# Technical Design Specification: ENFAStepLexer-StepParser Enhancements

Version: 1.0

Date: 2025-10-22

Status: FINAL

## 1. Introduction

This document details the required technical changes to transition the ENFAStepLexer-StepParser system from its current state (containing placeholders and simplifications) to a robust, fully featured parsing solution. It addresses shortcomings in lexical analysis (regex engine, ambiguity, Unicode), parsing logic (GLR stability, error handling), grammar loading (semantic actions, inheritance), CognitiveGraph integration, refactoring operations, and testing.

## 2. Goals

* Replace placeholder implementations with production-ready logic.
* Implement missing PCRE2 features outlined in the project roadmap.
* Ensure stable and correct handling of ambiguous grammars and recursive rules.
* Implement a functional execution engine for semantic actions and projections.
* Fully integrate CognitiveGraph for accurate semantic analysis and querying.
* Provide working implementations for location-based refactoring operations.
* Establish comprehensive test coverage and performance benchmarks.
* Achieve robustness against invalid inputs and grammars.

## 3. DevelApp.StepLexer Enhancements

### 3.1. Core Lexical Analysis Engine: Two-Pass, Multi-Path, Forward-Only Algorithm

* **Architectural Mandate:** The StepLexer's engine is a proprietary, forward-only, enhanced non-deterministic finite automaton (eNFA) algorithm. It is designed specifically for high performance and ReDoS vulnerability prevention through a two-pass matching process, multi-path ambiguity resolution, and a zero-copy, UTF-8 native foundation. The lexer has sole and complete responsibility for all regex processing.
* **Pattern Compilation:**
  + Upon loading, each TokenRule's regex pattern is compiled into one or more eNFA representations.
  + All individual eNFAs are then merged into a single, unified state machine structure. This allows for the efficient, simultaneous evaluation of all possible token patterns at any given point in the input stream.
* **Matching Process (Two-Pass Execution):**
  + **Phase 1: Pathfinding Scan (Forward-Only, Semantically Simplified)**
    - The lexer traverses the input stream byte-by-byte, advancing through the unified eNFA structure.
    - This phase is "semantically simplified" to guarantee linear performance. It explicitly **ignores ReDoS-prone constructs like capture groupings**. Its sole purpose is to rapidly determine *if* and *where* one or more TokenRule patterns have a high-likelihood match.
    - The process is strictly forward-only, with **no backtracking**. Fast lookbehind operations are achieved by leveraging the ZeroCopyStringView structure.
  + **Phase 2: Grouping and Content Resolution (On Confirmed Match)**
    - This phase is triggered only *after* Phase 1 has confirmed a potential match for a specific TokenRule over a defined region of the input.
    - It performs a detailed "full grouping" analysis on the already-identified ZeroCopyStringView slice corresponding to the match.
    - The purpose of this phase is to resolve the contents of capture groups and validate other complex semantics **without re-scanning the input**, thus isolating the computationally expensive parts of regex matching from the initial traversal.
* **Ambiguity and LexerPath Lifecycle:**
  + The lexer maintains one or more LexerPaths, each representing a possible tokenization sequence.
  + **Path Splitting:** When the input stream could satisfy multiple TokenRule patterns simultaneously (ambiguity), the current LexerPath is **split**. Each new path is forked to track one of the potential regex matches exclusively.
  + **Path Termination:** A LexerPath is terminated ("killed") under two conditions:
    1. **Lexical Invalidity:** The path's associated regex pattern no longer matches the subsequent input.
    2. **Grammatical Invalidity:** The StepParser signals that the token this path would produce is not grammatically valid in any of its current parsing states.
* **Error Handling:**
  + An error state is entered only when **all active LexerPaths have been terminated**.
  + At this point, a comprehensive error report is generated by combining state information from both the lexer and the parser based on the last viable LexerPath(s).

