The American Journal of Applied Sciences
ISSN 2689-0992 | Open Access

TYPE Original Research
PAGE NO. 13-19
DOI 10.37547/tajas/Volume07Issue03-03

Check for updates

Swift Proto Parser: A Framework for Native Tools for Protocol Buffers

Kish Aleksei

Senior Software Engineer / Technical Owner at Semrush Spain, Barcelona

OPEN ACCESS

SUBMITTED 16 January 2025
ACCEPTED 12 February 2025
PUBLISHED 12 March 2025
VOLUME Vol.07 Issue03 2025

CITATION
Kish Aleksei. (2025). Swift Proto Parser: A Framework for Native Tools for Protocol Buffers. The American Journal of Applied Sciences, 13–19. https://doi.org/10.37547/tajas/Volume07Issue03-03

Abstract: This article presents the development of a native Swift library that parses .proto files—commonly used in Protocol Buffers (Protobuf)—without relying on the external protoc tool. It addresses a critical gap in Swift's ecosystem by enabling direct, run-time access to Protobuf schemas. Drawing on the theoretical foundations of lexical analysis, the paper outlines how the library's lexer and parser transform raw .proto input into descriptor structures amenable to dynamic serialization and integration with SwiftProtoReflect. The proposed approach substantially reduces dependency on external code generation and simplifies tasks such as documentation generation, code formatting, linting, and IDE-based tooling. Experimental evidence and test results show that the library can reliably handle core Protobuf features, including nested messages, enums, services, and custom options, thereby laying the groundwork for a fully Swift-based workflow in gRPC and Protocol Buffers development.

Keywords: Swift, Protocol Buffers, lexical analysis, parsing, dynamic serialization, gRPC, reflection, .proto files.

Introduction: Modern distributed systems and microservice architectures increasingly adopt the gRPC protocol based on Protocol Buffers (Protobuf) for high-performance data serialization [2]. However, until recently, the Swift language lacked fully native tools for flexible and dynamic interaction with Protobuf messages. Developers had to rely on protoc and associated utilities, complicating project compilation and deployment [5]. This issue is particularly critical in environments with frequent service specification updates, where each .proto file modification necessitates code generation and application recompilation [3].

The problem becomes even more pronounced when

dynamic serialization and deserialization are required without statically generated classes—such as in cases where the set of Protobuf messages may change dynamically. The SwiftProtoReflect project addresses some of the challenges related to dynamic access to Protobuf metadata, but achieving full functionality requires a native lexer and parser for .proto files in Swift [4]. The absence of such tooling generates both scientific and practical interest in the lexical [6] and syntactic analysis of .proto files within the Swift ecosystem.

Existing research confirms gRPC's potential for low-latency interservice communication and efficient data serialization [7]. However, these studies typically do not focus on Swift-specific challenges, where dynamic message reflection remains limited [2]. In our previous works [3, 4], we introduced the SwiftProtoReflect library to enable dynamic serialization in Swift, thereby demonstrating the potential for runtime interaction with Protobuf data. These contributions provided a solid foundation and inspired further innovations. Building on that groundwork, the tool presented in this paper extends our earlier efforts by incorporating lexical and syntactic parsing of .proto files, thereby broadening the scope of Protobuf handling in Swift.

From a theoretical perspective, significant contributions to lexical analysis and parser construction have been summarized in the work by Pai T.V. and Aithal P.S. [6], emphasizing the importance of correctly segmenting input data into tokens and the lexer's role as the initial stage of a compiler or similar tool. However, specific data formats such as .proto require extensions to classical approaches, considering comments, escaped strings, nested structures, and field compatibility rules.

While existing research and tools—particularly those designed for C++, Java, and Python—already provide Protocol Buffers parsers, a gap remained in Swift: no fully native solution could parse .proto files natively without invoking external protoc [4].

The scientific and practical gap lies in the absence of a systematic implementation of lexicon-syntactic analysis for .proto descriptions specifically tailored for Swift. Such an implementation should enable:

- Eliminating the need for code generation via protoc and external scripts.
- Dynamic interaction with message and service structures (e.g., in IDE plugins, formatting tools, linters).
- Retaining metadata (comments, options) for documentation generation and analysis.

The objective of this study is to develop and demonstrate Swift Proto Parser, a native Swift library providing a complete pipeline for the lexical and syntactic analysis of .proto files. Based on the resulting abstract syntax tree (AST) or descriptors, dynamic interaction with messages and services can be achieved using SwiftProtoReflect or other tools.

