§ The Construction Studies

How ARO Was Built: A Technical Examination

* * *

For Whom This Book Is Written

This book is for compiler developers, language designers, and students of compiler construction who want to understand the internal architecture of a real, working language implementation.

We assume familiarity with: - Parsing theory (context-free grammars, recursive descent, operator precedence) - Abstract syntax trees and visitor patterns - Basic compiler pipeline concepts (lexing, parsing, semantic analysis, code generation) - LLVM IR fundamentals - Swift programming language basics

What This Book Is Not

This is not a user guide. We do not explain ARO's syntax or teach you how to write ARO programs. For that, see The Language Guide or the project wiki.

This is not marketing material. We do not argue that ARO is better than other languages. We document what we built, why we built it that way, and where the design falls short.

What You Will Learn

- How a constrained domain-specific language differs architecturally from general-purpose languages
- Practical trade-offs in lexer and parser design
- Event-driven runtime architecture with pub-sub semantics
- LLVM IR generation without the LLVM C++ API
- Swift-C interoperability patterns for language runtimes
- Honest assessment of design decisions that didn't work out

Reading Order

Chapters are designed to be read sequentially, following the compilation pipeline from source text to executable binary. However, each chapter is self-contained enough for reference use.

* * *

ARO Language Project January 2026

Contents

§ Chapter 1: Design Philosophy

The Constraint Hypothesis

ARO operates on a hypothesis that runs counter to mainstream programming language design: **expressiveness and predictability are inversely correlated**. General-purpose languages maximize expressiveness—you can write anything. ARO minimizes it—you can write only certain things in certain ways.

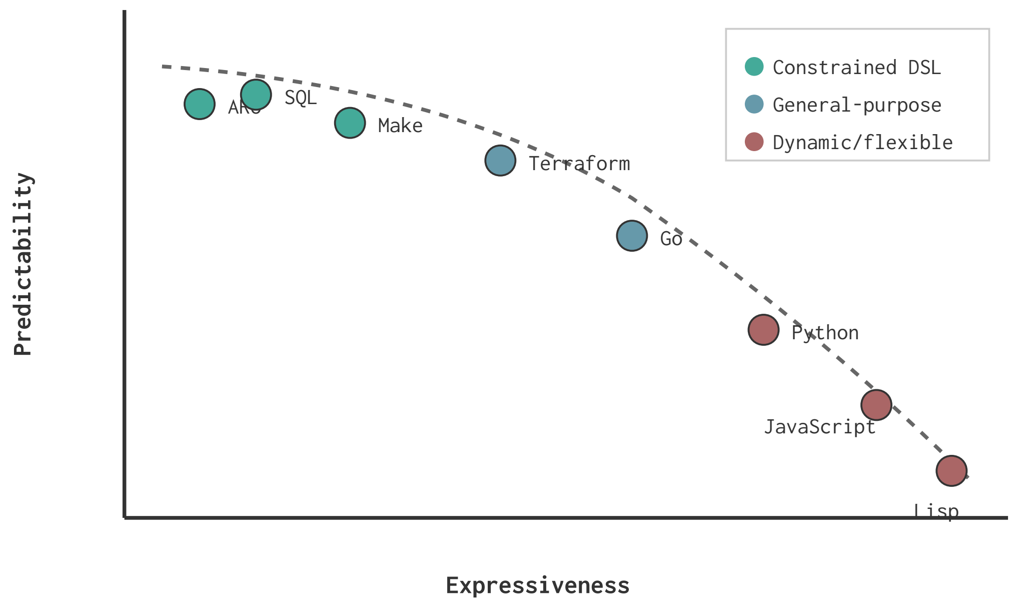


Figure 1

Figure 1.1: The expressiveness-predictability trade-off. Languages cluster along an inverse relationship. ARO occupies the high-predictability, low-expressiveness corner deliberately.

This is not a universal truth—it is a design bet. The bet is that for certain problem domains, the benefits of predictability (uniform tooling, auditable code, consistent execution) outweigh the costs of limited expressiveness.

What "Constraint" Means Architecturally

In a general-purpose language, the AST node types proliferate. Python's AST has over 40 statement types and 20+ expression types. JavaScript's has similar complexity. Each new construct adds parsing rules, semantic analysis passes, and code generation cases.

ARO has five statement types: 1. `AROStatement` (the action-result-object form) 2. `PublishStatement` (variable export) 3. `ForEachStatement` (iteration) 4. `WhenStatement` (conditional guard) 5. `MatchStatement` (pattern matching)

This constraint propagates through the entire implementation: - The parser is simpler (fewer production rules) - Semantic analysis has fewer cases - Code generation is more uniform - Tooling can make stronger assumptions

* * *

Data Flow as Organizing Principle

ARO classifies every action by its data flow direction. This is not just documentation—it is enforced at the type level through the `ActionRole` enum.

```
public enum ActionRole: String, Sendable, CaseIterable {
    case request    // External → Internal
    case own        // Internal → Internal
    case response   // Internal → External
    case export     // Internal → Persistent
}
```

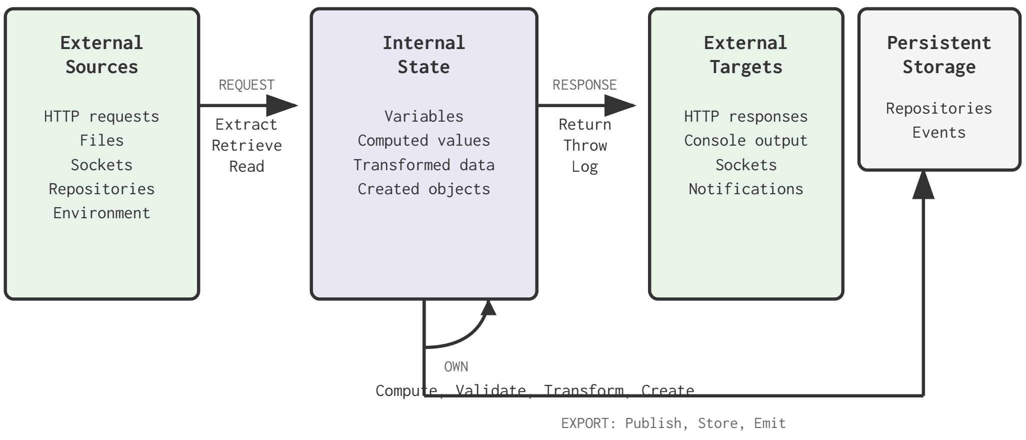


Figure 2

Figure 1.2: Action role data flow. Every action in ARO belongs to exactly one of four roles, determining where data flows.

Why Roles Matter for Implementation

The role classification enables:

1. **Static analysis**: The semantic analyzer can verify that REQUEST actions are sourcing external data and OWN actions are operating on internal state.
2. **Preposition validation**: Each role has valid prepositions. REQUEST actions use `from` (source); RESPONSE actions use `to` (destination).
3. **Code generation**: The LLVM code generator knows which bridge functions to call based on role.
4. **Runtime optimization**: REQUEST actions may be cached; EXPORT actions may be batched.

Implementation reference: `Sources/ARORuntime/Actions/ActionProtocol.swift:12-18`

* * *

Immutability by Default

Variables in ARO cannot be rebound. This is not a convention—it is enforced by both the semantic analyzer and the runtime.

```swift
// SemanticAnalyzer.swift - compile-time check
if symbolTable.contains(name) && !name.hasPrefix("_") {
    diagnostics.append(Diagnostic(
        severity: .error,
        message: "Cannot rebind variable '\(name)' - variables are immutable",
        span: span
    ))
}

// RuntimeContext.swift - runtime safety check (should never trigger)
if bindings[name] != nil && !name.hasPrefix("_") {
    fatalError("Runtime Error: Cannot rebind immutable variable '\(name)'")
}
```

Architectural Consequences

Immutability simplifies the execution model:

1. **No aliasing problems**: If `x` cannot change, you never need to track whether `y` also points to the same mutable value.
2. **Parallel safety**: Immutable bindings are inherently thread-safe. The `Sendable` conformance of `SymbolTable` relies on this.
3. **Simpler code generation**: LLVM IR generation does not need to track which variables might be modified.
4. **Predictable debugging**: The value of a variable at any point is the value it was given when bound.

The escape hatch is the `_` prefix for framework-internal variables, which are exempt from immutability checks.

* * *

The “Code is the Error Message” Philosophy

ARO’s error handling is unusual: there is none. Programmers write only the successful case, and the runtime generates error messages from the source code itself.

```
Statement:
<Retrieve> the <user> from the <user-repository> where id = <id>.

On failure, becomes:
"Cannot retrieve the user from the user-repository where id = 530."
```

This is not a debugging convenience—it is a fundamental design principle with implementation consequences.

Implementation in ErrorReconstructor

```swift
// ErrorReconstructor.swift
public func reconstructError(
    from statement: AROStatement,
    context: ExecutionContext
) -> String {
    var message = "Cannot \(statement.action.text.lowercased()) the \
        (statement.result.base)"

    if let object = statement.object {
        message += " \(object.preposition.rawValue) the \(object.base)"
    }

    // Substitute resolved values
    for (name, value) in context.bindings {
        message = message.replacingOccurrences(of: "<\(name)>", with: "\(value)")
    }

    return message
}
```

Trade-offs

Gained: - Zero error-handling code in ARO programs - Error messages always match the code - Full debugging context in every error

Lost: - No custom error messages (without escape to Throw) - Security-sensitive information may leak - No programmatic error handling

The security issue is real. Error messages expose variable values, repository names, and internal state. ARO is explicitly not designed for production systems handling sensitive data.

Implementation reference: `Sources/ARORuntime/Core/ErrorReconstructor.swift`

* * *

Trade-off Analysis

What ARO Gave Up

Lost Feature	Why It Was Removed	Consequence
General loops	Encourage declarative thinking	Use `for each` or custom actions
Arbitrary functions	Feature sets are not functions	Cannot factor common code easily
Complex conditionals	Guards and match instead	Nested logic requires workarounds
Custom operators	Fixed expression grammar	Cannot extend syntax
Exception handling	Happy path only	Errors terminate execution
Type annotations	OpenAPI schemas only	Limited static typing

What ARO Gained

Gained Property	How It Was Achieved	Benefit
Uniform AST	Five statement types	Simple tooling
Predictable execution	Linear statement flow	Easy debugging
Auditable code	One way to do things	Code review is trivial
Consistent error messages	Code-derived errors	Debugging by reading
Safe concurrency	Immutable bindings	No race conditions in user code

The Central Trade-off

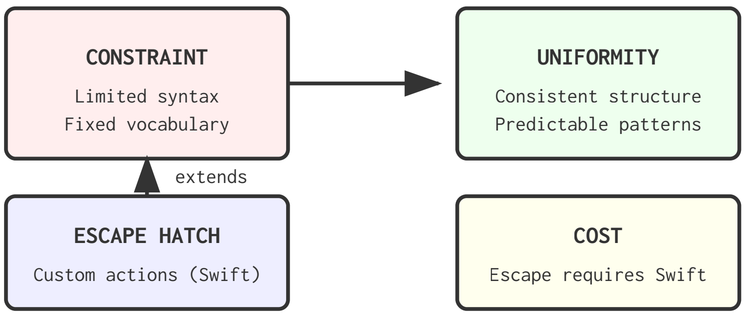


Figure 3

Figure 1.3: The constraint-uniformity trade-off with escape hatch.

The escape hatch (custom actions written in Swift) is essential. Without it, ARO would be too limited for real use. With it, the constraint becomes a default rather than a prison—you stay within ARO's vocabulary unless you genuinely need to escape.

* * *

Chapter Summary

ARO's design philosophy rests on four pillars:

1. **Constraint over expressiveness**: Fewer constructs means simpler implementation and more predictable behavior.
2. **Data flow classification**: Every action has a role that determines its valid operations and enables static analysis.
3. **Immutability by default**: Variables cannot change, eliminating whole categories of bugs and enabling safe concurrency.
4. **Code as error message**: The source code itself becomes the debugging tool, at the cost of security.

These choices have concrete implementation consequences throughout the codebase. The following chapters examine how each compiler phase and runtime component realizes these principles.

* * *

Next: Chapter 2 — Lexical Analysis

§ Chapter 2: Lexical Analysis

The Lexer Architecture

ARO's lexer (`Lexer.swift` , 703 lines) is a hand-written scanner that produces tokens for the parser. It maintains source location tracking, handles string interpolation, and disambiguates between regex literals and division.

```
public final class Lexer: @unchecked Sendable {
    private let source: String
    private var currentIndex: String.Index
    private var location: SourceLocation
    private var tokens: [Token] = []
    private var lastTokenKind: TokenKind?
}
```

The `@unchecked Sendable` annotation is necessary because `String.Index` is not `Sendable` , but the lexer is used single-threaded during parsing.

* * *

Character Classification

The scanner processes characters one at a time, advancing through the source string. Character classification happens in the main `scanToken()` switch statement.

Figure 1

Figure 1

Figure 2.1: Character classification state machine. The lexer advances character by character, branching based on the current character into specialized scanning functions.

* * *

First-Class Language Elements

A distinctive feature of ARO's lexer is that articles and prepositions are first-class token types, not just keywords or identifiers.

```swift
// Token.swift:203-230
public enum Article: String, Sendable, CaseIterable {
    case a = "a"
    case an = "an"
    case the = "the"
}

public enum Preposition: String, Sendable, CaseIterable {
    case from = "from"
    case `for` = "for"
    case against = "against"
    case to = "to"
    case into = "into"
    case via = "via"
    case with = "with"
    case on = "on"
    case at = "at"
    case by = "by"
}
```

Why This Matters

Most languages would treat `from` as either a keyword or an identifier. In ARO, it is a `TokenKind.preposition(.from)`. This distinction enables:

1. **Parser simplification**: The parser can match on `.preposition` directly without string comparisons.
2. **Semantic information in tokens**: `Preposition.indicatesExternalSource` property allows early classification of data sources.
3. **Error messages**: "Expected preposition, found identifier" is more helpful than "Expected 'from', found 'foo'".

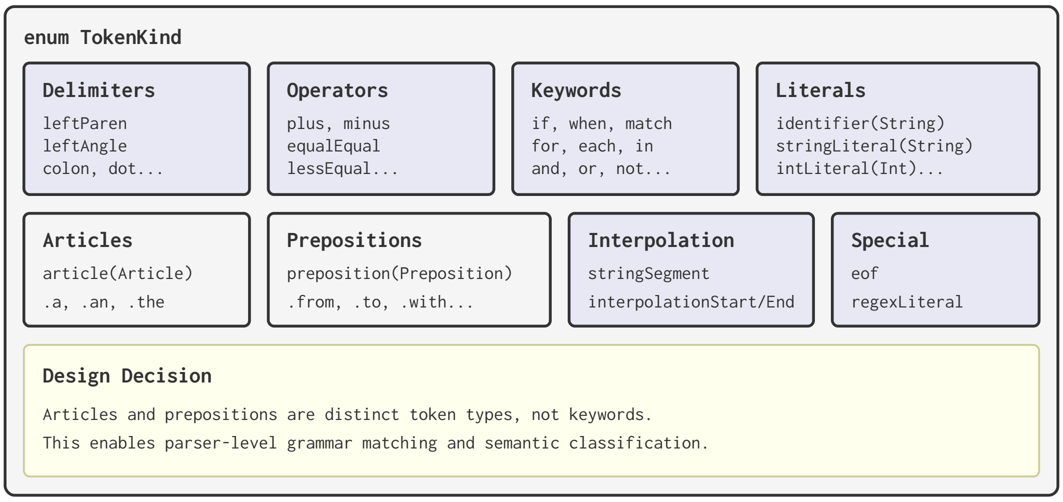


Figure 2

Figure 2.2: Token type hierarchy. Articles and prepositions (green) are separate categories from keywords, enabling grammar-level matching.

* * *

String Interpolation Challenge

String interpolation (`"Hello ${<name>}!"`) requires the lexer to emit multiple tokens for a single string literal. This is handled by a state machine within `scanString()` .

The Problem

A naive approach would produce:

```
stringLiteral("Hello ${<name>}!")  // Wrong: expression is lost
```

ARO needs:

```
stringSegment("Hello ")
interpolationStart
leftAngle
identifier("name")
rightAngle
interpolationEnd
```

Implementation Strategy

```swift
// Lexer.swift:263-276
} else if char == "$" && peekNext() == "{" {
    hasInterpolation = true
    let segmentStart = location
    if !value.isEmpty {
        segments.append((value, segmentStart))
        value = ""
    }
    _ = advance() // $
    _ = advance() // {
    segments.append(("${", location))
    try scanInterpolationContent(quote: quote, start: start, segments: &segments)
}
```

The key insight is that interpolation content is **re-lexed**. The scanner extracts the content between `${` and `}`, then creates a fresh lexer to tokenize it:

```swift
// Lexer.swift:386-393
if let exprTokens = try? Lexer.tokenize(exprContent) {
    for token in exprTokens where token.kind != .eof {
        tokens.append(token)
    }
}
```

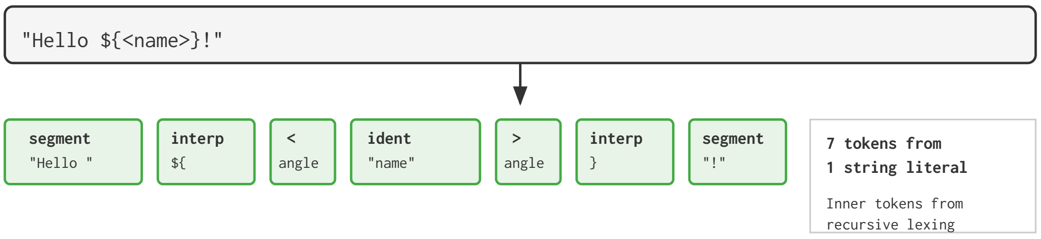


Figure 3

Figure 2.3: Interpolation token sequence. A single interpolated string produces multiple tokens, with the interpolated expression re-lexed recursively.

Nested Brace Handling

Interpolations can contain nested braces (e.g., `${<map>["key"]}`). The lexer tracks brace depth:

```swift
// Lexer.swift:340-360
while !isAtEnd && braceDepth > 0 {
    let char = peek()
    if char == "{" {
        braceDepth += 1
        content.append(advance())
    } else if char == "}" {
        braceDepth -= 1
        if braceDepth > 0 {
            content.append(advance())
        } else {
            _ = advance() // closing }
        }
    }
    // ...
}
```

* * *

Regex vs Division Ambiguity

The character `/` is ambiguous: it could start a regex literal (`/pattern/flags`) or be a division operator (`a / b`). This is a classic lexer challenge that ARO solves with context and lookahead.