### 3.2. PCRE2 Feature Implementation

* **Current State:** Several features like inline modifiers, \Q...\E, and comments are missing, as noted in PCRE2-Support.md.
* **Change:** Implement parsing and matching logic for features listed in Phase 2 of the project roadmap.
* **Implementation:**
  + **Inline Modifiers ((?i), (?m), etc.):** Modify ScanGroup in StepLexer.cs to recognize modifier sequences. Introduce state flags within the matching engine's state to track active modifiers (e.g., case-insensitivity). These modifiers will alter the behavior of the eNFA matching logic for the path they are on.
  + **Literal Text (\Q...\E):** Enhance ScanEscapeSequence to recognize \Q, then scan byte-by-byte until \E is found. The enclosed content will be treated as a sequence of literal bytes, bypassing any special character interpretation. This will generate LiteralText tokens.
  + **Comments ((?#...)):** Enhance ScanGroup to recognize (?# and scan until the matching closing ). The content will be parsed as a RegexComment token, which will be subsequently ignored by the matching engine.
* **Affected Files:** StepLexer.cs, TokenType enum.

### 3.3. Ambiguity Resolution Logic

* **Current State:** The methods CanSplitToken, GenerateSplitTokens, and SelectBestAlternative in StepLexer.cs are simplified placeholders and do not correctly model the separation between lexical and grammatical ambiguity.
* **Change:** Implement a robust ambiguity management system where lexical ambiguity is handled exclusively by cloning LexerPaths, while grammatical ambiguity is handled separately by the StepParser.
* **Implementation:**
  + **Structural Relationship:** Each ParserPath can be associated with one or more LexerPaths. A single ParserPath represents a hypothesis about the grammatical structure, and its associated LexerPath(s) represent the various lexical possibilities for the *next token* on that grammatical path.
  + **Ambiguity Detection:** During pattern compilation, states in the unified eNFA that can transition based on more than one TokenRule will be marked as ambiguity points.
  + **Path Splitting (Lexical Ambiguity):** When the Phase 1 scan, operating on behalf of a single ParserPath, encounters an ambiguity point, only the current **LexerPath is cloned**. The original ParserPath now becomes associated with multiple LexerPaths, each tracking one of the potential token matches. **The ParserPath itself is not cloned at this stage.**
  + **Resolution Strategy (Parser-Directed):**
    1. Continuous Elimination: Lexical ambiguities are resolved by eliminating invalid LexerPaths associated with a ParserPath. This occurs through:  
       a. Lexical Elimination: LexerPaths are killed when their associated pattern no longer matches the input.  
       b. Grammatical Elimination: The StepParser signals to the lexer which potential tokens (and thus, which LexerPaths) are grammatically valid for the current ParserPath. Invalid LexerPaths are killed. (This is detailed in Section 3.5).
    2. Grammar-Specified Tie-Breaking: If multiple LexerPaths associated with a single ParserPath are still viable after a token is fully formed, the StepLexer queries the StepParser.  
       a. The parser will inspect the active GrammarDefinition for a LexicalResolutionStrategy flag.  
       b. If the strategy is specified in the grammar (e.g., "prefer longest match"), the parser instructs the lexer to apply the tie-breaking rule (longest match, then priority), kill the non-preferred LexerPaths, and proceed with the single resolved token.  
       c. If no strategy is specified (the default), the lexical ambiguity is passed up to the parser. This will result in the cloning of the ParserPath at this stage—one new ParserPath for each of the remaining viable LexerPaths. The StepParser will then continue to process these multiple grammatical hypotheses in parallel, handling the ambiguity at the syntax level.
* **Affected Files:** StepLexer.cs, StepParser.cs, GrammarLoader.cs (to parse resolution strategy), LexerPath.cs, ParserPath.cs.