The hypothesis states that implementing a Protocol Buffers parser—including both lexer and syntax analyzer—natively in Swift will eliminate reliance on protoc, thereby increasing the flexibility of gRPC service development in Swift projects, accelerating integration of changes, and simplifying tooling (at the level of IDEs, linters, and refactoring utilities).

1. Theoretical aspects of lexical analysis and working with Protobuf in Swift

Lexical analysis, also known as tokenization, is the first stage in transforming the source text of any formal specification or program code into a machine-readable representation [1]. It involves breaking a stream of characters into lexemes, each classified and converted into a token of a specific type, such as an identifier, keyword, operator, or literal. The resulting set of tokens is later interpreted by a syntax parser, which relies on a context-free grammar. In classical compilers and interpreters, this approach effectively separates the task of "recognizing basic language constructs" from "building a syntax tree" [1].

However, lexical analysis is not only essential in compilers for widely recognized programming languages (such as C or Java) but also in numerous applied data formats, including Protocol Buffers (Protobuf) files that define message and service structures. The challenge lies in the fact that each format has unique characteristics, such as string escaping methods, nested block structures, and support for specific comments or directives, all of which must be accounted for when building a lexer [6]. Failing to address these nuances would make it impossible to construct a correct syntax tree and subsequently operate on the data structure (e.g., for documentation generation or dynamic serialization).

Protobuf files include descriptions of messages (message), services (service), file-level directives (syntax, import, option), and other constructs. Each construct contains distinct lexemes, such as curly and round brackets, colons, semicolons, and keywords like message, service, enum, and rpc [8]. String literals, which may contain escape sequences (\n, \t, \\), as well as single-line and multi-line comments, are also

significant—not only for readability but also for future documentation generation [3].

Table 1 provides a generalized classification of key token categories in .proto files, covering common cases that the lexer must handle.

Table 1. Main Token Categories in Protobuf (.proto Files)

Category	Examples of Lexemes	Comments
Keywords	message, service, enum, rpc	Reserved words defining the structure of .proto files.
Brackets and Delimiters	{, }, (,), ;, =, .	Used to define block boundaries, enumerations, and expressions.
Identifiers	UserProfile, test_field, APIv2	Names of messages, fields, and services; must conform to formal language rules.
Literals	"hello", 123, false	String and numeric constants, boolean values; may contain escape sequences.
Comments	// ..., /* ... */	Can be preserved by the lexer for documentation generation or other purposes (linters, explanations, etc.).

From a theoretical standpoint [6], a lexer for .proto files is built on the same principles as any other: rules (regular expressions or finite-state automata) define valid token forms, and a procedure iterates through the input string to generate corresponding objects. The most critical aspects for Protobuf include:

- Correct handling of escaped strings (e.g., \"Hello\").
- Support for nested declarations, where one message construct can be contained within another.
- Retaining comments (at least by associating them with the nearest tokens) to allow their use in documentation generation, auto-suggestions, etc. [4].

These features add complexity compared to more straightforward languages. However, the core logic of lexical analysis remains unchanged: converting a stream of characters into a stream of tokens, which will subsequently be parsed by the syntax analyzer.

The Protocol Buffers (Protobuf) protocol is a mechanism for serializing structured data, introduced by Google [8]. In a .proto file, a developer can define messages with their fields and assigned numbers, as well as services with RPC methods. The compilation process of a .proto file using standard protoc tools generates language-specific classes or structures that enable serialization and deserialization of data into either the binary Protobuf format or its text-based (JSON) equivalent [5].

However, previous studies [3, 4] have highlighted the absence of a built-in mechanism in Swift for dynamic interaction with .proto descriptions without code generation. Such functionality requires two key components: first, the elimination of a static approach, where every schema change necessitates re-running protoc and rebuilding the application, and second, a runtime mechanism capable of constructing message structures and handling serialization/deserialization dynamically based on .proto data (or via gRPC Reflection).

The study in [3] introduced SwiftProtoReflect, a library that enables dynamic access to Protobuf descriptors. It allows querying the structure of a message and assigning field values at runtime, without generating static code. This significantly reduces friction when modifying a service. However, SwiftProtoReflect still relies on obtaining descriptors externally—a complete set of metadata about messages, fields, and services. Typically, these metadata are retrieved either from a server (via the gRPC Reflection API) or from pre-generated .proto files (via protoc).

