @EnvironmentValue
        guard member.is(VariableDeclSyntax.self) else {
            return []
        }

        return [
            AttributeSyntax(
                atSignToken: .atSignToken(),
                attributeName: SimpleTypeIdentifierSyntax(name: .identifier("EnvironmentValue"))
            )
        ]
    }
}
```

```swift
public struct EnvironmentStorage: MemberAttributeMacro {
    public static func expansion(
        of node: AttributeSyntax,
        attachedTo declaration: some DeclGroupSyntax,
        providingAttributesFor member: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AttributeSyntax] {

        // Only attach macro if member is a variable.
        // Otherwise, it will also get attached to the structs generated by @EnvironmentValue
        guard member.is(VariableDeclSyntax.self) else {
            return []
        }

        return [
            AttributeSyntax(
                atSignToken: .atSignToken(),
                attributeName: SimpleTypeIdentifierSyntax(name: .identifier("EnvironmentValue"))
            )
        ]
    }
}
```

```
26
27 @EnvironmentStorage
28 extension EnvironmentValues {
       @EnvironmentValue                    @EnvironmentStorage
29     var secondaryFont: Font?
30 }
31
```

```
/// Creates an unique EnvironmentKey for the variable and adds getters and setters.
/// The initial value of the variable becomes the default value of the EnvironmentKey.
@attached(peer, names: prefixed(EnvironmentKey_))
@attached(accessor, names: named(get), named(set))
public macro EnvironmentValue() = #externalMacro(
    module: "SwiftUIMacrosImpl",
    type: “AttachedMacroEnvironmentKey"
)
```

```
/// Creates an unique EnvironmentKey for the variable and adds getters and setters.
/// The initial value of the variable becomes the default value of the EnvironmentKey.
@attached(peer, names: prefixed(EnvironmentKey_))
@attached(accessor, names: named(get), named(set))
public macro EnvironmentValue() = #externalMacro(
    module: "SwiftUIMacrosImpl",
    type: “AttachedMacroEnvironmentKey"
)
```

```
/// Creates an unique EnvironmentKey for the variable and adds getters and setters.
/// The initial value of the variable becomes the default value of the EnvironmentKey.
@attached(peer, names: prefixed(EnvironmentKey_))
@attached(accessor, names: named(get), named(set))
public macro EnvironmentValue() = #externalMacro(
    module: "SwiftUIMacrosImpl",
    type: “AttachedMacroEnvironmentKey"
)
```

```swift
public struct AttachedMacroEnvironmentKey: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard var binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
```

```swift
public struct AttachedMacroEnvironmentKey: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard var binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
```

```swift
public struct AttachedMacroEnvironmentKey: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard var binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
```

```swift
public struct AttachedMacroEnvironmentKey: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard var binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
```

```swift
public struct AttachedMacroEnvironmentKey: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard var binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
```

```
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.noDefaultArgument))
            return []
        }

        return [
            """
            internal struct EnvironmentKey_\(raw: identifier): EnvironmentKey {
                static var \(binding)
            }
            """
        ]
    }
}
```

```
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.noDefaultArgument))
            return []
        }

        return [
            """
            internal struct EnvironmentKey_\(raw: identifier): EnvironmentKey {
                static var \(binding)
            }
            """
        ]
    }
}
```

```
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.noDefaultArgument))
            return []
        }

        return [
            """
            internal struct EnvironmentKey_\(raw: identifier): EnvironmentKey {
                static var \(binding)
            }
            """
        ]
    }
}
```

```
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.noDefaultArgument))
            return []
        }

        return [
            """
            internal struct EnvironmentKey_\(raw: identifier): EnvironmentKey {
                static var \(binding)
            }
            """
        ]
    }
}
```

```
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        binding.pattern = PatternSyntax(IdentifierPatternSyntax(identifier: .identifier("defaultValue")))

        let isOptionalType = binding.typeAnnotation?.type.is(OptionalTypeSyntax.self) ?? false
        let hasDefaultValue = binding.initializer != nil

        guard isOptionalType || hasDefaultValue else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.noDefaultArgument))
            return []
        }

        return [
            """
            internal struct EnvironmentKey_\(raw: identifier): EnvironmentKey {
                static var \(binding)
            }
            """
        ]
    }
}
```