### 3.4. Enhanced Unicode Support

* **Current State:** AdvancedUnicodeSupport.cs contains simplified, placeholder logic for Unicode property matching and processing, especially for characters outside the Basic Multilingual Plane (BMP).
* **Change:** Replace the placeholder logic with a full, robust implementation that properly leverages the ICU4N library for accurate Unicode handling.
* **Implementation:**
  + **Unicode Property Matching:** The UnicodePropertyMatcher.MatchesProperty method will be rewritten. The simplified switch statement and helper methods will be replaced with direct calls to the ICU4N library's functions for checking Unicode properties, categories, scripts, and blocks (e.g., UChar.HasBinaryProperty, UChar.GetIntPropertyValue). A caching mechanism will be implemented to store the results of property lookups to improve performance.
  + **Normalization:** The NormalizeIfNeeded method will be verified to correctly handle all Unicode normalization forms (NFC, NFD, NFKC, NFKD), using ICU4N's Normalizer2 class if the built-in .NET normalization proves insufficient for specific edge cases.
  + **Grapheme Cluster Boundaries:** The simplified GetGraphemeClusterBoundaries method will be replaced with an implementation that uses ICU4N's BreakIterator.GetGraphemeInstance() to accurately identify user-perceived character boundaries (e.g., for complex emoji sequences).
* **Affected Files:** AdvancedUnicodeSupport.cs, UTF8Utils.cs (to ensure it correctly decodes all codepoints passed to the Unicode support classes).

### 3.5. Lexer-Parser Interaction and Path Viability

* **Current State:** The mechanism for the parser to guide the lexer is mentioned but not formally defined.
* **Change:** Define and implement the communication protocol between the StepParser and StepLexer that allows the parser to validate the grammatical viability of potential tokens and prune invalid LexerPaths.
* **Implementation:**
  1. **Structural Link:** Finalize the data structures to link a ParserPath to its set of active LexerPaths. This will likely be a List<LexerPath> property on the ParserPath class.
  2. **Viability Query:** Before the StepLexer finalizes a token, it will perform a "viability query" against the StepParser. This query will contain information about the potential tokens from all its active, ambiguous LexerPaths (e.g., a list of TokenTypes).
  3. **Parser Validation:** The StepParser, upon receiving the query for a specific ParserPath, will check its current state to determine which of the potential token types are grammatically valid as the next token. This involves checking its parsing table or looking ahead in its production rules to see which tokens could be legally shifted.
  4. **Path Pruning:** The StepParser will respond with a list of viable TokenTypes. The StepLexer will then use this response to kill any LexerPaths that would produce a grammatically invalid token. This pruning happens *before* the token is fully formed, allowing the system to fail fast and avoid unnecessary work.
* **Affected Files:** StepParser.cs, StepLexer.cs, ParserPath.cs, LexerPath.cs.

## 4. DevelApp.StepParser Enhancements

### 4.1. GLR Parser Stability and Correctness

* **Current State:** Tests indicate potential hangs or infinite loops.
* **Change:** Refactor the core GLR stepping logic in StepParser.cs for stability.
* **Implementation:**
  + **State Merging:** Implement robust state merging in MergeParserPaths to correctly combine parser paths that have identical parsing stack signatures and lookahead states. This is crucial for managing ambiguity without exponential path explosion.
  + **Recursion:** Implement a mechanism to handle left-recursion without infinite loops. This typically involves detecting when a reduction would result in the same state and input position, and handling it gracefully within the GLR framework.
  + **Ambiguity:** Ensure that when grammatical ambiguities cause ParserPaths to split, they are processed in parallel correctly. The final CognitiveGraph should represent these ambiguities using "packed nodes".
  + **Progress Checks:** Enhance the safety limits in StepParserEngine.Parse to monitor for states that are not progressing (e.g., stuck in a loop of reductions without consuming input).
* **Affected Files:** StepParser.cs, StepParserEngine.cs.

### 4.2. Error Recovery

* **Current State:** Minimal implementation.
* **Change:** Implement error recovery strategies triggered by the termination of all LexerPaths.
* **Implementation:**
  + Error handling will be a collaborative process. When the lexer reports that all LexerPaths have terminated, the StepParserEngine will initiate recovery.
  + **Panic Mode:** The primary strategy will be to discard input tokens until a "synchronization token" (defined in the grammar, e.g., ;, }) is found. The parser will then attempt to resume from a known stable state.
  + A comprehensive error report will be generated using state from the last viable LexerPath and ParserPath(s) to give precise diagnostic information.
* **Affected Files:** StepParser.cs, StepParserEngine.cs, GrammarLoader.cs.