The Heuristic

```swift
// Lexer.swift:131-150
let isAfterIdentifier: Bool
if case .identifier = lastTokenKind {
    isAfterIdentifier = true
} else {
    isAfterIdentifier = false
}

let shouldTryRegex = !isAtEnd &&
    peek() != " " && peek() != "\n" && peek() != "\t" &&
    lastTokenKind != .dot &&
    !isAfterIdentifier
```

The rules: 1. If the previous token was an identifier, `/` is division (e.g., `a / b`) 2. If the previous token was `.`, `/` is division (import paths like `../../shared`) 3. If followed by whitespace, `/` is division 4. Otherwise, attempt regex scanning

Regex Scanning with Backtracking

If the heuristic suggests a regex, the lexer attempts to scan it. If scanning fails (no closing `/`), it backtracks:

```swift
// Lexer.swift:513-556
private func tryScanRegex(start: SourceLocation) -> (pattern: String, flags:
        String)? {
    let savedIndex = currentIndex
    let savedLocation = location

    // Attempt to scan pattern...

    if !foundClosingSlash || pattern.isEmpty {
        currentIndex = savedIndex
        location = savedLocation
        return nil
    }

    return (pattern: pattern, flags: flags)
}
```

This is one of the few places in the lexer that requires backtracking. The alternative would be a more complex grammar or forcing regex literals to use different delimiters.

* * *

Source Location Tracking

Every token carries a `SourceSpan` indicating where it came from in the source. This is essential for error reporting.

```swift
// SourceLocation.swift
public struct SourceLocation: Sendable, Equatable {
    public let line: Int      // 1-based
    public let column: Int    // 1-based
    public let offset: Int    // 0-based byte offset
}

public struct SourceSpan: Sendable, Equatable {
    public let start: SourceLocation
    public let end: SourceLocation
}
```

The lexer updates location on every `advance()`:

```
// Lexer.swift:660-665
@discardableResult
private func advance() -> Character {
    let char = source[currentIndex]
    currentIndex = source.index(after: currentIndex)
    location = location.advancing(past: char)
    return char
}
```

The `advancing(past:)` method handles newlines specially, incrementing the line number and resetting the column.

* * *

Keyword Recognition

ARO's keywords are recognized after identifier scanning, not during character classification. This avoids the "keyword vs identifier" problem where a language accidentally reserves useful names.

```swift
// Lexer.swift:572-601
private func scanIdentifierOrKeyword(start: SourceLocation) throws {
    while !isAtEnd && (peek().isLetter || peek().isNumber || peek() == "_") {
        _ = advance()
    }

    let lexeme = String(source[...])
    let lowerLexeme = lexeme.lowercased()

    // Check keywords first
    if let keyword = Self.keywords[lowerLexeme] {
        addToken(keyword, lexeme: lexeme, start: start)
        return
    }

    // Check articles
    if let article = Article(rawValue: lowerLexeme) {
        addToken(.article(article), lexeme: lexeme, start: start)
        return
    }

    // Check prepositions
    if let preposition = Preposition(rawValue: lowerLexeme) {
        addToken(.preposition(preposition), lexeme: lexeme, start: start)
        return
    }

    // Regular identifier
    addToken(.identifier(lexeme), lexeme: lexeme, start: start)
}
```

Note that keyword matching is case-insensitive (`lowerLexeme`), but the original case is preserved in the lexeme for error messages.

* * *

Comment Handling

ARO supports two comment styles: - Block comments: `(* comment *)` - Line comments: `// comment`

Comments are skipped entirely; they do not produce tokens:

```
// Lexer.swift:605-640
private func skipWhitespaceAndComments() {
    while !isAtEnd {
        let char = peek()
        if char.isWhitespace {
            _ = advance()
        } else if char == "(" && peekNext() == "*" {
            skipBlockComment()
        } else if char == "/" && peekNext() == "/" {
            skipLineComment()
        } else {
            break
        }
    }
}
```

Block comments use `(* *)` rather than `/* */` to avoid ambiguity with the star operator in multiplication expressions.

* * *

Error Handling

Lexer errors are thrown as `LexerError`:

```
public enum LexerError: Error {
    case unexpectedCharacter(Character, at: SourceLocation)
    case unterminatedString(at: SourceLocation)
    case invalidEscapeSequence(Character, at: SourceLocation)
    case invalidUnicodeEscape(String, at: SourceLocation)
    case invalidNumber(String, at: SourceLocation)
}
```

Each error carries the source location where it occurred, enabling precise error reporting:

```
Error: Unterminated string literal
  at line 5, column 12
```

* * *

Chapter Summary

ARO’s lexer demonstrates several design choices:

1. **Articles and prepositions as token types**: Enables grammar-level matching rather than string comparison in the parser.
2. **String interpolation via recursive lexing**: The content of `${...}` is extracted and re-tokenized, producing multiple tokens from a single string.
3. **Regex/division disambiguation**: Uses context (previous token) and backtracking to resolve the `/` ambiguity.
4. **Source location on every token**: Enables precise error messages throughout the compilation pipeline.

The lexer is 703 lines—small for a language implementation. The constrained syntax (no user-defined operators, fixed token types) keeps it manageable.

Implementation reference: `Sources/AROParser/Lexer.swift`

* * *

Next: Chapter 3 — Syntactic Analysis

§ Chapter 3: Syntactic Analysis

Hybrid Parser Design

ARO's parser (`Parser.swift` , 1700+ lines) uses a hybrid approach: **recursive descent** for statements and program structure, **Pratt parsing** for expressions. This combination leverages the strengths of each technique.

```
public final class Parser {
    private let tokens: [Token]
    private var current: Int = 0
    private let diagnostics: DiagnosticCollector
}
```

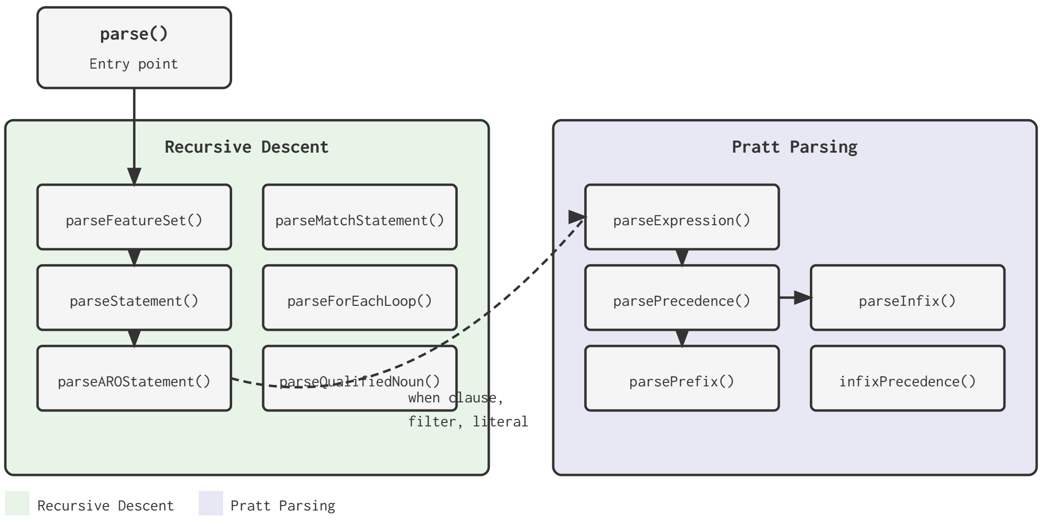


Figure 1

Figure 3.1: Parser architecture. Recursive descent handles statements; Pratt parsing handles expressions. The dashed line shows where statement parsing calls into expression parsing.

* * *

Why Recursive Descent for Statements

ARO's statement syntax has a predictable structure that maps naturally to recursive descent:

```
FeatureSet  ::= "(" name ":" activity ")" "{" Statement* "}"
Statement   ::= AROStatement | MatchStatement | ForEachLoop
AROStatement ::= "<" verb ">" [article] "<" result ">" preposition [article] "<"
object ">" "."
```

Each grammar rule becomes a parsing function:

```swift
// Parser.swift:119-159
private func parseFeatureSet() throws -> FeatureSet {
    let startToken = try expect(.leftParen, message: "'('")

    let name = try parseIdentifierSequence()
    try expect(.colon, message: "':'")
    let activity = try parseIdentifierSequence()

    try expect(.rightParen, message: "')'")
    try expect(.leftBrace, message: "'{'")

    var statements: [Statement] = []
    while !check(.rightBrace) && !isAtEnd {
        let statement = try parseStatement()
        statements.append(statement)
    }

    let endToken = try expect(.rightBrace, message: "'}'")
    return FeatureSet(name: name, businessActivity: activity, statements:
        statements, span: ...)
}
```

The advantages for statements: 1. **One-to-one mapping**: Each grammar rule is a function 2. **Natural error handling**: Try-catch at each level 3. **Easy to extend**: Adding a new statement type is adding a function

* * *

Why Pratt for Expressions

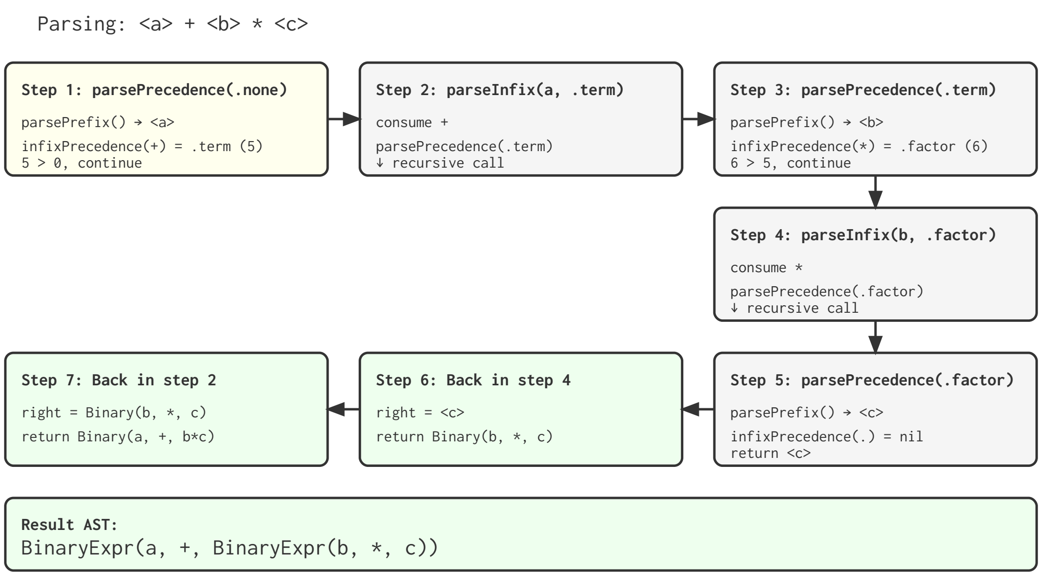
Expression parsing requires handling operator precedence. Consider: `<a> + <b> * <c>`. This should parse as `<a> + (<b> * <c>)` because multiplication binds tighter than addition.

Pratt parsing (also called "top-down operator precedence") handles this elegantly with precedence levels:

```
// Parser.swift:1227-1241
private enum Precedence: Int, Comparable {
    case none = 0
    case or = 1             // or
    case and = 2            // and
    case equality = 3       // == != is
    case comparison = 4     // < > <= >=
    case term = 5           // + - ++
    case factor = 6         // * / %
    case unary = 7          // - not
    case postfix = 8        // . []
}
```

The core loop is remarkably simple:

```
// Parser.swift:1252-1269
private func parsePrecedence(_ minPrecedence: Precedence) throws -> any Expression {
    // Parse prefix (primary or unary)
    var left = try parsePrefix()

    // Parse infix operators at or above minPrecedence
    while let prec = infixPrecedence(peek()), prec > minPrecedence {
        left = try parseInfix(left: left, precedence: prec)
    }

    // Handle postfix existence check: <expr> exists
    if check(.exists) {
        advance()
        left = ExistenceExpression(expression: left, span: left.span)
    }

    return left
}
```



```
Parsing: <a> + <b> * <c>

Step 1: parsePrecedence(.none)
parsePrefix() → <a>
infixPrecedence(+) = .term (5)
5 > 0, continue

Step 2: parseInfix(a, .term)
consume +
parsePrecedence(.term)
↓ recursive call

Step 3: parsePrecedence(.term)
parsePrefix() → <b>
infixPrecedence(*) = .factor (6)
6 > 5, continue

Step 4: parseInfix(b, .factor)
consume *
parsePrecedence(.factor)
↓ recursive call

Step 5: parsePrecedence(.factor)
parsePrefix() → <c>
infixPrecedence(.) = nil
return <c>

Step 6: Back in step 4
right = <c>
return Binary(b, *, c)

Step 7: Back in step 2
right = Binary(b, *, c)
return Binary(a, +, b*c)

Result AST:
BinaryExpr(a, +, BinaryExpr(b, *, c))
```

Figure 2

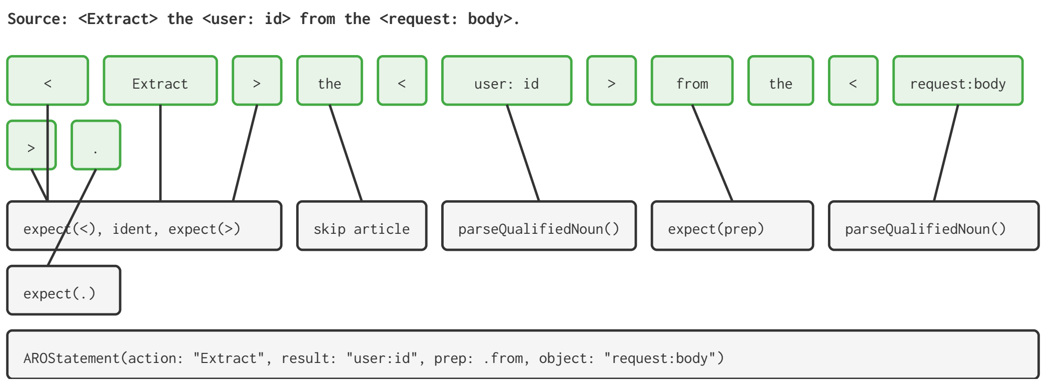
Figure 3.2: Precedence climbing for `<a> + <b> * <c>`. The key insight: when parsing the right side of `+`, we call `parsePrecedence(.term)`. This means `*` (which has higher precedence) will be consumed, but `+` (same or lower) would not.

* * *

The AROStatement Parse

The core statement form is: `<Action> [article] <Result> preposition [article] <Object> [clauses] .`

```swift
// Parser.swift:197-350 (simplified)
private func parseAROStatement(startToken: Token) throws -> AROStatement {
    // Parse action verb
    let actionToken = try expectIdentifier(message: "action verb")
    let action = Action(verb: actionToken.lexeme, span: actionToken.span)
    try expect(.rightAngle, message: "'>'")

    // Skip optional article before result
    if case .article = peek().kind { advance() }

    // Parse result
    try expect(.leftAngle, message: "'<'")
    let result = try parseQualifiedNoun()
    try expect(.rightAngle, message: "'>'")

    // Parse preposition
    guard case .preposition(let prep) = peek().kind else {
        throw ParserError.unexpectedToken(expected: "preposition", got: peek())
    }
    advance()

    // Skip optional article before object
    if case .article = peek().kind { advance() }

    // Parse object
    try expect(.leftAngle, message: "'<'")
    let objectNoun = try parseQualifiedNoun()
    try expect(.rightAngle, message: "'>'")

    // Parse optional clauses (where, when)
    // ...

    try expect(.dot, message: "'.'")

    return AROStatement(action: action, result: result, preposition: prep, object:
        objectNoun, ...)
}
```



Source: <Extract> the <user: id> from the <request: body>.
<
Extract
>
the
<
user: id
>
from
the
<
request:body
>
.
expect(<), ident, expect(>)
skip article
parseQualifiedNoun()
expect(prep)
parseQualifiedNoun()
expect(.)
AROStatement(action: "Extract", result: "user:id", prep: .from, object: "request:body")

Figure 3

Figure 3.3: Statement parsing flowchart. Each token is consumed in order, with optional articles skipped.

* * *

Error Recovery Strategy

When parsing fails, the parser needs to recover and continue. ARO uses **synchronization points** to find safe places to resume.

Feature Set Level Recovery

```swift
// Parser.swift:1191-1199
private func synchronize() {
    while !isAtEnd {
        // Look for the start of a new feature set
        if check(.leftParen) {
            return
        }
        advance()
    }
}
```

If parsing a feature set fails, skip tokens until we find `(` which starts the next feature set.

Statement Level Recovery

```
// Parser.swift:1201-1221
private func synchronizeToNextStatement() {
    while !isAtEnd {
        // If we just passed a dot, we're at a new statement
        if previous().kind == .dot {
            return
        }

        // If we see a closing brace, stop
        if check(.rightBrace) {
            return
        }

        // If we see an opening angle bracket, we might be at a new statement
        if check(.leftAngle) {
            return
        }

        advance()
    }
}
```

The synchronization points are: 1. `.` (statement terminator—most reliable) 2. `}` (feature set end) 3. `<` (possible statement start)

Diagnostic Collection

Errors are collected rather than immediately thrown:

```
// Parser.swift:44-51
while !isAtEnd {
    do {
        let featureSet = try parseFeatureSet()
        featureSets.append(featureSet)
    } catch let error as ParserError {
        diagnostics.report(error)  // Collect, don't abort
        synchronize()              // Recover
    }
}
```

This enables reporting multiple errors in a single parse pass.

* * *

Single Lookahead Limitation

ARO’s parser uses single-token lookahead (`peek()` returns the current token, `advance()` moves forward). This creates disambiguation challenges.

The `<` Ambiguity

The character `<` can mean: - Start of a variable reference: `<user>` - Less-than operator: `<a> < <b>` - Start of an action: `<Extract>`

The parser uses context to disambiguate:

```swift
// Parser.swift:1367-1376
case .leftAngle, .rightAngle:
    // Only treat as comparison if not followed by identifier
    let nextIndex = current + 1
    if nextIndex < tokens.count {
        if case .identifier = tokens[nextIndex].kind {
            // This could be starting a variable ref, don't treat as comparison
            return nil
        }
    }
    return .comparison
```

The `.` Ambiguity

The character `.` can mean: - Member access: `<user>.name` - Statement terminator: `... object>.`

```swift
// Parser.swift:1381-1389
case .dot:
    // Only treat . as member access if followed by identifier
    let nextIndex = current + 1
    if nextIndex < tokens.count {
        if case .identifier = tokens[nextIndex].kind {
            return .postfix
        }
    }
    return nil  // Statement terminator
```

The `for` Ambiguity

The keyword `for` is both: - The preposition `.for` (“for the user”) - The loop keyword `.for` (“for each”)

```swift
// Parser.swift:252-256
} else if case .for = peek().kind {
    // Accept "for" keyword as the preposition .for
    prep = .for
    advance()
}
```

These disambiguations work because ARO’s grammar is constrained. A more complex language would need multi-token lookahead or backtracking.