This leads to a paradox: if a .proto file is physically present in the project but protoc is not used, there is still no built-in mechanism to extract descriptors for SwiftProtoReflect. Until recently, Swift lacked a native

lexical analysis tool for parsing .proto files [4]. This is a fundamental limitation of Swift: the language does not provide a built-in tool for flexible runtime parsing of formal specifications, unlike other languages with more advanced reflection capabilities [2].

Consequently, a native Swift library capable of parsing .proto files and constructing descriptors directly—without invoking external utilities—becomes essential. As outlined in the introduction, such a library would eliminate the need for external code generation, enable dynamic interaction with message and service structures, and retain extended metadata for documentation, linting, formatting, and related tasks. Additionally, it would seamlessly integrate with tools for dynamic serialization (e.g., SwiftProtoReflect) and visualization (e.g., potential plugins for Xcode, VSCode, or other IDEs).

Considering general lexical analysis theory [1, 6] and practical experience with Protobuf [8], it can be concluded that .proto grammar is parseable using classical methods, but only if specific nuances are accounted for—such as escaped strings, nested declarations, comments, and other structural elements. Previous work [4] has already emphasized the importance of retaining such "secondary" elements like comments, which can significantly enhance the Protobuf toolchain.

Thus, Swift Proto Parser aligns with this broader concept: it handles lexical and syntactic analysis, producing descriptors (or equivalent AST structures). These descriptors can then be immediately used for dynamic serialization(e.g., SwiftProtoReflect), migration tools, linters, formatters, and automated documentation generation. In other words, this approach establishes a self-sufficient ecosystem for working with Protobuf in Swift, eliminating the need for external C++/Java-based tools or intermediate files that compromise the clean architecture of a Swift project.

Demonstrating the practical significance of this solution, it is important to note that Swift development, especially on Apple platforms, often faces challenges when integrating third-party binary tools like protoc. These challenges arise due to architectural differences (e.g., ARM64 vs x86_64), code-signing requirements, and other platform-specific constraints. A fully native Swift parser eliminates these issues, ensuring a consistent and streamlined approach to building and deploying Swift-based applications [3, 4].

2. Practical implementation and application of Swift Proto Parser

The internal structure of Swift Proto Parser can be logically divided into three main modules:

1. Lexer – responsible for tokenizing the input .proto file, identifying keywords, identifiers, operators, brackets, string literals, and comments.

2. Parser – constructs an intermediate representation (AST) or file descriptor based on the token stream, incorporating information about declared messages, fields, enums, services, options, etc.

3. Validation - after parser is done, we have to ensure all proto elements are correct, not syntactically (it's parser's responsibility) but grammatically, if everything complies with protobuf.

3. Data structures – a set of Swift types (e.g., FileNode, MessageNode, EnumNode, ServiceNode, etc.) that store the final model of the .proto file.

The Lexer is implemented as a class, with the core method nextToken() sequentially reads characters and generates tokens. It handles:

- Recognition of keywords (message, service, rpc, etc.) with case sensitivity (if unmatched, the token is classified as an identifier).
- Support for string literals with escaped characters (\n, \t, \\, etc.).
- Recognition of numeric literals (including integers, floating-point numbers, and exponential notation).
- Processing of comments (// and /* ... */), which are stored in additional token fields for potential restoration or analysis [4].

In the LexerTests.swift file [9], the unit test testSingleCharacterTokensWithoutSpaces verifies whether the lexer correctly interprets a sequence of characters such as {}, [], <>, and others without separating spaces:

```
func
testSingleCharacterTokensWithoutSpaces()
throws {
    let input = "{}[]<>(),;="
    let tokens = try getAllTokens(from: input)
    // Expecting 12 operators + EOF
    XCTAssertEqual(tokens.count, 13)
    XCTAssertEqual(tokens[0].type, .leftBrace)
    // ...
    XCTAssertEqual(tokens[11].type, .period)
    XCTAssertEqual(tokens[12].type, .eof)
}
```

This test ensures that the lexer correctly "slices" tokens regardless of the presence of spaces [4].