```
@EnvironmentStorage
extension EnvironmentValues {
    @EnvironmentValue                                   @EnvironmentStorage
    var secondaryFont: Font?
        internal struct EnvironmentKey_secondaryFont: EnvironmentKey {
            static var defaultValue: Font?
        }
    {                                                   @EnvironmentValue
        get {
            self [EnvironmentKey_secondaryFont.self]
        }
        set {
            self [EnvironmentKey_secondaryFont.self] = newValue
        }
    }
}
```

```
@EnvironmentStorage
extension EnvironmentValues {
    @EnvironmentValue                                    @EnvironmentStorage
    var secondaryFont: Font?
        internal struct EnvironmentKey_secondaryFont: EnvironmentKey {
            static var defaultValue: Font?
        }
    {                                                    @EnvironmentValue
        get {
            self [EnvironmentKey_secondaryFont.self]
        }
        set {
            self [EnvironmentKey_secondaryFont.self] = newValue
        }
    }
}
```

```swift
extension AttachedMacroEnvironmentKey: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """
```

```
extension AttachedMacroEnvironmentKey: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """
```

```swift
extension AttachedMacroEnvironmentKey: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """
```

```swift
extension AttachedMacroEnvironmentKey: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """
```

```swift
extension AttachedMacroEnvironmentKey: AccessorMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingAccessorsOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [AccessorDeclSyntax] {

        // Skip declarations other than variables
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """
```

```
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """,
            """
            set {
                self[EnvironmentKey_\(raw: identifier).self] = newValue
            }
            """
        ]
    }
}
```

```
        guard let varDecl = declaration.as(VariableDeclSyntax.self) else {
            return []
        }

        guard let binding = varDecl.bindings.first?.as(PatternBindingSyntax.self)else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.missingAnnotation))
            return []
        }

        guard let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier.text else {
            context.diagnose(Diagnostic(node: Syntax(node), message: Feedback.notAnIdentifier))
            return []
        }

        return [
            """
            get {
                self[EnvironmentKey_\(raw: identifier).self]
            }
            """,
            """
            set {
                self[EnvironmentKey_\(raw: identifier).self] = newValue
            }
            """
        ]
    }
}
```

```
@EnvironmentStorage
extension EnvironmentValues {
    @EnvironmentValue                                   @EnvironmentStorage
    var secondaryFont: Font?
        internal struct EnvironmentKey_secondaryFont: EnvironmentKey {
            static var defaultValue: Font?
        }
    {                                                   @EnvironmentValue
        get {
            self [EnvironmentKey_secondaryFont.self]
        }
        set {
            self [EnvironmentKey_secondaryFont.self] = newValue
        }
    }
}
```

```
@EnvironmentStorage
extension EnvironmentValues {
    @EnvironmentValue                                   @EnvironmentStorage
    var secondaryFont: Font?
        internal struct EnvironmentKey_secondaryFont: EnvironmentKey {
            static var defaultValue: Font?
        }
    {                                                   @EnvironmentValue
        get {
            self [EnvironmentKey_secondaryFont.self]
        }
        set {
            self [EnvironmentKey_secondaryFont.self] = newValue
        }
    }
}
```

#05 – @AddPublisher

https://github.com/ShenghaiWang/SwiftMacros

```
struct MyType {
    @AddPublisher
    private let mySubject = PassthroughSubject<Void, Never>()
}
```

```
struct MyType {
    @AddPublisher
    private let mySubject = PassthroughSubject<Void, Never>()
        var mySubjectPublisher: AnyPublisher<Void, Never> {
            mySubject.eraseToAnyPublisher()
        }
}
```

```
/// Generate a Combine publisher to a Combine subject in order to
avoid overexposing subject variable
@attached(peer, names: suffixed(Publisher))
public macro AddPublisher() = #externalMacro(
    module: "Macros",
    type: “AddPublisher"
)
```

```
/// Generate a Combine publisher to a Combine subject in order to
avoid overexposing subject variable
@attached(peer, names: suffixed(Publisher))
public macro AddPublisher() = #externalMacro(
    module: "Macros",
    type: “AddPublisher"
)
```