* * *

Qualified Noun Parsing

The `QualifiedNoun` is a core concept: `base` optionally followed by `:` `specifier`.

```swift
// Parser.swift:950-967
private func parseQualifiedNoun() throws -> QualifiedNoun {
    let startToken = peek()
    let base = try parseCompoundIdentifier()
    var typeAnnotation: String? = nil

    if check(.colon) {
        advance()
        typeAnnotation = try parseTypeAnnotation()
    }

    return QualifiedNoun(
        base: base,
        typeAnnotation: typeAnnotation,
        span: startToken.span.merged(with: previous().span)
    )
}
```

Compound Identifiers

ARO allows hyphenated identifiers: `user-service`, `created-at`, `first-name`.

```swift
// Parser.swift:1069-1079
private func parseCompoundIdentifier() throws -> String {
    var result = try expectIdentifier(message: "identifier").lexeme

    while check(.hyphen) {
        advance()
        result += "-"
        result += try expectIdentifier(message: "identifier after '-'").lexeme
    }

    return result
}
```

This is implemented in the parser, not the lexer. The lexer produces separate tokens (`user` , `-` , `service`), and the parser combines them.

* * *

Identifier Sequence Parsing

Feature set names and business activities are space-separated identifiers:

```
(User Authentication: Security and Access Control)
 └─────────────────┘  └──────────────────────────┘
      name                  business activity
```

```swift
// Parser.swift:1084-1120
private func parseIdentifierSequence() throws -> String {
    var parts: [String] = []

    while peek().kind.isIdentifierLike {
        var compound = advance().lexeme

        // Handle hyphens within identifiers
        while check(.hyphen) {
            advance()
            compound += "-"
            if peek().kind.isIdentifierLike {
                compound += advance().lexeme
            }
        }

        parts.append(compound)
    }

    return parts.joined(separator: " ")
}
```

The `isIdentifierLike` property allows certain keywords (like `Error`) to appear in names:

```swift
// Token.swift:260-270
public var isIdentifierLike: Bool {
    switch self {
    case .identifier:
        return true
    case .error, .match, .case, .otherwise, .if, .else:
        return true
    default:
        return false
    }
}
```

* * *

Chapter Summary

ARO's parser demonstrates several techniques:

1. **Hybrid design**: Recursive descent for statements (clear structure), Pratt for expressions (elegant precedence).

2. **Single lookahead with context**: Disambiguates `<`, `.`, `for` based on what follows.

3. **Error recovery via synchronization**: Finds safe restart points (`.` , `}` , `<`) to continue after errors.

4. **Compound identifiers in parser**: Hyphenated names are assembled from separate tokens.

The parser is 1700+ lines—larger than the lexer but still manageable. The constrained grammar (five statement types, fixed expression operators) keeps complexity bounded.

Implementation reference: `Sources/AROParser/Parser.swift`

* * *

Next: Chapter 4 — Abstract Syntax

§ Chapter 4: Abstract Syntax

AST Node Hierarchy

ARO's AST (`AST.swift` , 1315 lines) defines the tree structure produced by parsing. Every node conforms to `ASTNode` , which requires `Sendable` (Swift concurrency safety), source location tracking, and visitor pattern support.

```swift
public protocol ASTNode: Sendable, Locatable, CustomStringConvertible {
    func accept<V: ASTVisitor>(_ visitor: V) throws -> V.Result
}
```

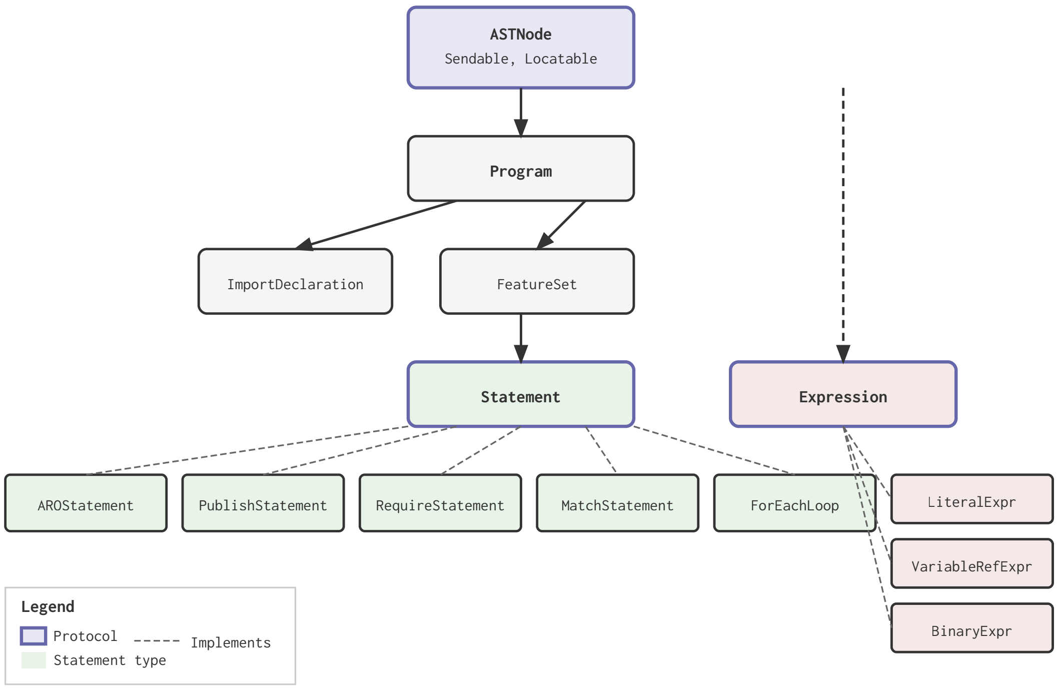


Figure 1

Figure 4.1: Complete AST node hierarchy. `Program` contains `FeatureSet` s, which contain `Statement` s. `Expression` is a parallel hierarchy used within statements.

* * *

Statement vs Expression Dichotomy

ARO distinguishes between statements (actions that do things) and expressions (values that can be computed). This is not just syntactic—it reflects the language's philosophy.

Statements

Statements perform actions and produce side effects:

```
public protocol Statement: ASTNode {}
```

The five statement types:

Statement	Purpose	Example
`AROStatement`	Core action-result-object	`<Extract> the <user> from the <request>.`
`PublishStatement`	Variable export	`<Publish> as <alias> <variable>.`
`RequireStatement`	Dependency declaration	`<Require> the <config> from the <environment>.`
`MatchStatement`	Pattern matching	`match <status> { case "active" { ... } }`
`ForEachLoop`	Iteration	`for each <item> in <items> { ... }`

Expressions

Expressions compute values without side effects:

```
public protocol Expression: ASTNode {}
```

Expression types include: - `LiteralExpression` — constants - `VariableRefExpression` — variable access - `BinaryExpression` — operators - `UnaryExpression` — negation, not - `MemberAccessExpression` — `.property` - `SubscriptExpression` — `[index]`

Why the Distinction Matters

1. **Semantic analysis**: Statements create variables; expressions read them.
2. **Code generation**: Statements map to bridge calls; expressions map to LLVM instructions.
3. **Error messages**: “Cannot execute statement” vs “Cannot evaluate expression”.

* * *

The QualifiedNoun Pattern

`QualifiedNoun` is ARO’s representation of a variable reference with optional type information.

```swift
// AST.swift:519-570
public struct QualifiedNoun: Sendable, Equatable, CustomStringConvertible {
    public let base: String
    public let typeAnnotation: String?
    public let span: SourceSpan

    public var specifiers: [String] {
        guard let type = typeAnnotation else { return [] }
        if type.contains("<") {
            return [type]  // Generic type like List<User>
        }
        return type.split(separator: ".").map(String.init)
    }

    public var fullName: String {
        if let type = typeAnnotation {
            return "\(base): \(type)"
        }
        return base
    }
}
```

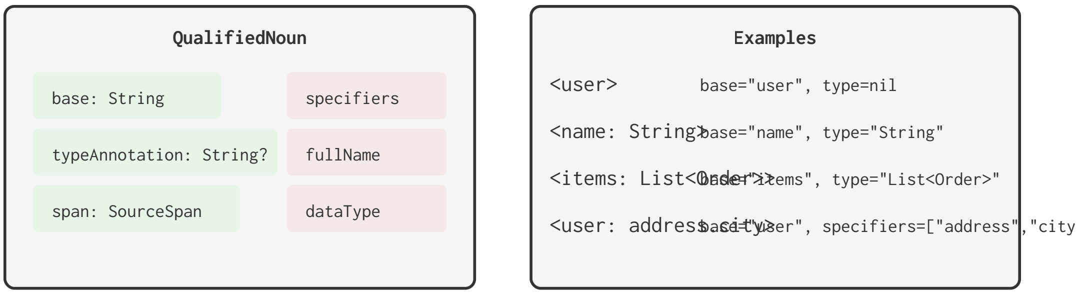


Figure 2

Figure 4.2: QualifiedNoun structure. Stored properties (green) hold the parsed data; computed properties (red) derive useful forms.

Design Rationale

The `QualifiedNoun` pattern serves multiple purposes:

1. **Variable naming**: `base` becomes the variable name
2. **Type annotation**: `typeAnnotation` provides optional type hints
3. **Property access**: `specifiers` enable nested property paths like `user.address.city`
4. **Generic types**: Handles `List<User>` without parsing complications

* * *

Visitor Pattern Implementation

ARO uses the classic visitor pattern to decouple AST traversal from AST structure.

```swift
// AST.swift:974-999
public protocol ASTVisitor {
    associatedtype Result

    func visit(_ node: Program) throws -> Result
    func visit(_ node: ImportDeclaration) throws -> Result
    func visit(_ node: FeatureSet) throws -> Result
    func visit(_ node: AROStatement) throws -> Result
    func visit(_ node: PublishStatement) throws -> Result
    func visit(_ node: RequireStatement) throws -> Result
    func visit(_ node: MatchStatement) throws -> Result
    func visit(_ node: ForEachLoop) throws -> Result

    // Expression visitors
    func visit(_ node: LiteralExpression) throws -> Result
    func visit(_ node: ArrayLiteralExpression) throws -> Result
    func visit(_ node: MapLiteralExpression) throws -> Result
    func visit(_ node: VariableRefExpression) throws -> Result
    func visit(_ node: BinaryExpression) throws -> Result
    func visit(_ node: UnaryExpression) throws -> Result
    func visit(_ node: MemberAccessExpression) throws -> Result
    func visit(_ node: SubscriptExpression) throws -> Result
    func visit(_ node: GroupedExpression) throws -> Result
    func visit(_ node: ExistenceExpression) throws -> Result
    func visit(_ node: TypeCheckExpression) throws -> Result
    func visit(_ node: InterpolatedStringExpression) throws -> Result
}
```

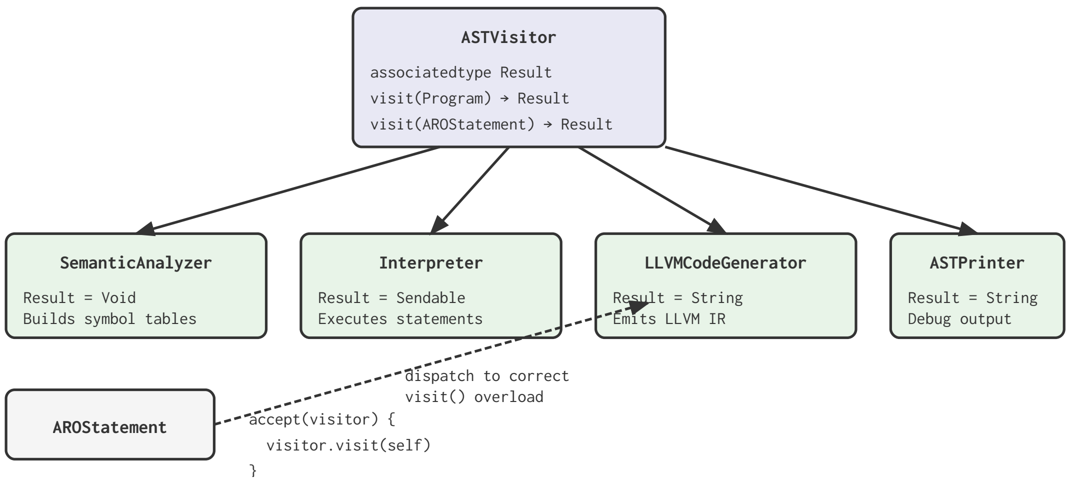


Figure 3

Figure 4.3: Visitor pattern classes. Each visitor implementation provides different behavior for the same AST structure.

Default Implementations

For visitors that only need to traverse (not transform), default implementations recurse into children:

```swift
// AST.swift:1001-1020
public extension ASTVisitor where Result == Void {
    func visit(_ node: Program) throws {
        for importDecl in node.imports {
            try importDecl.accept(self)
        }
        for featureSet in node.featureSets {
            try featureSet.accept(self)
        }
    }

    func visit(_ node: FeatureSet) throws {
        for statement in node.statements {
            try statement.accept(self)
        }
    }
    // ...
}
```

This means concrete visitors only need to override the nodes they care about.

* * *

Sendable Conformance

Swift 6 requires types shared across concurrency boundaries to conform to `Sendable`. ARO's AST nodes are all `Sendable`:

```swift
public struct Program: ASTNode { ... }        // Implicitly Sendable
public struct FeatureSet: ASTNode { ... }     // All fields are Sendable
public struct AROStatement: Statement { ... } // Sendable via stored properties
```

The Expression Challenge

Expressions use `any Expression` which is existential, not directly `Sendable`. The workaround:

```
public struct AROStatement: Statement {
    public let expression: (any Expression)?
    // ...
}
```

The compiler accepts this because `Expression` inherits from `ASTNode` which requires `Sendable`.

Why This Matters

Semantic analysis and code generation can run concurrently on different feature sets. `Sendable` AST nodes enable this safely.

* * *

Span Propagation

Every AST node carries a `SourceSpan` indicating its location in source code.

```
public struct SourceSpan: Sendable, Equatable {
    public let start: SourceLocation
    public let end: SourceLocation

    public func merged(with other: SourceSpan) -> SourceSpan {
        SourceSpan(start: start, end: other.end)
    }
}
```

Spans propagate through parsing via `merged(with:)`:

```
// When parsing: <Extract> the <user> from the <request>.
let startToken = try expect(.leftAngle, ...)  // start location
// ... parse contents ...
let endToken = try expect(.dot, ...)          // end location

return AROStatement(
    // ...
    span: startToken.span.merged(with: endToken.span)
)
```

Error Reporting

Spans enable precise error messages:

```
Error: Cannot retrieve the user from the user-repository
  at line 5, columns 4-52

  5 |     <Retrieve> the <user> from the <user-repository>.
    |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The underline is computed from `span.start.column` to `span.end.column` .

* * *

AROStatement Structure

The core statement type has grown complex to accommodate language features:

```swift
// AST.swift:98-151
public struct AROStatement: Statement {
    public let action: Action
    public let result: QualifiedNoun
    public let object: ObjectClause
    public let literalValue: LiteralValue?      // with "string"
    public let expression: (any Expression)?     // computed value
    public let aggregation: AggregationClause?   // sum(<field>)
    public let whereClause: WhereClause?         // where <field> == value
    public let byClause: ByClause?               // by /delimiter/
    public let toClause: (any Expression)?       // date ranges
    public let withClause: (any Expression)?     // set operations
    public let whenCondition: (any Expression)?  // guards
    public let resultExpression: (any Expression)? // sink syntax
    public let span: SourceSpan
}
```

This is a design smell—the struct has 13 optional fields. A cleaner design might use an enum of clause types. But the current structure is simpler to parse and execute.

* * *

Action Semantic Classification

Actions carry their semantic role for static analysis:

```swift
// AST.swift:488-508
public enum ActionSemanticRole: String, Sendable, CaseIterable {
    case request    // External → Internal
    case own        // Internal → Internal
    case response   // Internal → External
    case export     // Makes available to other features

    public static func classify(verb: String) -> ActionSemanticRole {
        let lower = verb.lowercased()

        let requestVerbs = ["extract", "parse", "retrieve", "fetch", ...]
        let responseVerbs = ["return", "throw", "send", "emit", ...]
        let exportVerbs = ["publish", "export", "expose", "share"]

        if requestVerbs.contains(lower) { return .request }
        if responseVerbs.contains(lower) { return .response }
        if exportVerbs.contains(lower) { return .export }
        return .own
    }
}
```

This classification is used by: - Semantic analyzer for data flow validation - Runtime for choosing execution strategy - LLVM code generator for bridge function selection

* * *

Chapter Summary

ARO's AST design reflects the language's constraints:

1. **Five statement types**: The uniform structure enables simple tooling.
2. **QualifiedNoun pattern**: Handles variable naming, type annotation, and property access in one structure.
3. **Visitor pattern**: Decouples traversal from structure; enables semantic analysis, interpretation, and code generation with the same AST.
4. **Sendable throughout**: Swift 6 concurrency safety is enforced at compile time.
5. **Span propagation**: Every node knows its source location for error reporting.

The AST is 1315 lines—larger than it would be for a minimal language, but manageable. The complexity comes from supporting clauses (`where` , `when` , `by`) that extend the basic action-result-object form.

Implementation reference: `Sources/AROParser/AST.swift`

* * *

Next: Chapter 5 — Semantic Analysis

§ Chapter 5: Semantic Analysis

Overview

Semantic analysis (`SemanticAnalyzer.swift`) bridges parsing and execution. It builds symbol tables, tracks data flow, enforces immutability, and detects problems that parsing alone cannot catch.

```swift
public final class SemanticAnalyzer {
    private let diagnostics: DiagnosticCollector
    private let globalRegistry: GlobalSymbolRegistry

    public func analyze(_ program: Program) -> AnalyzedProgram {
        // Four-pass analysis
    }
}
```

The analyzer performs four passes: 1. Build symbol tables and detect duplicates 2. Verify external dependencies 3. Detect circular event chains 4. Detect orphaned event emissions

* * *

Symbol Table Design

Each feature set gets its own symbol table, built during the first pass.