After converting the input text into a stream of tokens, the Parser begins its work. It follows rules similar to the context-free grammar of the .proto format [8]. Each call to parseFile() creates a FileNode structure containing:

- syntax – the syntax version (default: "proto3"),
- package – the package name,
- imports – a list of imported files,
- options – global options,
- messages, enums, services – declarations of messages, enums, and services.

In ParserTests.swift, the test testBasicMessageDefinition illustrates how an input text of the following form:

```
message Test {
   string name = 1;
   int32 id = 2;
   bool active = 3;
}
```

is parsed into the corresponding tree node:

```
func testBasicMessageDefinition() throws {
   let input = """
   message Test {
      string name = 1;
      int32 id = 2;
      bool active = 3;
   }
   """
   let file = try parse(input)
   XCTAssertEqual(file.messages.count, 1)
   let message = file.messages[0]
   XCTAssertEqual(message.name, "Test")
   XCTAssertEqual(message.fields.count, 3)
}
```

This test verifies the message name (Test) and the number of fields. If errors are encountered, such as duplicate field numbers, the library generates the appropriate exception ParserError.duplicateFieldNumber, which is also covered by tests [4].

The parser's output consists of Swift objects (structures) that reflect the semantics of a .proto file. For example, a MessageNode stores:

- name (the message name),
- fields (a list of fields, each described by type, number, and options),
- nested messages and enums.

Similar structures exist for EnumNode, ServiceNode, and FileNode. This representation makes the data easily usable for further operations, such as documentation generation or dynamic serialization [3].

Error handling and comment processing are integral aspects of the system: if the input does not comply with syntactic rules (such as incorrect syntax, invalid field number, duplicate field, or incorrect keyword), the lexer throws a LexerError while the parser raises a ParserError, both of which include the exact line and column where the issue occurred; moreover, instead of simply ignoring comments, the lexer preserves them in additional fields—namely, leadingComments and (in a planned extension) trailingComment —thereby enabling their restoration for auto-documentation or visualization tools [9].

The project contains the main directory Sources/SwiftProtoParser, which includes the implementations of Lexer.swift, Parser.swift, and the associated data structure modules. Tests are located in Tests/SwiftProtoParserTests. Files like LexerTests.swift and ParserTests.swift contain test cases covering a wide range of constructs—from syntax and package declarations to nested messages and edge cases (e.g., duplicate numbers, invalid characters, unmatched brackets). This approach ensures high reliability and reproducibility [4].

After performing lexical and syntactic analysis, Swift Proto Parser generates a structured representation that fully reflects the contents of the input .proto file. The following example illustrates a core test case, confirming correct parsing of the syntax declaration.

In most .proto files, the first line is syntax = "proto3";. The test case testValidSyntaxDeclaration [9] verifies that the parser correctly reads this directive and stores it inFileNode.syntax:

```
func testValidSyntaxDeclaration() throws {
   let input = """
   syntax = "proto3";
   """
   let file = try parse(input)
   XCTAssertEqual(file.syntax, "proto3")
}
```

If the syntax version is invalid (e.g., "proto2" or a non-string value), an invalidSyntaxVersion error is thrown

[8].

Next, .proto files often declare package some.package; or include imports such as import "other.proto";. The parser supports these constructs, including import public and import weak, storing them in file.imports (see testValidImports in ParserTests.swift).

Support for message, enum, service, and rpc

- Message: The parser verifies field name correctness, ensures unique field numbers, and checks field types (scalar, map, nested message).
- Enum: Options such as option allow_alias = true; are supported. If allow_alias is disabled, duplicate numbers are not allowed.
- Service: The parser processes rpc ... (Request) returns (Response); methods and options within RPC bodies. Stream syntax is supported.

The following is a simplified example (from testBasicService in ParserTests.swift), where the parser extracts a service and its methods:

```
func testBasicService() throws {
    let input = """
    service Greeter {
        rpc SayHello (HelloRequest) returns
(HelloResponse);
    }
    """
    let file = try parse(input)
    let service = file.services[0]
    XCTAssertEqual(service.rpcs.count, 1)
    XCTAssertEqual(service.rpcs[0].name,
"SayHello")
}
```

However, Swift Proto Parser itself does not perform serialization or deserialization. Instead, it constructs a structure (AST/descriptors) that can be passed to SwiftProtoReflect [3]. The latter can dynamically marshal and unmarshal messages if provided with a field descriptor (type, number, options, nesting, etc.). When a .proto file is present in a project, the following steps are performed:

1. Parse the .proto file using SwiftProtoParser.parseFile(...).
2. Obtain a structured model (a list of messages, services, etc.).
3. Pass the model to SwiftProtoReflect, which allows dynamic message object creation, field assignments, and serialization into the Protobuf binary format—without calling protoc.