```swift
public struct AddPublisher: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {
        guard let variableDecl = declaration.as(VariableDeclSyntax.self),
              variableDecl.modifiers.map({ $0.name.text }).contains("private") else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Please make the subject private and use the automated generated publisher variable outside of this type"
            )
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/CurrentValueSubject) variable declaration"
            )
        }

        let publisher: DeclSyntax =
        """
        var \(raw: identifier.text)Publisher: AnyPublisher<\(genericArgumentClause.arguments)> {
            \(raw: identifier.text).eraseToAnyPublisher()
        }
        """
        return [publisher]
    }
}
```

```swift
public struct AddPublisher: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {
        guard let variableDecl = declaration.as(VariableDeclSyntax.self),
              variableDecl.modifiers.map({ $0.name.text }).contains("private") else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Please make the subject private and use the automated generated publisher variable outside of this type"
            )
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/CurrentValueSubject) variable declaration"
            )
        }
```

```swift
public struct AddPublisher: PeerMacro {
    public static func expansion(
        of node: AttributeSyntax,
        providingPeersOf declaration: some DeclSyntaxProtocol,
        in context: some MacroExpansionContext
    ) throws -> [DeclSyntax] {
        guard let variableDecl = declaration.as(VariableDeclSyntax.self),
              variableDecl.modifiers.map({ $0.name.text }).contains("private") else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Please make the subject private and use the automated generated publisher variable outside of this type"
            )
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/CurrentValueSubject) variable declaration"
            )
        }
```

```
    ) throws -> [DeclSyntax] {
        guard let variableDecl = declaration.as(VariableDeclSyntax.self),
              variableDecl.modifiers.map({ $0.name.text }).contains("private") else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Please make the subject private and use the automated generated publisher variable outside of this type"
            )
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/CurrentValueSubject) variable declaration"
            )
        }

        let publisher: DeclSyntax =
        """
        var \(raw: identifier.text)Publisher: AnyPublisher<\(genericArgumentClause.arguments)> {
            \(raw: identifier.text).eraseToAnyPublisher()
        }
```

```
    ) throws -> [DeclSyntax] {
        guard let variableDecl = declaration.as(VariableDeclSyntax.self),
              variableDecl.modifiers.map({ $0.name.text }).contains("private") else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Please make the subject private and use the automated generated
publisher variable outside of this type"
            )
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/
CurrentValueSubject) variable declaration"
            )
        }

        let publisher: DeclSyntax =
        """
        var \(raw: identifier.text)Publisher: AnyPublisher<\(genericArgumentClause.arguments)> {
            \(raw: identifier.text).eraseToAnyPublisher()
        }
```

```
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/
CurrentValueSubject) variable declaration"
            )
        }

        let publisher: DeclSyntax =
        """
        var \(raw: identifier.text)Publisher: AnyPublisher<\(genericArgumentClause.arguments)> {
            \(raw: identifier.text).eraseToAnyPublisher()
        }
        """
        return [publisher]
    }
}
```

```
        }

        guard let binding = variableDecl.bindings.first,
              let identifier = binding.pattern.as(IdentifierPatternSyntax.self)?.identifier,
              let genericArgumentClause = binding.genericArgumentClause,
              ["PassthroughSubject", "CurrentValueSubject"].contains(binding.typeName) else {
            throw MacroDiagnostics.errorMacroUsage(
                message: "Can only be applied to a subject(PassthroughSubject/
CurrentValueSubject) variable declaration"
            )
        }

        let publisher: DeclSyntax =
        """
        var \(raw: identifier.text)Publisher: AnyPublisher<\(genericArgumentClause.arguments)> {
            \(raw: identifier.text).eraseToAnyPublisher()
        }
        """
        return [publisher]
    }
}
```

```
struct MyType {
    @AddPublisher
    private let mySubject = PassthroughSubject<Void, Never>()
        var mySubjectPublisher: AnyPublisher<Void, Never> {
            mySubject.eraseToAnyPublisher()
        }
}
```

So do you still want to implement your own macros?

Here are a few pitfalls you really want to watch out for!

#01 – Macros operate on a portion of your code