```swift
// SemanticAnalyzer.swift:29-50
public struct AnalyzedFeatureSet: Sendable {
    public let featureSet: FeatureSet
    public let symbolTable: SymbolTable
    public let dataFlows: [DataFlowInfo]
    public let dependencies: Set<String>  // External dependencies
    public let exports: Set<String>       // Published symbols
}
```

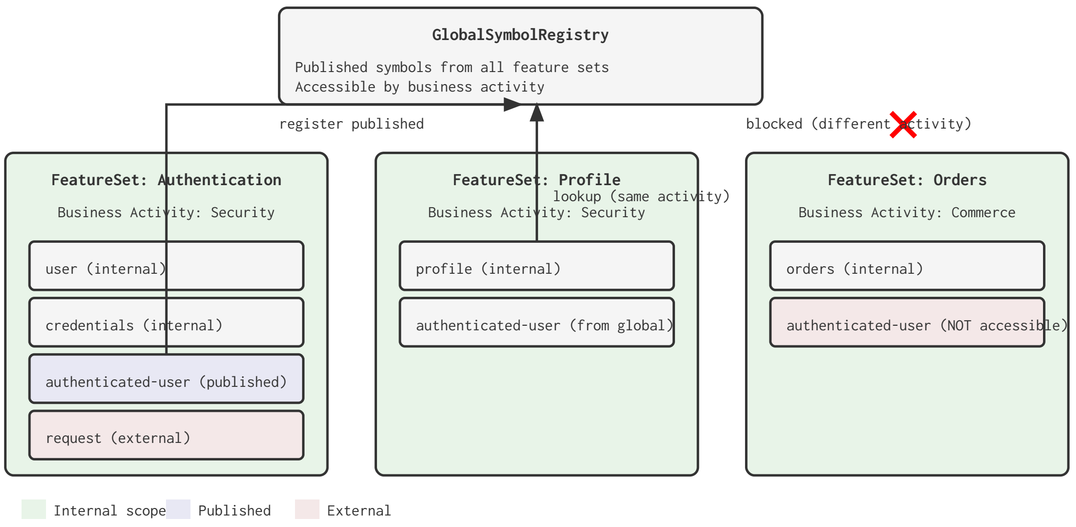

GlobalSymbolRegistry
Published symbols from all feature sets
Accessible by business activity
register published
blocked (different activity)
FeatureSet: Authentication
Business Activity: Security
user (internal)
credentials (internal)
authenticated-user (published)
request (external)
FeatureSet: Profile
lookup (same activity)
Business Activity: Security
profile (internal)
authenticated-user (from global)
FeatureSet: Orders
Business Activity: Commerce
orders (internal)
authenticated-user (NOT accessible)
Internal scope
Published
External

Figure 1

Figure 5.1: Symbol table scope hierarchy. Published symbols are registered globally but only accessible within the same business activity.

* * *

Visibility Levels

Symbols have three visibility levels:

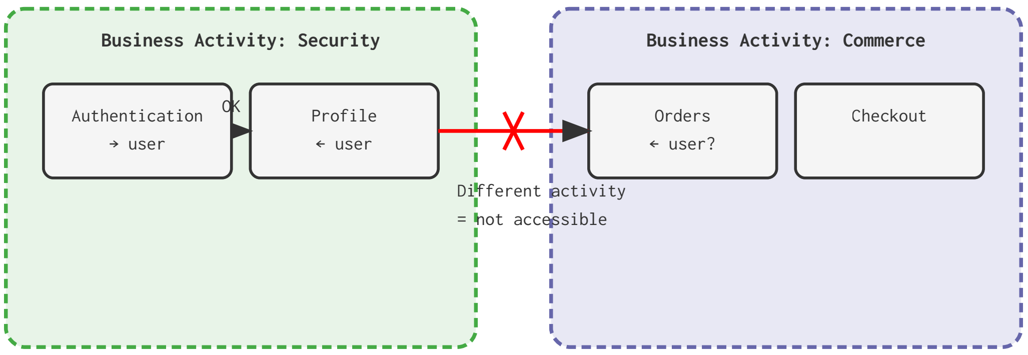
```
public enum SymbolVisibility: Sendable {
    case `internal`   // Private to feature set
    case published    // Exported via Publish
    case external     // Provided by runtime
}
```

Visibility	Created By	Accessible From
`internal`	AROStatement result	Same feature set only
`published`	PublishStatement	Same business activity
`external`	Runtime (request, context)	Any feature set

Business Activity Isolation

Published symbols are scoped by business activity—not globally visible:

```
// SemanticAnalyzer.swift:104-106
for symbol in analyzed.symbolTable.publishedSymbols.values {
    globalRegistry.register(symbol: symbol, fromFeatureSet: featureSet.name)
}
```



Business Activity: Security
Authentication → user
OK
Profile ← user
Different activity = not accessible
Business Activity: Commerce
Orders ← user?
Checkout

Figure 2

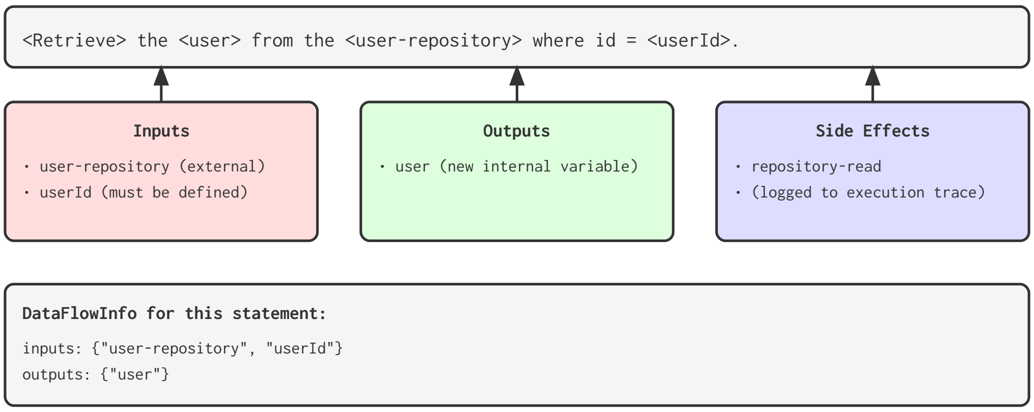
Figure 5.2: Business activity boundaries. The `user` symbol published in “Security” is only visible to other feature sets in “Security”, not to “Commerce”.

* * *

Data Flow Classification

The analyzer tracks what each statement consumes and produces:

```
// SemanticAnalyzer.swift:11-25
public struct DataFlowInfo: Sendable, Equatable {
    public let inputs: Set<String>      // Variables consumed
    public let outputs: Set<String>     // Variables produced
    public let sideEffects: [String]    // External effects
}
```



<Retrieve> the <user> from the <user-repository> where id = <userId>.
Inputs
· user-repository (external)
· userId (must be defined)
Outputs
· user (new internal variable)
Side Effects
· repository-read
· (logged to execution trace)
DataFlowInfo for this statement:
inputs: {"user-repository", "userId"}
outputs: {"user"}

Figure 3

Figure 5.3: Data flow analysis for a single statement. The analyzer determines inputs (must exist), outputs (will be created), and side effects.

Action Role Determines Flow

```swift
// SemanticAnalyzer.swift:292-330
switch statement.action.semanticRole {
case .request:
    // REQUEST: external -> internal
    inputs.insert(objectName)
    outputs.insert(resultName)

case .own:
    // OWN: internal -> internal
    inputs.insert(objectName)
    outputs.insert(resultName)

case .response:
    // RESPONSE: internal -> external
    inputs.insert(resultName)
    inputs.insert(objectName)
    sideEffects.append("response")

case .export:
    // EXPORT: internal -> persistent
    inputs.insert(resultName)
    sideEffects.append("export-\(objectName)")
}
```

* * *

Immutability Enforcement

ARO enforces that variables cannot be rebound. This is checked during analysis:

```swift
// SemanticAnalyzer.swift:194-204
private func isInternalVariable(_ name: String) -> Bool {
    return name.hasPrefix("_")
}

private func isRebindingAllowed(_ verb: String) -> Bool {
    let rebindingVerbs: Set<String> = ["accept", "update", "modify", "change",
         "set"]
    return rebindingVerbs.contains(verb.lowercased())
}
```

When a statement would create a variable that already exists:

```swift
// Check for immutability violation
if definedSymbols.contains(resultName) &&
   !isInternalVariable(resultName) &&
   !isRebindingAllowed(statement.action.verb) {
    diagnostics.error(
        "Cannot rebind variable '\(resultName)' - variables are immutable",
        at: statement.span.start
    )
}
```

Special Cases

Variable	Can Rebind?	Reason
`user`	No	Normal variable
`_internal`	Yes	Framework prefix
With `Update`	Yes	Explicit rebind verb
With `Accept`	Yes	State transition verb

* * *

Unused Variable Detection

After building the symbol table, the analyzer checks for variables that are defined but never used:

```swift
// SemanticAnalyzer.swift:157-181
var usedVariables: Set<String> = []
for flow in dataFlows {
    usedVariables.formUnion(flow.inputs)
}

for (name, symbol) in symbolTable.symbols {
    if symbol.visibility == .published { continue }  // Used externally
    if case .alias = symbol.source { continue }       // Original tracked
    if symbol.visibility == .external { continue }    // Runtime-provided

    if !usedVariables.contains(name) {
        diagnostics.warning(
            "Variable '\(name)' is defined but never used",
            at: symbol.definedAt.start
        )
    }
}
```

This is a warning, not an error—unused variables don't break execution.

* * *

Circular Event Chain Detection

ARO's event system allows feature sets to emit events that trigger other feature sets. The analyzer detects circular chains:

```
FeatureSet A emits EventX
  → triggers Handler for EventX
    → which emits EventY
      → triggers Handler for EventY
        → which emits EventX  // CYCLE!
```

```swift
// SemanticAnalyzer.swift - detectCircularEventChains
private func detectCircularEventChains(_ analyzedSets: [AnalyzedFeatureSet]) {
    // Build event emission graph
    var emissionGraph: [String: Set<String>] = [:]

    for analyzed in analyzedSets {
        let emittedEvents = findEmittedEvents(in: analyzed.featureSet)
        // Map: event -> events it can transitively emit
    }

    // DFS to detect cycles
    for eventName in emissionGraph.keys {
        var visited: Set<String> = []
        var path: [String] = []
        if hasCycle(from: eventName, graph: emissionGraph, visited: &visited, path:
        &path) {
            diagnostics.error(
                "Circular event chain detected: \(path.joined(separator: " → "))",
                at: ...
            )
        }
    }
}
```

* * *

Orphaned Event Detection

Events that are emitted but never handled are flagged as warnings:

```swift
// SemanticAnalyzer.swift - detectOrphanedEventEmissions
private func detectOrphanedEventEmissions(_ analyzedSets: [AnalyzedFeatureSet]) {
    var emittedEvents: Set<String> = []
    var handledEvents: Set<String> = []

    for analyzed in analyzedSets {
        emittedEvents.formUnion(findEmittedEvents(in: analyzed.featureSet))

        // Check if this feature set handles events
        if analyzed.featureSet.businessActivity.contains("Handler") {
            let eventName = extractEventName(from:
analyzed.featureSet.businessActivity)
            handledEvents.insert(eventName)
        }
    }

    for event in emittedEvents {
        if !handledEvents.contains(event) {
            diagnostics.warning(
                "Event '\(event)' is emitted but no handler is registered",
                at: ...
            )
        }
    }
}
```

* * *

Multi-Pass Architecture

The four passes serve specific purposes:

```
Pass 1: Build Symbol Tables
  - Create symbols for each statement
  - Track defined variables
  - Build data flow info
  - Register published symbols in global registry

Pass 2: Verify Dependencies
  - Check that required variables exist
  - Validate cross-feature-set references
  - Enforce business activity boundaries

Pass 3: Detect Circular Events
  - Build event emission graph
  - DFS for cycles
  - Report circular chains

Pass 4: Detect Orphans
  - Collect all emitted events
  - Collect all handled events
  - Warn about orphans
```

Why multiple passes? Some checks require information from all feature sets (circular events, orphan events). Single-pass analysis would miss these.

* * *

Diagnostic Collection

Rather than failing on the first error, the analyzer collects all diagnostics:

```swift
public final class DiagnosticCollector: @unchecked Sendable {
    private var diagnostics: [Diagnostic] = []

    public func error(_ message: String, at location: SourceLocation) {
        diagnostics.append(Diagnostic(severity: .error, message: message, location:
        location))
    }

    public func warning(_ message: String, at location: SourceLocation) {
        diagnostics.append(Diagnostic(severity: .warning, message: message,
        location: location))
    }

    public var hasErrors: Bool {
        diagnostics.contains { $0.severity == .error }
    }
}
```

This enables reporting all problems in a single compilation:

```
Warning: Variable 'temp' is defined but never used
  at line 3, column 13

Error: Cannot rebind variable 'user' - variables are immutable
  at line 5, column 13

Error: Circular event chain detected: UserCreated → NotificationSent → UserCreated
  at line 12, column 4
```

* * *

Chapter Summary

ARO's semantic analysis enforces the language's design principles:

1. **Symbol tables per feature set**: Each scope is isolated; sharing requires explicit Publish.
2. **Business activity boundaries**: Published symbols are scoped to their business activity, not globally visible.
3. **Data flow tracking**: Inputs, outputs, and side effects are computed for each statement.
4. **Immutability enforcement**: Variables cannot be rebound except with special verbs or `_` prefix.
5. **Event chain validation**: Circular event chains and orphaned events are detected.

The analyzer is designed for reporting, not aborting. Multiple errors can be collected and shown to the user at once.

Implementation reference: `Sources/AROParser/SemanticAnalyzer.swift`

* * *

Next: Chapter 6 — Interpreted Execution

§ Chapter 6: Interpreted Execution

Execution Engine Architecture

The runtime engine orchestrates program execution. It manages the event bus, registers feature sets, and dispatches events to handlers.

```swift
// ExecutionEngine.swift
public final class ExecutionEngine {
    private let eventBus: EventBus
    private let actionRegistry: ActionRegistry
    private var featureSetExecutors: [String: FeatureSetExecutor]
}
```

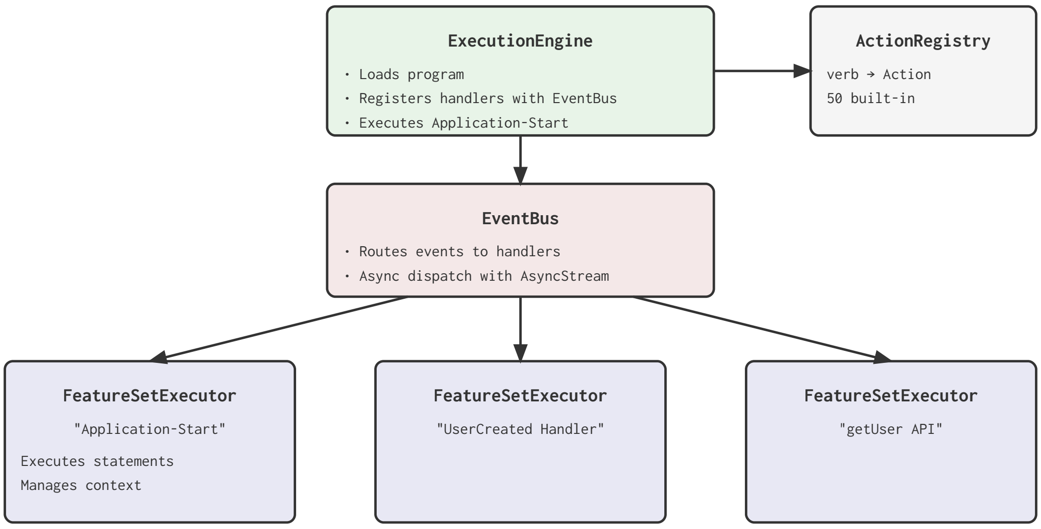


Figure 1

Figure 6.1: Execution engine architecture. The engine coordinates between EventBus and FeatureSetExecutors.

* * *

ExecutionContext Protocol

Actions access runtime services through the context:

```
public protocol ExecutionContext: AnyObject, Sendable {
    // Variable management
    func resolve<T: Sendable>(_ name: String) -> T?
    func require<T: Sendable>(_ name: String) throws -> T
    func bind(_ name: String, value: any Sendable)
    func exists(_ name: String) -> Bool

    // Service access
    func service<S>(_ type: S.Type) -> S?
    func register<S: Sendable>(_ service: S)

    // Repository access
    func repository<T: Sendable>(named: String) -> (any Repository<T>)?

    // Response management
    func setResponse(_ response: Response)
    func getResponse() -> Response?