This approach ensures a "pure Swift stack", eliminating dependencies on external utilities [4].

Practical applications of Swift Proto Parser beyond direct .proto file parsing

1. Custom documentation generators

- Extraction of comments, message/service declarations for automated description generation (Markdown, HTML, PDF).
- Visualization of relationships (e.g., identifying which RPC calls which service, and how messages are nested within each other).

2. Formatters and linters for .proto files

- Automatic formatting: adjusting indentation, spacing, and sorting fields by number.
- Linters: enforcing corporate style guidelines, detecting restricted constructs.
- Refactoring: bulk renaming of fields and services. When combined with SwiftProtoReflect, even associated Swift code can be updated automatically to reflect schema changes.

3. Visual editors and IDE integration

- Plugins for Xcode, Visual Studio Code, JetBrains IDEs, etc., enabling live syntax highlighting, keyword suggestions, and file structure validation.
- API Prototyping: developers can modify or add services and fields dynamically, seeing real-time results.

4. Migration tools

- Bulk updates for legacy .proto files (e.g., renaming deprecated fields or transitioning from proto2 to proto3).
- Automated version conflict resolution, useful when multiple development branches introduce different changes to .proto files.
- Validation of "evolving" fields to prevent backward compatibility issues.

5. Impact analysis

- When a field is removed or renamed, identifying all occurrences across .proto files or even in Swift code if the parser is linked to the codebase.
- Compatibility warnings for previously released clients.

All these use cases rely on a single key feature—retrieving the syntactic structure of a .proto file in a native Swift format. Previously, similar tasks required manual protoc execution or complex workarounds, slowing development and complicating build processes

[2, 4]. The presence of Swift Proto Parser simplifies the creation of a comprehensive set of tools and enhances flexibility in managing Protobuf schemas.

CONCLUSION

The study demonstrates how a native parser for .proto files, written entirely in Swift, can eliminate the complexities associated with external tools like protoc and markedly expand the potential for Protobuf-based solutions within Swift projects. By combining theoretical insights into lexical analysis with a practical and test-driven implementation, the library achieves full support for Protobuf language features while retaining comments and other metadata vital for advanced tooling. Integrated with SwiftProtoReflect, it offers a coherent environment for dynamic serialization and real-time adaptation of schema definitions. This synergy addresses a longstanding limitation in the Swift ecosystem, transforming what was once a compilation-stage dependency into a flexible, run-time amenity that significantly enhances both developer productivity and project maintainability.

REFERENCES:

Aho A. V. et al. Lexical analysis //Compilers: Principles, Techniques, and Tools. 2nd ed. Pearson Education: Inc. – 2006. – С. 109-90.

Barik R. et al. Optimization of Swift Protocols //Proceedings of the ACM on Programming Languages. – 2019. – Т. 3. – №. OOPSLA. – С. 1-27.

Kish A. Proto Reflection Implementation For Dynamic Interaction With gRPC Services In High-load Systems //The American Journal Of Engineering And Technology. – 2024. – vol. 6. – №. 12 – pp. 84-91.

Kish A. The Future of Api Development: How Proto Reflection Transforms GRPC Interactions //International Journal of Scientific Research and Engineering Development. – 2025. – vol. 8. – №. 1 – pp. 406-411.

Nimpattanavong C. et al. Improving Data Transfer Efficiency for AIs in the DareFightingICE using gRPC //2023 8th International Conference on Business and Industrial Research (ICBIR). – IEEE, 2023. – pp. 286-290.

Pai T V., Aithal P. S. A systematic literature review of lexical analyzer implementation techniques in compiler design //International Journal of Applied Engineering and Management Letters (IJAEML). – 2020. – Т. 4. – №. 2. – С. 285-301.

Sangwai A. et al. Barricading System-System Communication using gRPC and Protocol Buffers //2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE). – IEEE, 2023. – С. 1-5.

Google. (2020). Protocol Buffers Documentation. Retrieved from https://developers.google.com/protocol-buffers

Swift-protoparser. Retrieved from https://github.com/truewebber/swift-protoparser