```
struct Person {
    let firstName: String
    let lastName: String
}
```

```
struct Person {
    let firstName: String
    let lastName: String
}

extension Person {
    var fullName: String {
        "\(firstName) \(lastName)"
    }
}
```

```
@KeyPathIterable
struct Person {
    let firstName: String
    let lastName: String
}

extension Person {
    var fullName: String {
        "\(firstName) \(lastName)"
    }
}
```

```swift
@KeyPathIterable
struct Person {
    let firstName: String
    let lastName: String
}

extension Person {
    var fullName: String {
        "\(firstName) \(lastName)"
    }
}
```

```swift
@KeyPathIterable
struct Person {
    let firstName: String
    let lastName: String
}

extension Person {
    var fullName: String {
        "\(firstName) \(lastName)"
    }
}
```

⚠️ The macro `@KeyPathIterable` will NOT see the property `fullName`!

#02 – Macros operate on syntax, not semantics

```
protocol MyProtocol { }
```

```
protocol MyProtocol { }

extension Date: MyProtocol { }
```

```
protocol MyProtocol { }

extension Date: MyProtocol { }

struct User {
    let name: String
    let creationDate: Date
}
```

```swift
protocol MyProtocol { }

extension Date: MyProtocol { }

@MyMacro
struct User {
    let name: String
    let creationDate: Date
}
```

```swift
protocol MyProtocol { }

extension Date: MyProtocol { }

@MyMacro
struct User {
    let name: String
    let creationDate: Date
}
```

```swift
protocol MyProtocol { }

extension Date: MyProtocol { }

@MyMacro
struct User {
    let name: String
    let creationDate: Date
}
```

⚠️ It’s impossible for `@MyMacro` to know which properties conform to `MyProtocol`!

#03 – Macros operate on syntax, not semantics (again!)

```
String? /* means the same than */ Optional<String>
```

```
String? /* means the same than */ Optional<String>
[Int] /* means the same than */ Array<Int>
```

```
String? /* means the same than */ Optional<String>
[Int] /* means the same than */ Array<Int>
() -> Void /* means the same than */ () -> ()
```

String?

- TypeAnnotation
  - :
  - OptionalType
    - IdentifierType
      - **String**
    - **?**

Optional<String>

- TypeAnnotation
  - :
  - IdentifierType
    - **Optional**
    - GenericArgumentClause
      - **<**
      - GenericArgumentList
        - GenericArgument
          - IdentifierType
            - **String**
      - **>**

With Macro Toolkit

```
function.returnsVoid
```

Without Macro Toolkit

```
func returnsVoid(_ function: FunctionDeclSyntax) -> Bool {
    // Function can either have no return type annotation, `()`, `Void`, or a nested single
    // element tuple with a Void-like inner type (e.g. `((((((Void))))))` or `(((((Void)))))`)
    func isVoid(_ type: TypeSyntax) -> Bool {
        if type.description == "Void" || type.description == "()" {
            return true
        }

        guard let tuple = type.as(TupleTypeSyntax.self) else {
            return false
        }

        if let element = tuple.elements.first, tuple.elements.count == 1 {
            let isUnlabeled = element.name == nil && element.secondName == nil
            return isUnlabeled && isVoid(TypeSyntax(element.type))
        }
        return false
    }

    guard let returnType = function.output?.returnType else {
        return false
    }
    return isVoid(returnType)
}
```

https://github.com/stackotter/swift-macro-toolkit

Recap

Recap

Recap

- Swift Macros are incredibly powerful…

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!
- Writing a macro is a different skill than writing app-level code

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!
- Writing a macro is a different skill than writing app-level code
- The community has implemented some great macros…

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!
- Writing a macro is a different skill than writing app-level code
- The community has implemented some great macros…
- …but understanding how they work is really important!

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!
- Writing a macro is a different skill than writing app-level code
- The community has implemented some great macros…
- …but understanding how they work is really important!
- It’s totally possible to roll out your own macro…

Recap

- Swift Macros are incredibly powerful…
- …but they are also incredibly complex!
- Writing a macro is a different skill than writing app-level code
- The community has implemented some great macros…
- …but understanding how they work is really important!
- It’s totally possible to roll out your own macro…
- …but be (very) aware of what you’re getting yourself into!


That's all Folks!

Newsletter

https://www.photoroom.com/company/