    // Event emission
    func emit(_ event: any RuntimeEvent)

    // Metadata
    var featureSetName: String { get }
    var businessActivity: String { get }
    var executionId: String { get }
}
```

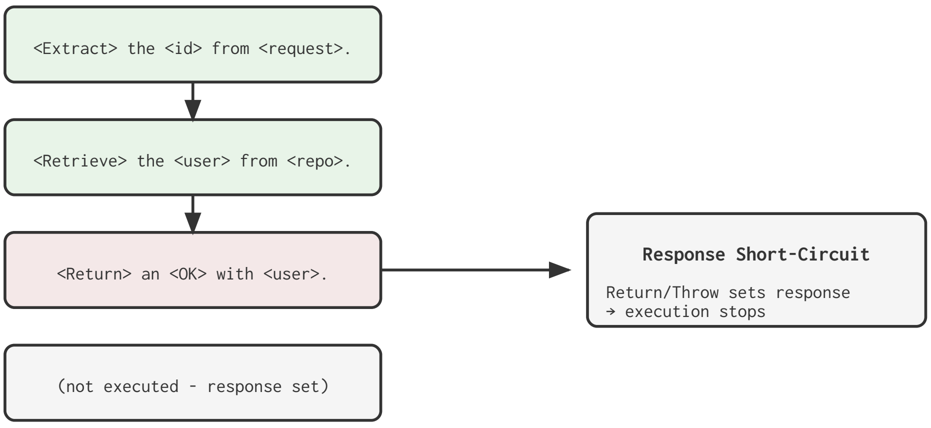
The context is the action's view of the world—it can read variables, bind new ones, access services, and emit events.

* * *

FeatureSetExecutor

Each feature set gets an executor that processes statements sequentially:

```swift
public final class FeatureSetExecutor {
    private let featureSet: FeatureSet
    private let actionRegistry: ActionRegistry
    private var context: RuntimeContext

    public func execute() async throws {
        for statement in featureSet.statements {
            try await executeStatement(statement)

            // Check for response short-circuit
            if context.getResponse() != nil {
                break
            }
        }
    }
}
```



<Extract> the <id> from <request>.
<Retrieve> the <user> from <repo>.
<Return> an <OK> with <user>.
(not executed - response set)
Response Short-Circuit
Return/Throw sets response
→ execution stops

Figure 2

Figure 6.2: Statement execution sequence. Return or Throw sets a response, causing remaining statements to be skipped.

* * *

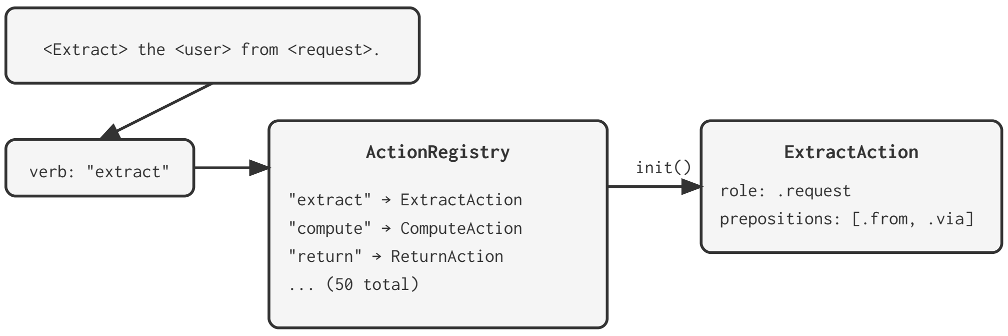
ActionRegistry Design

Actions are registered by their verbs and looked up at execution time:

```swift
public final class ActionRegistry: @unchecked Sendable {
    private var actions: [String: any ActionImplementation.Type] = [:]

    public func register<A: ActionImplementation>(_ action: A.Type) {
        for verb in A.verbs {
            actions[verb.lowercased()] = action
        }
    }

    public func action(for verb: String) -> (any ActionImplementation)? {
        guard let actionType = actions[verb.lowercased()] else { return nil }
        return actionType.init()  // Stateless instantiation
    }
}
```

ARO has 50 built-in actions. Each is registered at startup:

```swift
// ActionRegistry initialization
ActionRegistry.shared.register(ExtractAction.self)
ActionRegistry.shared.register(RetrieveAction.self)
ActionRegistry.shared.register(ComputeAction.self)
ActionRegistry.shared.register(ReturnAction.self)
// ... 46 more
```



<Extract> the <user> from <request>.
verb: "extract"
ActionRegistry
"extract" → ExtractAction
"compute" → ComputeAction
"return" → ReturnAction
... (50 total)
init()
ExtractAction
role: .request
prepositions: [.from, .via]

Figure 3

Figure 6.3: Action dispatch sequence. The verb is looked up in the registry, and a fresh action instance is created.

* * *

Descriptor-Based Invocation

Actions receive structured information via descriptors:

```
public struct ResultDescriptor: Sendable {
    public let base: String         // Variable to bind
    public let specifiers: [String] // Qualifiers
    public let span: SourceSpan
}

public struct ObjectDescriptor: Sendable {
    public let preposition: Preposition
    public let base: String
    public let specifiers: [String]
    public let keyPath: String      // "request.parameters.userId"
}
```

The executor builds these from the AST:

```
let resultDesc = ResultDescriptor(
    base: statement.result.base,
    specifiers: statement.result.specifiers,
    span: statement.result.span
)

let objectDesc = ObjectDescriptor(
    preposition: statement.object.preposition,
    base: statement.object.noun.base,
    specifiers: statement.object.noun.specifiers,
    ...
)
```

* * *

Context Hierarchy

For loops, child contexts are created:

```
for each <item> in <items> {
    // child context created per iteration
    // item is bound fresh each time
}
```

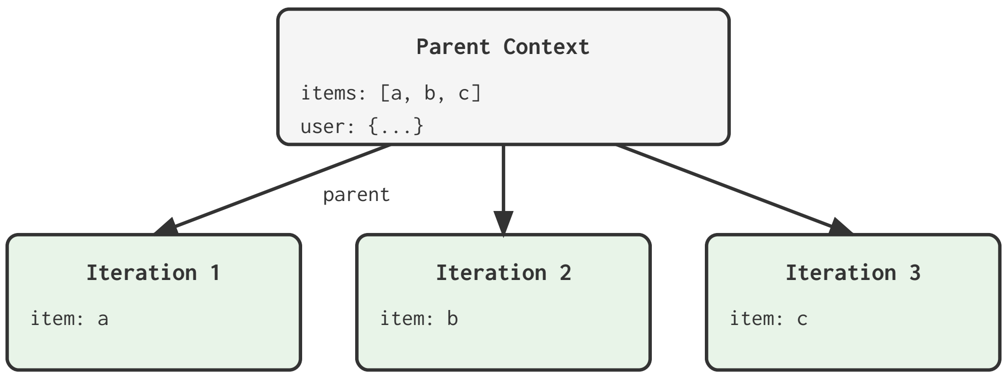


Figure 4

Figure 6.4: Context tree. Child contexts inherit from parent but have their own bindings for loop variables.

* * *

Chapter Summary

The interpreted execution model is straightforward:

1. **ExecutionEngine** loads the program and registers feature sets with EventBus
2. **EventBus** routes events to matching handlers
3. **FeatureSetExecutor** processes statements sequentially
4. **ActionRegistry** maps verbs to action implementations
5. **Descriptors** carry structured information to actions
6. **Context hierarchy** enables scoped variable binding for loops

The interpreter is the reference implementation. Native compilation (Chapter 8) generates code that calls the same action implementations through a C bridge.

Implementation references: `- Sources/ARORuntime/Core/ExecutionEngine.swift - Sources/ARORuntime/Core/FeatureSetExecutor.swift - Sources/ARORuntime/Actions/ActionRegistry.swift`

* * *

Next: Chapter 7 — Event Architecture

§ Chapter 7: Event Architecture

The EventBus Pattern

ARO uses a centralized publish-subscribe pattern for inter-component communication. The EventBus decouples event producers from consumers, enabling reactive programming without tight coupling.

```swift
// EventBus.swift
public final class EventBus: @unchecked Sendable {
    public typealias EventHandler = @Sendable (any RuntimeEvent) async -> Void

    private struct Subscription: Sendable {
        let id: UUID
        let eventType: String
        let handler: EventHandler
    }

    private var subscriptions: [Subscription] = []
    private var continuations: [UUID: AsyncStream<any RuntimeEvent>.Continuation] =
        [:]
}
```

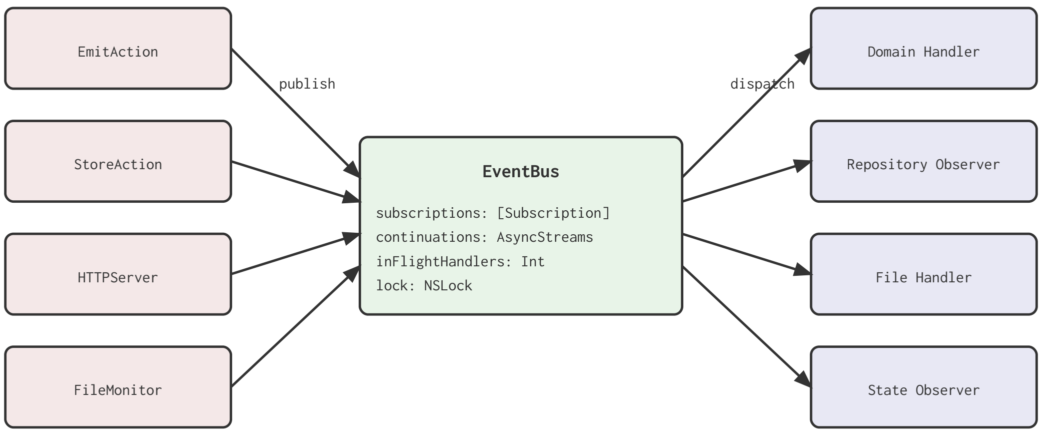


Figure 1

Figure 7.1: EventBus architecture. Producers publish events; the bus dispatches to matching subscribers.

* * *

RuntimeEvent Protocol

All events conform to a simple protocol:

```swift
public protocol RuntimeEvent: Sendable {
    static var eventType: String { get }
    var timestamp: Date { get }
}
```

The `eventType` is the routing key. Subscribers register for specific types or use `"*"` to receive all events.

ARO defines several event categories:

Category	Event Types	Source
Application	`application.started` , `application.stopping`	Lifecycle
HTTP	`http.request` , `http.response`	HTTPServer
File	`file.created` , `file.modified` , `file.deleted`	FileMonitor
Socket	`socket.connected` , `socket.data` , `socket.disconnected`	SocketServer
Repository	`repository.changed`	Store/Update/Delete actions
State	`state.transition`	Accept action
Domain	`domain` (with subtypes)	Emit action
Error	`error`	Any component

* * *

Handler Registration Timing

A critical design decision: **handlers are registered before the entry point executes**.

```swift
public func execute(_ program: AnalyzedProgram, entryPoint: String) async throws ->
        Response {
    // 1. Find entry point feature set
    guard let entryFeatureSet = program.featureSets.first(where: {
        $0.featureSet.name == entryPoint
    }) else {
        throw ActionError.entryPointNotFound(entryPoint)
    }

    // 2. Register ALL handlers BEFORE execution
    registerSocketEventHandlers(for: program, baseContext: context)
    registerDomainEventHandlers(for: program, baseContext: context)
    registerFileEventHandlers(for: program, baseContext: context)
    registerRepositoryObservers(for: program, baseContext: context)
    registerStateObservers(for: program, baseContext: context)

    // 3. Now execute entry point
    let response = try await executor.execute(entryFeatureSet, context: context)

    return response
}
```

This ordering ensures that events emitted during `Application-Start` have handlers ready to receive them.

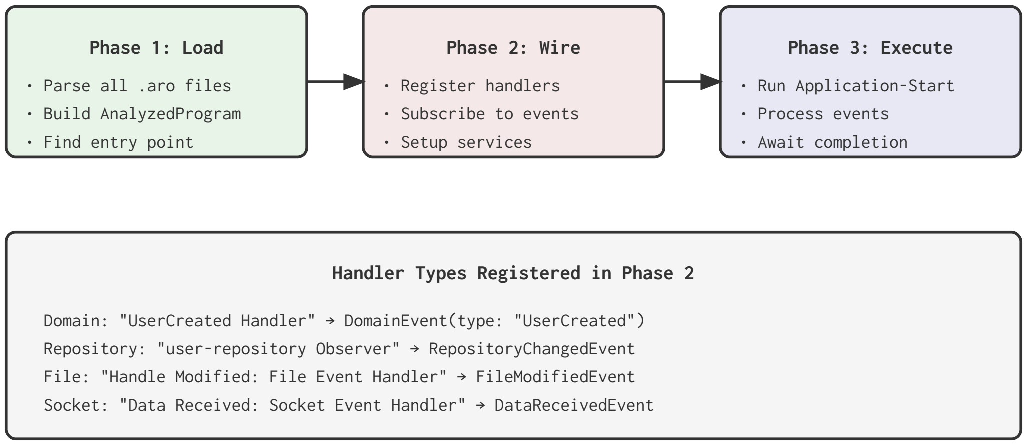

Phase 1: Load
• Parse all .aro files
• Build AnalyzedProgram
• Find entry point
Phase 2: Wire
• Register handlers
• Subscribe to events
• Setup services
Phase 3: Execute
• Run Application-Start
• Process events
• Await completion
Handler Types Registered in Phase 2
Domain: "UserCreated Handler" → DomainEvent(type: "UserCreated")
Repository: "user-repository Observer" → RepositoryChangedEvent
File: "Handle Modified: File Event Handler" → FileModifiedEvent
Socket: "Data Received: Socket Event Handler" → DataReceivedEvent

Figure 2

Figure 7.2: Execution phases. Handlers must be wired before the entry point runs.

* * *

Five Handler Types

ARO supports five categories of event handlers, distinguished by business activity patterns:

1. Domain Event Handlers

Pattern: `{EventName} Handler`

```
(Send Welcome Email: UserCreated Handler) {
    <Extract> the <user> from the <event: user>.
    <Send> the <welcome-email> to the <user: email>.
    <Return> an <OK: status> for the <notification>.
}
```

Registration logic:

```
private func registerDomainEventHandlers(for program: AnalyzedProgram, ...) {
    let domainHandlers = program.featureSets.filter { analyzedFS in
        let activity = analyzedFS.featureSet.businessActivity
        let hasHandler = activity.contains(" Handler")
        let isSpecialHandler = activity.contains("Socket Event Handler") ||
                               activity.contains("File Event Handler")
        return hasHandler && !isSpecialHandler
    }

    for analyzedFS in domainHandlers {
        // Extract "UserCreated" from "UserCreated Handler"
        let eventType = extractEventType(from: activity)

        eventBus.subscribe(to: DomainEvent.self) { event in
            guard event.domainEventType == eventType else { return }
            await self.executeDomainEventHandler(analyzedFS, event: event)
        }
    }
}
```

2. Repository Observers

Pattern: `{repository-name} Observer`

```
(Log User Changes: user-repository Observer) {
    <Extract> the <changeType> from the <event: changeType>.
    <Log> "User ${<changeType>}" to the <console>.
    <Return> an <OK: status> for the <logging>.
}
```

Repository observers are triggered by Store, Update, and Delete actions:

```
// Inside StoreAction.execute()
eventBus.publishAndTrack(RepositoryChangedEvent(
    repositoryName: repositoryName,
    changeType: .created,
    entityId: entityId,
    newValue: entity
))
```

3. File Event Handlers

Pattern: `{description}: File Event Handler`

```
(Handle Modified: File Event Handler) {
    <Extract> the <path> from the <event: path>.
    <Log> "File changed: ${<path>}" to the <console>.
    <Return> an <OK: status> for the <notification>.
}
```

The feature set name determines which file event type is handled: - Contains “created” → `FileCreatedEvent` - Contains “modified” → `FileModifiedEvent` - Contains “deleted” → `FileDeletedEvent`

4. Socket Event Handlers

Pattern: `{description}: Socket Event Handler`

```
(Data Received: Socket Event Handler) {
    <Extract> the <data> from the <packet: data>.
    <Send> the <data> to the <packet: connectionId>.
    <Return> an <OK: status> for the <echo>.
}
```

5. State Observers

Pattern: `{fieldName} StateObserver` or `{fieldName} StateObserver<from_to_to>`

```
(* Observe all transitions on 'status' field *)
(Audit Changes: status StateObserver) {
    <Extract> the <fromState> from the <transition: fromState>.
    <Extract> the <toState> from the <transition: toState>.
    <Log> "Status: ${<fromState>} → ${<toState>}" to the <console>.
    <Return> an <OK: status> for the <audit>.
}

(* Observe specific transition: draft → placed *)
(Notify Placed: status StateObserver<draft_to_placed>) {
    <Extract> the <order> from the <transition: entity>.
    <Send> the <order-confirmation> to the <order: customerEmail>.
    <Return> an <OK: status> for the <notification>.
}
```

* * *

State Guards

Handlers can filter events based on entity field values using angle bracket syntax:

```
(* Only handle when status = "paid" *)
(Process Paid Orders: OrderCreated Handler<status:paid>) {
    ...
}

(* Multiple values with OR logic *)
(Handle Active Users: UserUpdated Handler<status:active,premium>) {
    ...
}

(* Multiple fields with AND logic *)
(VIP Processing: OrderCreated Handler<status:paid;tier:gold>) {
    ...
}
```

The `StateGuard` and `StateGuardSet` types parse and evaluate these conditions:

```swift
public struct StateGuard: Sendable {
    public let fieldPath: String      // e.g., "status" or "entity.status"
    public let validValues: Set<String>  // OR logic within guard

    public func matches(payload: [String: any Sendable]) -> Bool {
        guard let fieldValue = resolveFieldPath(fieldPath, in: payload) else {
            return false
        }
        return validValues.contains(fieldValue.lowercased())
    }
}

public struct StateGuardSet: Sendable {
    public let guards: [StateGuard]  // AND logic between guards

    public func allMatch(payload: [String: any Sendable]) -> Bool {
        guards.allSatisfy { $0.matches(payload: payload) }
    }
}
```

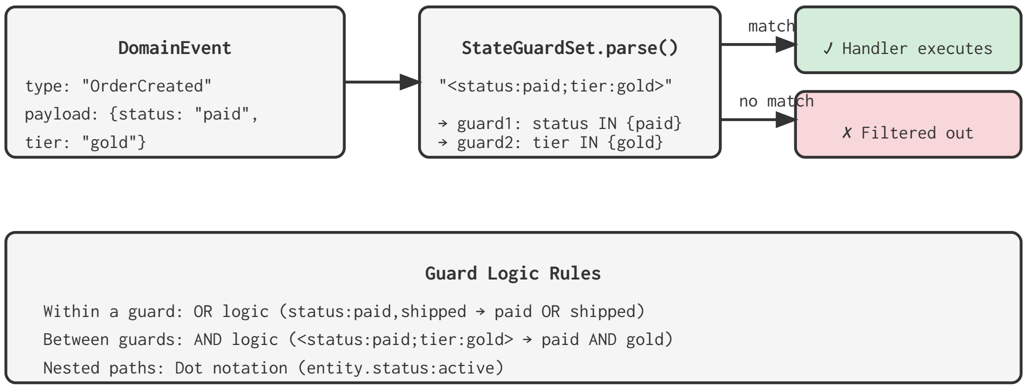

DomainEvent
type: "OrderCreated"
payload: {status: "paid",
tier: "gold"}
StateGuardSet.parse()
"<status:paid;tier:gold>"
→ guard1: status IN {paid}
→ guard2: tier IN {gold}
match
✓ Handler executes
no match
✗ Filtered out
Guard Logic Rules
Within a guard: OR logic (status:paid,shipped → paid OR shipped)
Between guards: AND logic (<status:paid;tier:gold> → paid AND gold)
Nested paths: Dot notation (entity.status:active)

Figure 3

Figure 7.3: State guard filtering. Guards provide declarative event filtering without code.

* * *

In-Flight Tracking

Events can trigger handlers that emit more events. The EventBus tracks in-flight handlers to ensure all cascaded processing completes:

```
public func publishAndTrack(_ event: any RuntimeEvent) async {
    let matchingSubscriptions = getMatchingSubscriptions(for: eventType)

    await withTaskGroup(of: Void.self) { group in
        for subscription in matchingSubscriptions {
            // Increment before spawning
            withLock { inFlightHandlers += 1 }

            group.addTask {
                await subscription.handler(event)

                // Decrement after completion, notify waiters if zero
                let continuationsToResume = self.withLock { () -> [...] in
                    self.inFlightHandlers -= 1
                    if self.inFlightHandlers == 0 {
                        let continuations = self.flushContinuations
                        self.flushContinuations.removeAll()
                        return continuations
                    }
                    return []
                }

                for continuation in continuationsToResume {
                    continuation.resume()
                }
            }
        }
    }
}
```

After `Application-Start` completes, the engine waits for all handlers:

```
// After entry point execution
let completed = await eventBus.awaitPendingEvents(timeout: 10.0)
if !completed {
    print("[WARNING] Event handlers did not complete within timeout")
}
```

* * *

Race Condition Prevention

The EventBus uses careful lock ordering to prevent race conditions:

```swift
public func awaitPendingEvents(timeout: TimeInterval) async -> Bool {
    return await withTaskGroup(of: Bool.self) { group in
        // Task 1: Wait for handlers
        group.addTask {
            await withCheckedContinuation { continuation in
                self.withLock {
                    // CRITICAL: Both check AND append inside same lock
                    if self.inFlightHandlers == 0 {
                        continuation.resume()  // Already done
                    } else {
                        self.flushContinuations.append(continuation)  // Wait
                    }
                }
            }
            return true
        }

        // Task 2: Timeout
        group.addTask {
            try? await Task.sleep(nanoseconds: UInt64(timeout * 1_000_000_000))
            return false
        }

        // First to complete wins
        if let result = await group.next() {
            group.cancelAll()
            return result
        }
        return false
    }
}
```

The key insight: checking `inFlightHandlers == 0` and appending to `flushContinuations` must happen atomically. Otherwise, a handler could complete between the check and append, leaving the continuation waiting forever.

* * *

AsyncStream Integration

The EventBus supports both callback-based and stream-based subscriptions:

```swift
// Callback-based
eventBus.subscribe(to: "http.request") { event in
    await handleRequest(event)
}

// Stream-based
let stream = eventBus.stream(for: HTTPRequestReceivedEvent.self)
for await event in stream {
    await handleRequest(event)
}
```

Stream subscriptions use AsyncStream continuations:

```swift
public func stream<E: RuntimeEvent>(for type: E.Type) -> AsyncStream<E> {
    AsyncStream { continuation in
        let id = subscribe(to: type) { event in
            continuation.yield(event)
        }

        continuation.onTermination = { [weak self] _ in
            self?.unsubscribe(id)
        }
    }
}
```

* * *

Handler Execution Pattern

When an event matches a handler, a child context is created:

```
private func executeDomainEventHandler(
    _ analyzedFS: AnalyzedFeatureSet,
    event: DomainEvent
) async {
    // Create isolated context for this handler
    let handlerContext = RuntimeContext(
        featureSetName: analyzedFS.featureSet.name,
        businessActivity: analyzedFS.featureSet.businessActivity,
        eventBus: eventBus,
        parent: baseContext  // Inherit services
    )

    // Bind event payload for extraction
    handlerContext.bind("event", value: event.payload)

    // Execute the handler's statements
    let executor = FeatureSetExecutor(...)
    do {
        _ = try await executor.execute(analyzedFS, context: handlerContext)
    } catch {
        // Publish error event (handlers are fault-isolated)
        eventBus.publish(ErrorOccurredEvent(
            error: String(describing: error),
            context: analyzedFS.featureSet.name,
            recoverable: true
        ))
    }
}
```

Note: Handler errors are **isolated**. A failing handler doesn't crash the application; it publishes an error event and continues.

* * *

Chapter Summary

The event architecture enables loosely coupled, reactive programming:

1. **EventBus** provides centralized publish-subscribe with type-based routing
2. **Handler Registration** happens before entry point execution
3. **Five Handler Types** cover domain events, repositories, files, sockets, and state transitions
4. **State Guards** enable declarative filtering without code
5. **In-Flight Tracking** ensures cascaded events complete before shutdown
6. **Race Prevention** uses careful lock ordering for correctness

7. **AsyncStream** integration supports modern Swift concurrency patterns

The event system is the glue between services. When an HTTP request arrives, the server publishes an event. When a file changes, the monitor publishes an event. Feature sets subscribe to these events and react—without knowing or caring about the source.

Implementation references: - `Sources/ARORuntime/Events/EventBus.swift` (287 lines) - `Sources/ARORuntime/Events/EventTypes.swift` (321 lines) - `Sources/ARORuntime/Events/StateGuard.swift` (129 lines) - `Sources/ARORuntime/Core/ExecutionEngine.swift` (851 lines)

* * *

Next: Chapter 8 — Native Compilation

§ Chapter 8: Native Compilation

Why Native Compilation?

The interpreter works well for development and debugging, but production deployments benefit from: - **Startup time**: No parsing or compilation at runtime - **Single binary**: Self-contained executable, no runtime dependencies - **Performance**: Direct machine code execution

ARO generates LLVM IR text, compiles it to object code via `llc`, and links with the Swift runtime.

* * *

Compilation Pipeline

```
Source Files (.aro)
       ↓
    Parser
       ↓
AnalyzedProgram (AST)
       ↓
 LLVMCodeGenerator
       ↓
  LLVM IR (.ll)
       ↓
    llc
       ↓
Object File (.o)
       ↓
   Linker
       ↓
Executable
```

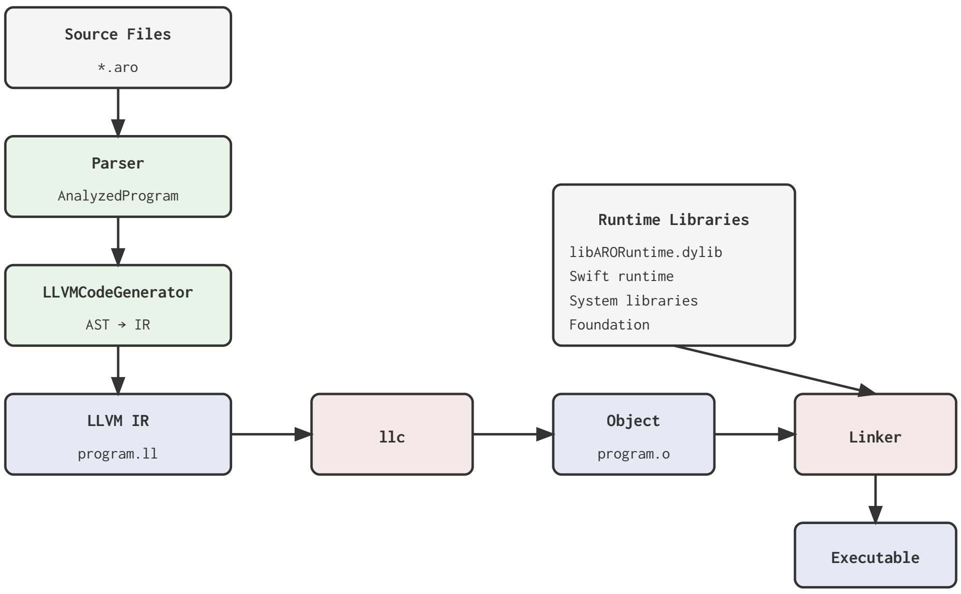


Figure 1

Figure 8.1: Compilation pipeline. ARO generates LLVM IR, `llc` emits object code, the linker produces the final executable.

* * *

Why Textual LLVM IR?

ARO generates LLVM IR as text rather than using the LLVM C API. This decision has trade-offs:

Advantages: - No C++ dependency—easier to build and distribute - Debuggable—human-readable IR for inspection - Portable—same generator works across LLVM versions - Simpler—string concatenation vs. complex API calls

Disadvantages: - Slower generation—parsing text is slower than API calls - No compile-time type checking—IR errors found at `llc` time - Version sensitivity—textual format can change

The textual approach works well for a DSL compiler where IR generation is straightforward and debugging visibility matters more than raw speed.

* * *

Module Structure

Every generated module starts with a standard header:

```
; ModuleID = 'aro_program'
source_filename = "aro_program.ll"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx14.0.0"
```

The target triple is platform-specific: - **macOS ARM64**: `arm64-apple-macosx14.0.0` - **macOS x86_64**: `x86_64-apple-macosx14.0.0` - **Linux ARM64**: `aarch64-unknown-linux-gnu` - **Linux x86_64**: `x86_64-unknown-linux-gnu` - **Windows**: `x86_64-pc-windows-msvc`

* * *

Type Definitions

ARO uses two struct types to pass statement metadata to runtime actions:

```
; AROResultDescriptor: { base, specifiers, count }
%AROResultDescriptor = type { ptr, ptr, i32 }

; AROObjectDescriptor: { base, preposition, specifiers, count }
%AROObjectDescriptor = type { ptr, i32, ptr, i32 }
```

These mirror the Swift `ResultDescriptor` and `ObjectDescriptor` types. The runtime bridge converts between C structs and Swift types.

* * *

External Declarations

The generated code calls into the runtime through C-callable functions:

```
; Runtime lifecycle
declare ptr @aro_runtime_init()
declare void @aro_runtime_shutdown(ptr)
declare ptr @aro_context_create(ptr)
declare void @aro_context_destroy(ptr)

; Variable operations
declare void @aro_variable_bind_string(ptr, ptr, ptr)
declare void @aro_variable_bind_int(ptr, ptr, i64)
declare ptr @aro_variable_resolve(ptr, ptr)

; Actions
declare ptr @aro_action_extract(ptr, ptr, ptr)
declare ptr @aro_action_compute(ptr, ptr, ptr)
declare ptr @aro_action_return(ptr, ptr, ptr)
; ... 47 more action declarations
```

The generator emits declarations for all 50 built-in actions, plus runtime lifecycle and variable operations.

* * *

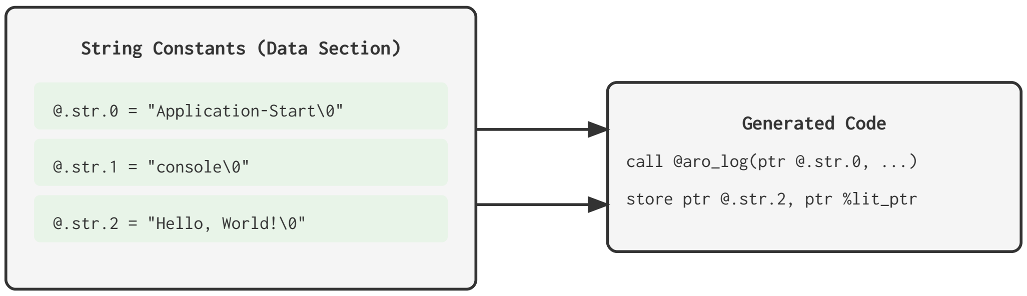
String Constant Collection

Before generating code, the generator collects all string constants used in the program:

```
private func collectStringConstants(_ program: AnalyzedProgram) {
    for featureSet in program.featureSets {
        registerString(featureSet.featureSet.name)
        registerString(featureSet.featureSet.businessActivity)

        for statement in featureSet.featureSet.statements {
            collectStringsFromStatement(statement)
        }
    }

    // Always register internal marker strings
    registerString("_literal_")
    registerString("_expression_")
}
```

Each unique string gets a global constant:

```
@.str.0 = private unnamed_addr constant [18 x i8] c"Application-Start\00"
@.str.1 = private unnamed_addr constant [8 x i8] c"console\00"
@.str.2 = private unnamed_addr constant [14 x i8] c"Hello, World!\00"
```



String Constants (Data Section)
@.str.0 = "Application-Start\0"
@.str.1 = "console\0"
@.str.2 = "Hello, World!\0"
Generated Code
call @aro_log(ptr @.str.0, ...)
store ptr @.str.2, ptr %lit_ptr

Figure 2

Figure 8.2: String constants are collected first, then referenced by pointer in generated code.

* * *

Feature Set Generation

Each feature set becomes an LLVM function:

```
private func generateFeatureSet(_ featureSet: AnalyzedFeatureSet) throws {
    let funcName = mangleFeatureSetName(featureSet.featureSet.name)

    emit("define ptr @\(funcName)(ptr %ctx) {")
    emit("entry:")
    emit("  %__result = alloca ptr")
    emit("  store ptr null, ptr %__result")

    for (index, statement) in featureSet.featureSet.statements.enumerated() {
        try generateStatement(statement, index: index)
    }

    emit("  %final_result = load ptr, ptr %__result")
    emit("  ret ptr %final_result")
    emit("}")
}
```

The function takes a context pointer and returns a result pointer. Name mangling converts `"Application-Start"` to `aro_fs_application_start`.

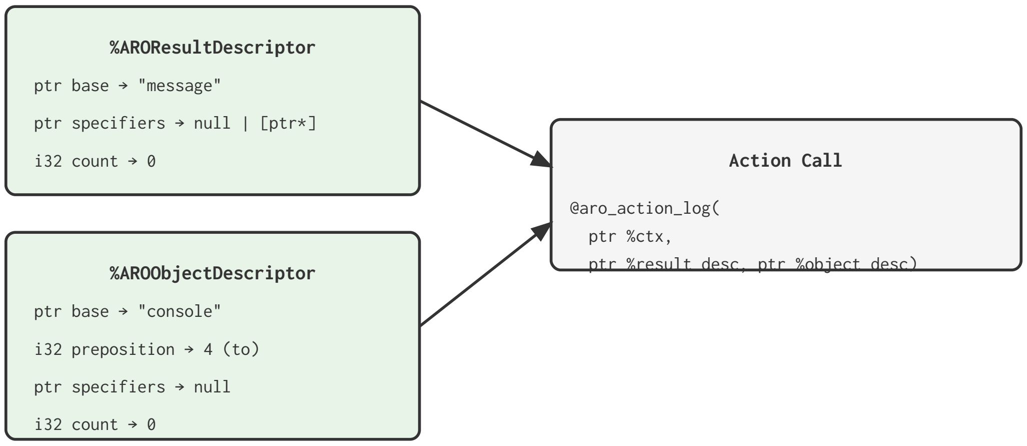
* * *

Statement Generation

For each ARO statement, the generator:

1. **Binds literal values** to special variables
2. **Allocates descriptors** on the stack
3. **Fills descriptor fields** with string pointers and counts
4. **Calls the action function**

```
; <Log> "Hello, World!" to the <console>.
s0:
  ; Bind literal to _literal_
  call void @aro_variable_bind_string(ptr %ctx, ptr @.str._literal_, ptr
        @.str.hello)

  ; Allocate result descriptor
  %s0_result_desc = alloca %AROResultDescriptor
  %s0_rd_base_ptr = getelementptr inbounds %AROResultDescriptor, ptr
        %s0_result_desc, i32 0, i32 0
  store ptr @.str.message, ptr %s0_rd_base_ptr
  ; ... fill specifiers and count

  ; Allocate object descriptor
  %s0_object_desc = alloca %AROObjectDescriptor
  ; ... fill base, preposition, specifiers, count

  ; Call action
  %s0_action_result = call ptr @aro_action_log(ptr %ctx, ptr %s0_result_desc, ptr
        %s0_object_desc)
  store ptr %s0_action_result, ptr %__result
```



```
@aro_action_log(
  ptr %ctx,
  ptr %result_desc, ptr %object_desc)
```

Figure 3

Figure 8.3: Descriptor struct layout. Base names are string pointers, specifiers are arrays of pointers.

* * *

Control Flow: When Guards

When guards generate conditional branches:

```
// ARO-0004: Handle when clause
if let whenCondition = statement.whenCondition {
    let whenJSON = expressionToEvalJSON(whenCondition)
    emit("  %when_result = call i32 @aro_evaluate_when_guard(ptr %ctx, ptr
        @.str.when)")
    emit("  %when_pass = icmp ne i32 %when_result, 0")
    emit("  br i1 %when_pass, label %s0_body, label %s0_skip")
    emit("s0_body:")
}
```

The guard expression is serialized to JSON and evaluated at runtime.

* * *

Control Flow: Match Expressions

Match statements generate a chain of comparisons:

```
; match <status>
m0:
  %m0_subject_val = call ptr @aro_variable_resolve(ptr %ctx, ptr @.str.status)
  %m0_subject_str = call ptr @aro_value_as_string(ptr %m0_subject_val)
  br label %m0_case0_check

m0_case0_check:
  %m0_case0_cmp = call i32 @strcmp(ptr %m0_subject_str, ptr @.str.active)
  %m0_case0_match = icmp eq i32 %m0_case0_cmp, 0
  br i1 %m0_case0_match, label %m0_case0_body, label %m0_case1_check

m0_case0_body:
  ; ... statements for "active" case
  br label %m0_end

m0_case1_check:
  ; ... check next pattern
```

* * *

Control Flow: For-Each Loops

Sequential loops create iteration contexts:

```
fe0_header:
  %fe0_i = phi i64 [ 0, %fe0_init ], [ %fe0_next_i, %fe0_continue ]

  ; Create child context for this iteration (avoids immutability violation)
  %fe0_iter_ctx = call ptr @aro_context_create_child(ptr %ctx, ptr null)

  ; Get element and bind to item variable
  %fe0_element = call ptr @aro_array_get(ptr %fe0_collection, i64 %fe0_i)
  call void @aro_variable_bind_value(ptr %fe0_iter_ctx, ptr @.str.item, ptr
        %fe0_element)

  ; Execute body statements (using child context)
  ; ...

  ; Cleanup
  call void @aro_context_destroy(ptr %fe0_iter_ctx)
  br label %fe0_continue

fe0_continue:
  %fe0_next_i = add i64 %fe0_i, 1
  %fe0_done = icmp sge i64 %fe0_next_i, %fe0_count
  br i1 %fe0_done, label %fe0_end, label %fe0_header
```

* * *

Parallel Loops

Parallel for-each loops require a separate function for the body:

```
private func generateParallelForEachLoop(_ loop: ForEachLoop, ...) throws {
    // Generate unique loop body function name
    let bodyFuncName = "aro_loop_body_\(loopBodyCounter)"
    loopBodyCounter += 1

    // Call parallel executor
    emit("  %result = call i32 @aro_parallel_for_each_execute(")
    emit("      ptr %runtime, ptr %ctx, ptr %collection,")
    emit("      ptr @\(bodyFuncName), i64 \(concurrency), ...)")

    // Store for later generation
    pendingLoopBodies.append((bodyFuncName, loop, prefix))
}
```

The body function is generated after the feature set:

```
define ptr @aro_loop_body_0(ptr %loop_ctx, ptr %loop_item, i64 %loop_index) {
entry:
  call void @aro_variable_bind_value(ptr %loop_ctx, ptr @.str.item, ptr %loop_item)
  ; ... body statements using %loop_ctx
  ret ptr null
}
```

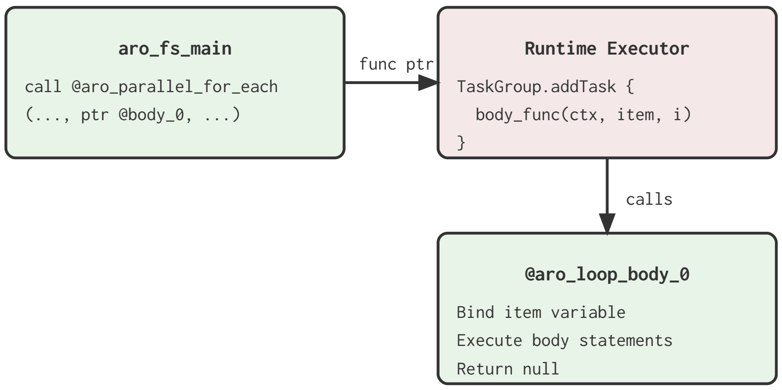


Figure 4

Figure 8.4: Parallel loop code structure. The feature set passes a function pointer to the runtime executor.

* * *

Main Function Generation

The `main` function orchestrates initialization, handler registration, and execution:

```
define i32 @main(i32 %argc, ptr %argv) {
entry:
  ; Initialize runtime
  %runtime = call ptr @aro_runtime_init()
  store ptr %runtime, ptr @global_runtime

  ; Set embedded OpenAPI spec (if available)
  call void @aro_set_embedded_openapi(ptr @.str.openapi_json)

  ; Create context
  %ctx = call ptr @aro_context_create_named(ptr %runtime, ptr @.str.app_start)

  ; Register event handlers BEFORE execution
  call void @aro_runtime_register_handler(ptr %runtime,
      ptr @.str.UserCreated, ptr @aro_fs_send_welcome_email)

  ; Execute Application-Start
  %result = call ptr @aro_fs_application_start(ptr %ctx)

  ; Wait for pending event handlers
  %await_result = call i32 @aro_runtime_await_pending_events(ptr %runtime, double
        10.0)

  ; Cleanup and exit
  call void @aro_context_print_response(ptr %ctx)
  call void @aro_context_destroy(ptr %ctx)
  call void @aro_runtime_shutdown(ptr %runtime)
  ret i32 0
}
```

* * *

Linking Process

After `llc` produces the object file, the linker combines it with:

1. **ARO Runtime Library** (`libARORuntime.dylib`)
2. **Swift Runtime** (`libswiftCore.dylib` , etc.)
3. **Foundation** and other system frameworks

4. **Platform libraries** (libc, libSystem)

```
public func link(objectFiles: [String], outputPath: String, ...) throws {
    var args = [findCompiler()]  // clang
    args.append(contentsOf: objectFiles)
    args.append("-o")
    args.append(outputPath)

    // Link ARO runtime
    if let runtimePath = runtimeLibraryPath {
        args.append("-L\(runtimePath)")
        args.append("-lARORuntime")
    }

    // Platform-specific flags
    #if os(macOS)
    args.append("-Xlinker")
    args.append("-rpath")
    args.append("-Xlinker")
    args.append("@executable_path/../lib")
    #endif

    try runProcess(args)
}
```

* * *

Platform-Specific Considerations

macOS

- Uses `@rpath` for library resolution
- Requires `swiftrt.o` for Swift runtime initialization
- Code signing may be needed for distribution

Linux

- Requires `libswiftCore.so` and related libraries
- May need `-Wl,-rpath` for runtime library paths
- Uses `ld` or `lld` as the linker

Windows

- Uses `clang` for IR compilation (no `llc` in standard LLVM)
- Links with MSVC runtime
- Different library naming conventions (`.dll` vs `.dylib`)

* * *

Optimization Levels

The `llc` tool accepts optimization flags:

```
public enum OptimizationLevel: String {
    case none = "-O0"
    case o1 = "-O1"
    case o2 = "-O2"
    case o3 = "-O3"
}
```

Size optimization (`-Os` , `-Oz`) is handled during linking:

```
if options.optimizeForSize {
    #if os(macOS)
    args.append("-Wl,-dead_strip")
    #else
    args.append("-Wl,--gc-sections")
    #endif
}

if options.strip {
    args.append("-s")  // Strip symbols
}
```

* * *

Chapter Summary

Native compilation transforms ARO programs into standalone executables:

1. **LLVMCodeGenerator** traverses the AST, emitting textual LLVM IR
2. **String constants** are collected first, then referenced by pointer

3. **Feature sets** become functions; **statements** become descriptor allocations and action calls
4. **Control flow** (when, match, for-each) uses LLVM branches and phi nodes
5. **Parallel loops** extract the body into a separate function
6. **Main function** initializes runtime, registers handlers, executes entry point
7. **Linking** combines object code with Swift runtime and ARO library

The textual IR approach trades some performance for simplicity and debuggability—a reasonable choice for a DSL compiler where generation speed is not critical.

Implementation references: - `Sources/AROCompiler/LLVMCodeGenerator.swift` (1895 lines) - `Sources/AROCompiler/Linker.swift` (1374 lines)

* * *

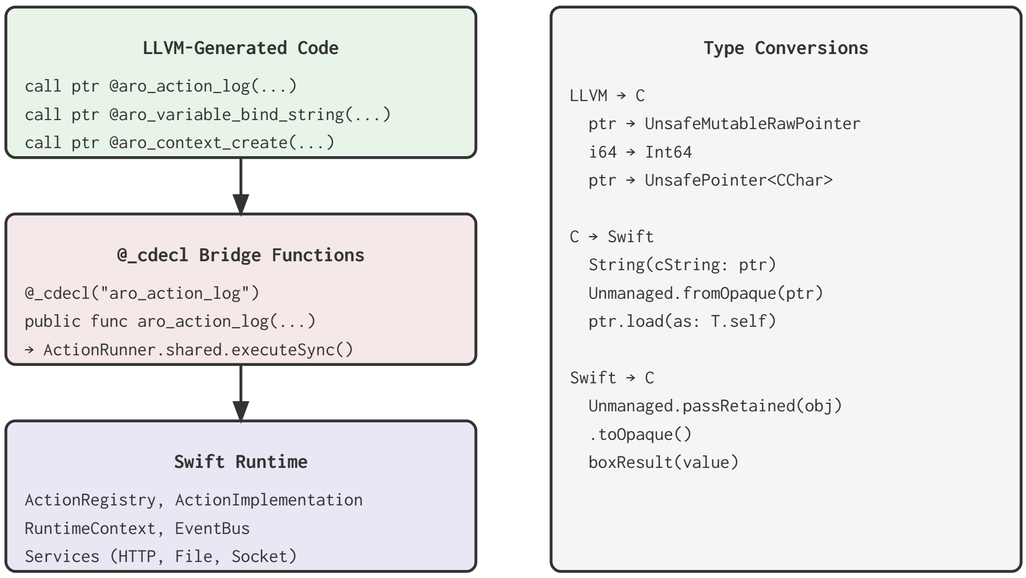
Next: Chapter 9 — Runtime Bridge

§ Chapter 9: Runtime Bridge

The Three-Layer Architecture

Compiled ARO binaries need to call Swift runtime code. LLVM generates native code that can call C functions. Swift can expose functions to C via `@_cdecl`. This creates a three-layer bridge:

```
LLVM-Generated Code (native)
        ↓ C calling convention
  @_cdecl Functions (Swift)
        ↓ Swift method calls
   Swift Runtime (Swift)
```



LLVM-Generated Code
call ptr @aro_action_log(...)
call ptr @aro_variable_bind_string(...)
call ptr @aro_context_create(...)
@_cdecl Bridge Functions
@_cdecl("aro_action_log")
public func aro_action_log(...)
→ ActionRunner.shared.executeSync()
Swift Runtime
ActionRegistry, ActionImplementation
RuntimeContext, EventBus
Services (HTTP, File, Socket)
Type Conversions
LLVM → C
ptr → UnsafeMutableRawPointer
i64 → Int64
ptr → UnsafePointer<CChar>
C → Swift
String(cString: ptr)
Unmanaged.fromOpaque(ptr)
ptr.load(as: T.self)
Swift → C
Unmanaged.passRetained(obj)
.toOpaque()
boxResult(value)

Figure 1

Figure 9.1: Three-layer bridge architecture. LLVM code calls C functions, which call Swift methods.

* * *

Handle Management

The bridge uses opaque pointers to pass Swift objects to/from C code:

```swift
/// Opaque runtime handle for C interop
final class AROCRuntimeHandle: @unchecked Sendable {
    let runtime: Runtime
    var contexts: [UnsafeMutableRawPointer: AROCContextHandle] = [:]
}

/// Opaque context handle for C interop
class AROCContextHandle {
    let context: RuntimeContext
    let runtime: AROCRuntimeHandle
}
```

Creating a handle:

```swift
@_cdecl("aro_runtime_init")
public func aro_runtime_init() -> UnsafeMutableRawPointer? {
    let handle = AROCRuntimeHandle()

    // Convert Swift object to opaque pointer
    let pointer = Unmanaged.passRetained(handle).toOpaque()

    // Store in global registry (prevents deallocation)
    handleLock.lock()
    runtimeHandles[pointer] = handle
    handleLock.unlock()

    return UnsafeMutableRawPointer(pointer)
}
```

Retrieving a handle:

```swift
@_cdecl("aro_runtime_shutdown")
public func aro_runtime_shutdown(_ runtimePtr: UnsafeMutableRawPointer?) {
    guard let ptr = runtimePtr else { return }

    // Convert opaque pointer back to Swift object
    let handle = Unmanaged<AROCRuntimeHandle>.fromOpaque(ptr).takeUnretainedValue()

    // Clean up
    handleLock.lock()
    runtimeHandles.removeValue(forKey: ptr)
    handleLock.unlock()

    // Release the retained object
    Unmanaged<AROCRuntimeHandle>.fromOpaque(ptr).release()
}
```

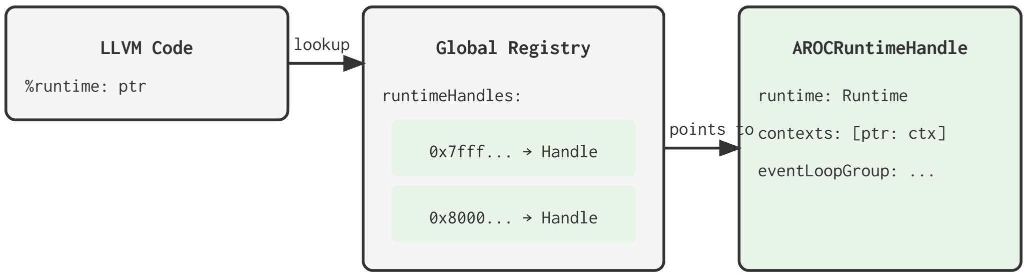


Figure 2

Figure 9.2: Handle lifecycle. Opaque pointers index into a global registry that holds Swift objects.

* * *

Descriptor Conversion

C structs are read with manual offset calculations:

```swift
/// Convert C object descriptor to Swift ObjectDescriptor
func toObjectDescriptor(_ ptr: UnsafeRawPointer) -> ObjectDescriptor {
    // C struct layout:
    // struct AROObjectDescriptor {
    //     const char* base;          // offset 0, 8 bytes
    //     int preposition;           // offset 8, 4 bytes
    //     // 4 bytes padding for pointer alignment
    //     const char** specifiers;   // offset 16, 8 bytes
    //     int specifier_count;       // offset 24, 4 bytes
    // };

    let basePtr = ptr.load(as: UnsafePointer<CChar>?.self)
    let base = basePtr.map { String(cString: $0) } ?? ""

    let prepInt = ptr.load(fromByteOffset: 8, as: Int32.self)
    let preposition = intToPreposition(Int(prepInt)) ?? .from

    // Account for padding: specifiers at offset 16, not 12
    let specsPtr = ptr.load(fromByteOffset: 16, as:
        UnsafeMutablePointer<UnsafePointer<CChar>?>?.self)
    let specCount = ptr.load(fromByteOffset: 24, as: Int32.self)

    var specifiers: [String] = []
    if let specs = specsPtr {
        for i in 0..<Int(specCount) {
            if let spec = specs[i] {
                specifiers.append(String(cString: spec))
            }
        }
    }

    return ObjectDescriptor(preposition: preposition, base: base, specifiers:
        specifiers, ...)
}
```

Critical insight: The padding between `preposition` (4 bytes at offset 8) and `specifiers` (pointer at offset 16) must match C struct alignment rules. Getting this wrong causes subtle memory corruption.

* * *

The @_cdecl Attribute

Swift's `@_cdecl` attribute exports a function with C linkage:

```swift
@_cdecl("aro_action_extract")
public func aro_action_extract(
    _ contextPtr: UnsafeMutableRawPointer?,
    _ resultPtr: UnsafeRawPointer?,
    _ objectPtr: UnsafeRawPointer?
) -> UnsafeMutableRawPointer? {
    return executeAction(verb: "extract", contextPtr: contextPtr,
                         resultPtr: resultPtr, objectPtr: objectPtr)
}
```

All 50 actions have thin wrappers that delegate to `ActionRunner`:

```swift
private func executeAction(
    verb: String,
    contextPtr: UnsafeMutableRawPointer?,
    resultPtr: UnsafeRawPointer?,
    objectPtr: UnsafeRawPointer?
) -> UnsafeMutableRawPointer? {
    guard let ctxHandle = getContext(contextPtr),
          let result = resultPtr,
          let object = objectPtr else { return nil }

    let resultDesc = toResultDescriptor(result)
    let objectDesc = toObjectDescriptor(object)

    // Execute through ActionRunner (same code path as interpreter)
    let actionResult = ActionRunner.shared.executeSync(
        verb: verb,
        result: resultDesc,
        object: objectDesc,
        context: ctxHandle.context
    )

    // Box result for return to C
    if let value = actionResult {
        return boxResult(value)
    }
    return boxResult("")
}
```

* * *

Synchronous Execution Challenge

Actions in the interpreter are `async`. But `@_cdecl` functions cannot be `async` (C has no concept of Swift concurrency). The solution: block the calling thread:

```swift
/// Synchronous action execution for compiled binaries
public func executeSync(
    verb: String,
    result: ResultDescriptor,
    object: ObjectDescriptor,
    context: ExecutionContext
) -> (any Sendable)? {
    let semaphore = DispatchSemaphore(value: 0)
    var resultValue: (any Sendable)?

    // Run async code on a detached task
    Task.detached {
        do {
            resultValue = try await self.execute(
                verb: verb,
                result: result,
                object: object,
                context: context
            )
        } catch {
            print("[ActionRunner] Error: \(error)")
        }
        semaphore.signal()
    }

    semaphore.wait()  // Block until async completes
    return resultValue
}
```

Warning: This can cause deadlocks if the executor pool is exhausted. Event handlers run on GCD threads (not the Swift cooperative executor) to prevent this.

* * *

Handler Registration

Compiled binaries register event handlers by passing function pointers:

```swift
@_cdecl("aro_runtime_register_handler")
public func aro_runtime_register_handler(
    _ runtimePtr: UnsafeMutableRawPointer?,
    _ eventType: UnsafePointer<CChar>?,
    _ handlerFuncPtr: UnsafeMutableRawPointer?
) {
    guard let ptr = runtimePtr,
          let eventTypeStr = eventType.map({ String(cString: $0) }),
          let handlerPtr = handlerFuncPtr else { return }

    let runtimeHandle =
        Unmanaged<AROCRuntimeHandle>.fromOpaque(ptr).takeUnretainedValue()

    // Capture handler address as Int (Sendable)
    let handlerAddress = Int(bitPattern: handlerPtr)

    runtimeHandle.runtime.eventBus.subscribe(to: DomainEvent.self) { event in
        guard event.domainEventType == eventTypeStr else { return }

        // CRITICAL: Run on GCD, not Swift executor
        await withCheckedContinuation { continuation in
            DispatchQueue.global().async {
                // Create context for handler
                let contextHandle = AROCContextHandle(runtime: runtimeHandle, ...)

                // Bind event data
                contextHandle.context.bind("event", value: event.payload)

                // Reconstruct function pointer
                guard let funcPtr = UnsafeMutableRawPointer(bitPattern:
        handlerAddress) else {
                    continuation.resume()
                    return
                }

                // Call compiled handler
                typealias HandlerFunc = @convention(c) (UnsafeMutableRawPointer?)
        -> UnsafeMutableRawPointer?
                let handlerFunc = unsafeBitCast(funcPtr, to: HandlerFunc.self)
                _ = handlerFunc(contextPtr)

                continuation.resume()
            }
        }
    }
}
```

* * *

Value Boxing

Values returned to C are boxed in reference-counted containers:

```swift
/// Boxed value for C interop
class AROCValue {
    let value: any Sendable

    init(value: any Sendable) {
        self.value = value
    }
}

/// Box a value for return to C
func boxResult(_ value: any Sendable) -> UnsafeMutableRawPointer {
    let boxed = AROCValue(value: value)
    return UnsafeMutableRawPointer(Unmanaged.passRetained(boxed).toOpaque())
}

/// Free a boxed value
@_cdecl("aro_value_free")
public func aro_value_free(_ valuePtr: UnsafeMutableRawPointer?) {
    guard let ptr = valuePtr else { return }
    Unmanaged<AROCValue>.fromOpaque(ptr).release()
}
```

* * *

Variable Operations

The bridge provides type-specific binding functions:

```swift
@_cdecl("aro_variable_bind_string")
public func aro_variable_bind_string(
    _ contextPtr: UnsafeMutableRawPointer?,
    _ name: UnsafePointer<CChar>?,
    _ value: UnsafePointer<CChar>?
) {
    guard let ptr = contextPtr,
          let nameStr = name.map({ String(cString: $0) }),
          let valueStr = value.map({ String(cString: $0) }) else { return }

    let contextHandle =
        Unmanaged<AROCContextHandle>.fromOpaque(ptr).takeUnretainedValue()
    contextHandle.context.bind(nameStr, value: valueStr)
}

@_cdecl("aro_variable_bind_int")
public func aro_variable_bind_int(
    _ contextPtr: UnsafeMutableRawPointer?,
    _ name: UnsafePointer<CChar>?,
    _ value: Int64
) {
    guard let ptr = contextPtr,
          let nameStr = name.map({ String(cString: $0) }) else { return }

    let contextHandle =
        Unmanaged<AROCContextHandle>.fromOpaque(ptr).takeUnretainedValue()
    contextHandle.context.bind(nameStr, value: Int(value))
}
```

Complex types (dictionaries, arrays) are passed as JSON strings:

```swift
@_cdecl("aro_variable_bind_dict")
public func aro_variable_bind_dict(
    _ contextPtr: UnsafeMutableRawPointer?,
    _ name: UnsafePointer<CChar>?,
    _ json: UnsafePointer<CChar>?
) {
    // ... parameter validation ...

    // Parse JSON to dictionary
    guard let data = jsonStr.data(using: .utf8),
          let parsed = try? JSONSerialization.jsonObject(with: data),
          let dict = parsed as? [String: Any] else {
        // Fallback: bind as string
        contextHandle.context.bind(nameStr, value: jsonStr)
        return
    }

    // Convert to Sendable and bind
    let sendableDict = convertToSendable(dict) as? [String: any Sendable] ?? [:]
    contextHandle.context.bind(nameStr, value: sendableDict)
}
```

* * *

Platform-Specific Type Handling

JSON deserialization behaves differently on Darwin vs Linux:

```swift
private func convertToSendable(_ value: Any) -> any Sendable {
    case let nsNumber as NSNumber:
        #if canImport(Darwin)
        // On Darwin, check CFBooleanGetTypeID for JSON booleans
        if CFGetTypeID(nsNumber) == CFBooleanGetTypeID() {
            return nsNumber.boolValue
        }
        #else
        // On Linux, use objCType to detect booleans
        let objCType = String(cString: nsNumber.objCType)
        if objCType == "c" || objCType == "B" {
            let intVal = nsNumber.intValue
            if intVal == 0 || intVal == 1 {
                return nsNumber.boolValue
            }
        }
        #endif
        // ...
}
```

* * *

Service Registration Limitations

A critical limitation: SwiftNIO doesn't work in compiled binaries:

```
init(runtime: AROCRuntimeHandle, featureSetName: String) {
    // ...

    // NOTE: Do NOT register AROHTTPServer (NIO-based) in compiled binaries.
    // SwiftNIO crashes because Swift's type metadata for NIO's internal
    // socket channel types is not available when the Swift runtime is
    // initialized from LLVM-compiled code.
    //
    // Instead, compiled binaries use native BSD socket HTTP server via
    // aro_native_http_server_start_with_openapi()
    self.httpServer = nil
}
```

This is why compiled ARO binaries use a native socket-based HTTP server instead of SwiftNIO.

* * *

Chapter Summary

The runtime bridge enables compiled code to call Swift:

1. **@_cdecl** exports Swift functions with C calling conventions
2. **Opaque pointers** wrap Swift objects for C code to hold
3. **Manual offset calculations** convert C structs to Swift types
4. **Semaphore blocking** makes async actions synchronous
5. **Function pointer casting** enables compiled handler callbacks
6. **Value boxing** provides reference-counted return values
7. **Platform-specific code** handles Darwin/Linux differences

The bridge is the most fragile part of native compilation. Memory layout assumptions, pointer casting, and synchronization all create potential failure modes. The interpreter path avoids these issues entirely—use it when stability matters more than startup time.

Implementation references: - `Sources/ARORuntime/Bridge/RuntimeBridge.swift` (1000+ lines) - `Sources/ARORuntime/Bridge/ActionBridge.swift` (500+ lines)

* * *

Next: Chapter 10 — Critical Assessment

§ Chapter 10: Critical Assessment

What This Chapter Is

This chapter provides an honest evaluation of ARO's design decisions—what works, what doesn't, and what we'd change with hindsight. Compiler design involves trade-offs; understanding them matters more than celebrating successes.

* * *

What Works Well

Uniform Syntax

ARO's rigid `<Action> the <Result> from the <Object>` structure enables simple tooling:

- **Parsing**: Every statement follows the same pattern
- **Analysis**: Data flow is explicit in the syntax
- **Formatting**: No debates about code style
- **Refactoring**: Find/replace works reliably

The constraint that seemed limiting during design has proven valuable in practice.

Predictable Execution

Sequential execution with explicit short-circuit points makes debugging straightforward:

```
<Extract> the <user> from the <request: body>.    (* 1 *)
<Validate> the <user> with <schema>.              (* 2 - might short-circuit *)
<Store> the <user> into the <user-repository>.   (* 3 *)
<Return> an <OK: status> for the <creation>.     (* 4 *)
```

When something fails, you know exactly which statement failed and why.

Event-Driven Architecture

The pub-sub model fits naturally with ARO's reactive use cases:

- HTTP routes → feature sets
- Domain events → handlers
- File changes → watchers

No explicit wiring code; the EventBus handles all routing.

Code-as-Documentation

The verbose syntax reads like prose:

```
<Extract> the <email> from the <user: profile: contact: email>.
<Send> the <welcome-message> to the <email>.
```

Even non-programmers can follow the logic.

* * *

What Doesn't Work

HTTP Disabled in Binary Mode

Problem: Compiled binaries crash when using SwiftNIO.

Technical Cause: SwiftNIO relies on Swift type metadata that isn't properly initialized when the Swift runtime starts from LLVM-compiled code. The crash occurs in `_swift_allocObject_` with a null metadata pointer when NIO tries to create socket channels.

Impact: HTTP servers only work in interpreter mode, negating the startup-time benefits of native compilation.

Workaround: A native BSD socket HTTP server is used for compiled binaries, but it's less robust than NIO.

Potential Fix: Investigate Swift runtime initialization sequence; may require changes to how we link against the Swift stdlib.

* * *

No LLVM Type Checking

Problem: Type errors in generated LLVM IR are caught at `llc` time, not generation time.

Technical Cause: We chose textual LLVM IR over the LLVM C API. Textual IR doesn't provide type checking during generation.

Impact: Malformed IR produces cryptic `llc` errors instead of helpful messages pointing to the source.

Example:

```
llc: error: program.ll:847:5: use of undefined value '%s0_result_desc'
```

Potential Fix: Use the LLVM C API (adds C++ build dependency) or add a verification pass before emitting IR.

* * *

Synchronous Action Execution

Problem: Actions block their calling thread in compiled binaries.

Technical Cause: `@_cdecl` functions cannot be `async`. We use `DispatchSemaphore.wait()` to block until async work completes.

Impact: Risk of thread pool exhaustion and deadlocks under load.

Example deadlock scenario: 1. Compiled handler starts on thread A 2. Handler action calls async service 3. Async service needs thread A (which is blocked) 4. Deadlock

Current Mitigation: Event handlers run on GCD threads, not the Swift cooperative executor.

Potential Fix: Design a custom async-compatible C bridge, or use libdispatch more carefully.

* * *

Function Pointer Fragility

Problem: Handler registration passes function pointers through integer casts.

Technical Cause: Swift closures aren't Sendable when they capture pointers. We cast to `Int`, then back to pointer in the callback.

```swift
let handlerAddress = Int(bitPattern: handlerPtr)
// ... later ...
let funcPtr = UnsafeMutableRawPointer(bitPattern: handlerAddress)
let handlerFunc = unsafeBitCast(funcPtr, to: HandlerFunc.self)
```

Impact: Works, but undefined behavior if address space assumptions are violated.

Potential Fix: Use stable function tables with integer indices instead of raw pointers.

* * *

Single Lookahead Limitation

Problem: The parser uses single-token lookahead, requiring disambiguation heuristics.

Technical Cause: Simplicity choice during initial development.

Impact Examples: - `/` could be division or regex start - `<` could be generic or less-than - Article usage affects parsing

Current Workaround: Context-dependent heuristics in the lexer/parser.

Potential Fix: PEG parser with unlimited lookahead, or packrat parsing.

* * *

Design Decisions We'd Reconsider

Preposition-Based Semantics

The idea: prepositions carry semantic meaning (from = source, to = destination, with = modification).

```
<Extract> the <data> from the <request>.   (* from = source *)
<Send> the <message> to the <user>.        (* to = destination *)
<Update> the <user> with <changes>.        (* with = modification *)
```

The Problem: Too subtle. Developers don't intuitively distinguish between: - `<Store> the <user> into the <repository>.` - `<Store> the <user> to the <repository>.`

Both seem valid, but only one is correct.

What We'd Change: Fewer prepositions with clearer rules, or remove semantic distinction entirely.

* * *

No Explicit Type Annotations

The idea: Types are inferred from usage and OpenAPI schemas.

```
<Extract> the <user> from the <request: body>.  (* user type comes from OpenAPI *)
```

The Problem: Tooling becomes harder: - IDE completion doesn't know types without analyzing the whole program - Error messages can't reference expected types - Documentation generators can't show types

What We'd Change: Optional type annotations with inference as fallback:

```
<Extract> the <user: User> from the <request: body>.
```

* * *

Business Activity Isolation

The idea: Variables are only visible within the same "business activity" scope.

The Problem: Confusing mental model. Developers expect either: - Global scope (everything visible everywhere) - Lexical scope (visible in nested blocks)

Business activity scope is neither.

What We'd Change: Explicit visibility keywords or simpler scoping rules.

* * *

Limitations Table

Limitation	Technical Cause	Impact	Difficulty to Fix
HTTP in binary	SwiftNIO metadata	No compiled HTTP servers	Hard
No IR type checking	Textual LLVM IR	Runtime-only errors	Medium
Sync action execution	@_cdecl constraint	Potential deadlocks	Hard
Function pointer fragility	Sendable constraints	Undefined behavior risk	Medium
Single lookahead	Parser simplicity	Disambiguation heuristics	Medium
Platform type handling	Darwin vs Linux differences	Boolean representation bugs	Easy
No stack traces in binary	LLVM generates minimal debug info	Hard debugging	Medium

* * *

What Students Should Learn

Simple Designs Have Hidden Complexity

ARO's "simple" Action-Result-Object syntax seemed easy to implement. But: - Articles (`a` / `an` / `the`) created parsing ambiguity - Prepositions required semantic classification -

Qualifier syntax (`<user: email>`) needed special handling - String interpolation required multi-token emission

The simplicity of the surface syntax hides significant implementation complexity.

Interop Layers Multiply Problems

Every language boundary creates friction: - **Swift → C**: Manual memory management, no async - **C → LLVM**: Manual struct layout, no type safety - **LLVM → native**: Platform-specific linking

A bug at any layer is hard to diagnose because you lose language guarantees.

“Code is Documentation” Requires Discipline

ARO’s verbose syntax aims to be self-documenting. But: - Bad variable names still make bad code - Complex logic still needs comments - The syntax enforces structure, not clarity

The language can encourage good practices; it can’t enforce them.

Trade-offs Are Real

Every design choice has costs:

We chose	We got	We lost
Textual LLVM IR	Simpler build, readable output	Type checking, performance
Immutable variables	Predictable data flow	Flexibility
Rigid syntax	Uniform tooling	Expressiveness
Event-driven model	Loose coupling	Explicit flow
@_cdecl bridge	C interop	Async support

There are no free lunches in language design.

* * *

Honest Assessment

ARO achieves its goals for a narrow use case: declarative, auditable business features with predictable execution. It's not a general-purpose language and shouldn't try to be.

The native compilation path works for some scenarios but has significant limitations. The interpreter is more reliable and should be the default choice.

The language design is opinionated—some developers will find it liberating, others constraining. That's intentional; ARO prioritizes team legibility over individual expressiveness.

For students: ARO is an example of what a constrained DSL looks like in practice. Study the trade-offs, not just the implementation.

* * *

Future Directions

If we were to continue development:

1. **Fix NIO in binary**: Investigate Swift runtime initialization deeply
2. **Add optional types**: `<user: User>` syntax for explicit typing
3. **Improve error messages**: Source-mapped errors from LLVM failures
4. **Reduce prepositions**: Simplify to 3-4 with clear semantics
5. **Add debugging support**: Better stack traces in compiled mode

These are not planned—they're lessons for future DSL designers.

* * *

Next: Appendix — Source Map

§ Appendix: Source Map

This appendix provides a quick reference to key source files in the ARO implementation.

* * *

Parser Package (`Sources/AROParser/`)

File	Lines	Description
`Lexer.swift`	~700	Tokenization, string interpolation, regex detection
`Token.swift`	~200	Token types, articles, prepositions
`Parser.swift`	~1700	Recursive descent + Pratt parsing
`AST.swift`	~1300	AST node definitions, visitor pattern
`SemanticAnalyzer.swift`	~800	Symbol tables, data flow analysis
`SymbolTable.swift`	~200	Symbol storage, visibility levels
`Compiler.swift`	~150	Pipeline orchestration
`DiagnosticEngine.swift`	~100	Error collection and reporting

Key Entry Points

Parsing a source file:

```
let lexer = Lexer(source: sourceCode)
let tokens = lexer.tokenize()
let parser = Parser(tokens: tokens)
let program = try parser.parse()
```

Semantic analysis:

```
let analyzer = SemanticAnalyzer()
let analyzed = try analyzer.analyze(program)
```

* * *

Runtime Package (`Sources/ARORuntime/`)

Core (`Core/`)

File	Lines	Description
`ExecutionEngine.swift`	~850	Program execution, handler registration
`FeatureSetExecutor.swift`	~600	Statement execution, control flow
`RuntimeContext.swift`	~300	Variable binding, service access
`Runtime.swift`	~150	Top-level runtime container

Actions (`Actions/`)

File	Lines	Description
`ActionRegistry.swift`	~100	Verb → implementation mapping
`ActionImplementation.swift`	~80	Action protocol definition
`ActionRunner.swift`	~200	Unified execution (sync/async)
`BuiltIn/ExtractAction.swift`	~150	Data extraction
`BuiltIn/ComputeAction.swift`	~300	Computations (length, hash, arithmetic)
`BuiltIn/ReturnAction.swift`	~100	Response generation
`BuiltIn/StoreAction.swift`	~200	Repository storage
`BuiltIn/EmitAction.swift`	~100	Event emission
(46 more actions)		

Events (`Events/`)

File	Lines	Description
`EventBus.swift`	~290	Pub-sub routing, in-flight tracking
`EventTypes.swift`	~320	RuntimeEvent implementations
`StateGuard.swift`	~130	Event filtering by entity state

Bridge (`Bridge/`)

File	Lines	Description
`RuntimeBridge.swift`	~1000	Runtime lifecycle, variable ops
`ActionBridge.swift`	~500	@_cdecl action wrappers
`ServiceBridge.swift`	~300	HTTP/File/Socket C interface

* * *

Compiler Package (`Sources/AROCompiler/`)

File	Lines	Description
`LLVMCodeGenerator.swift`	~1900	AST → LLVM IR transformation
`Linker.swift`	~1400	Object emission, platform linking
`Compiler.swift`	~200	High-level compilation API

LLVMCodeGenerator Key Methods

Method	Purpose
`generate(_:)`	Main entry point
`generateHeader()`	Module metadata, target triple
`generateTypeDefinitions()`	Struct types for descriptors
`generateExternalDeclarations()`	@_cdecl function declarations
`generateStringConstants()`	Collect and emit string pool
`generateFeatureSet(_:)`	Feature set → LLVM function
`generateStatement(_:)`	Statement → descriptor + call
`generateForEachLoop(_:)`	Sequential iteration
`generateParallelForEachLoop(_:)`	Parallel with body function
`generateMain()`	Entry point, handler registration

* * *

CLI Package (`Sources/AROCLI/`)

File	Lines	Description
`main.swift`	~50	Entry point
`Commands/RunCommand.swift`	~150	`aro run` - interpreter execution
`Commands/BuildCommand.swift`	~200	`aro build` - native compilation
`Commands/CheckCommand.swift`	~80	`aro check` - syntax validation
`Commands/CompileCommand.swift`	~100	`aro compile` - IR only

* * *

File Dependencies

```
                  ┌─────────────────┐
                  │   AROCLI        │
                  └────────┬────────┘
                           │
                           │
          ┌────────────────┼────────────────┐
          │                │                │
          ▼                ▼                ▼
┌─────────────┐   ┌─────────────┐   ┌─────────────┐
│ AROParser   │   │ ARORuntime  │   │ AROCompiler │
└──────┬──────┘   └──────┬──────┘   └──────┬──────┘
       │                 │                 │
       │                 │                 │
       └────────────┬────┴─────────────────┘
                    │
                    ▼
            ┌─────────────┐
            │ Foundation  │
            └─────────────┘
```

* * *

Build-Time Dependencies

Package	Used By	Purpose
`swift-argument-parser`	AROCLI	Command-line parsing
`swift-nio`	ARORuntime	HTTP server (interpreter)
`async-http-client`	ARORuntime	HTTP client
LLVM	AROCompiler	IR → object compilation

* * *

Reading Order for Understanding

1. **Start with syntax**: `Token.swift`, `Lexer.swift`
2. **Understand AST**: `AST.swift`, `Parser.swift`
3. **See execution**: `FeatureSetExecutor.swift`, `ActionImplementation.swift`
4. **Study events**: `EventBus.swift`, `EventTypes.swift`
5. **Explore compilation**: `LLVMCodeGenerator.swift`
6. **Understand bridge**: `RuntimeBridge.swift`, `ActionBridge.swift`

* * *

Finding Specific Functionality

If you want to understand...	Look at...
How tokens are classified	`Lexer.swift:classifyIdentifier()`
How expressions are parsed	`Parser.swift:parseExpression()`
How actions are registered	`ActionRegistry.swift`
How events are dispatched	`EventBus.swift:publishAndTrack()`
How LLVM IR is structured	`LLVMCodeGenerator.swift:generateHeader()`
How C calls Swift	`ActionBridge.swift:executeAction()`
How pointers are managed	`RuntimeBridge.swift:AROCRuntimeHandle`

* * *

Test Files

Directory	Contents
`Tests/AROParserTests/`	Lexer, parser, AST tests
`Tests/ARORuntimeTests/`	Action, context, event tests
`Tests/AROCompilerTests/`	IR generation tests
`Tests/AROIntegrationTests/`	End-to-end tests
`Examples/`	Working example applications

* * *

Documentation Files

File	Purpose
`CLAUDE.md`	Build commands, architecture overview
`README.md`	Project introduction
`OVERVIEW.md`	Developer documentation
`Proposals/*.md`	Language specifications
`Book/TheLanguageGuide/`	User documentation
`Book/TheConstructionStudies/`	This book

* * *

End of The Construction Studies