## 5. GrammarLoader.cs Enhancements

### 5.1. Grammar Composition, Interoperability, and Meta-Parsing

* **Current State:** The LoadBaseGrammar method is a placeholder and does not correctly distinguish between meta-parsing and language composition.
* **Change:** Implement a comprehensive system for grammar loading that separates the meta-parsing of the grammar file itself from the composition of the language being defined. This will be handled by two distinct keywords, Inherits and Extends.
* **Implementation:**
  1. **Foundational Grammar:**
     + A special, hardcoded "foundational grammar" will be built into the GrammarLoader. Its sole purpose is to parse the basic syntax of .grammar files (i.e., directives like Grammar:, Inherits:, Extends:, and rule definitions <...>::=...). This is the root of the meta-parsing chain.
  2. **Meta-Parsing with Inherits <MetaGrammarName>:**
     + **Purpose:** To specify the grammar that should be used to parse the *syntax of the current grammar file*. This allows for creating custom, extended syntaxes for defining grammars.
     + **Logic:** When GrammarLoader encounters an Inherits <MetaGrammarName> directive, it recursively loads the specified MetaGrammarName. It then uses that loaded grammar to parse the remainder of the current file. This creates a chain of parsers (e.g., MyLanguage.grammar is parsed by MyGrammarSyntax.grammar, which itself was parsed by the foundational grammar).
  3. **Language Composition with Extends <BaseGrammarName>:**
     + **Purpose:** To build a new language definition by adding to, modifying, or removing rules from an existing base language. This is for composing the actual language being parsed, not the grammar file's syntax.
     + **Logic:** After the current grammar file has been parsed (using its meta-grammar), if an Extends <BaseGrammarName> directive is present, the GrammarLoader will load the BaseGrammarName. It will then merge the rules from the current file into the base grammar using the following logic:
       - Rules in the current file **override** rules with the same name in the base grammar.
       - A new directive, %remove <RuleNameToRemove>, will be introduced to explicitly delete a rule from the base grammar.
  4. **External Grammar Importers:**
     + **Purpose:** To enable interoperability by translating grammars from other popular formats into the StepParser's GrammarDefinition format.
     + **Logic:** The GrammarLoader will use a pluggable importer system. When loading a file with a recognized extension, it will invoke the appropriate importer.
     + For standard formats (e.g., ANTLR .g4, Yacc/Bison parser files (.y), and Flex/Lex lexer files (.l)), it will invoke a specific importer (e.g., AntlrImporter, YaccImporter, FlexImporter).
     + Each importer will be a dedicated parser responsible for translating the external grammar syntax into a StepParser GrammarDefinition object.
     + Once imported, the GrammarDefinition can be used as the target for an Extends directive in another grammar file.
  5. **Documentation Mandate:**
     + The Grammar\_File\_Creation\_Guide.md **must** be updated to clearly define and provide examples for:
       - The crucial semantic difference between Inherits (specifies the parser for the file) and Extends (composes the language being defined).
       - The meta-parsing chain, starting from the foundational grammar.
       - How to use the %remove directive when extending a grammar.
       - How to import grammars from external formats like ANTLR or a Yacc/Lex pair and then use them in an Extends directive.
* **Affected Files:** GrammarLoader.cs (major rework), StepParserEngine.cs (to manage importers), and new files for each external grammar importer (e.g., AntlrImporter.cs, FlexImporter.cs). A mandatory update to Grammar\_File\_Creation\_Guide.md is also required.

### 5.2. Pluggable Semantic Action & Projection Engine

* **Current State:** CreateTokenAction, CreateSemanticAction, and CreateProjectionAction are simplified placeholders that do not execute real code.
* **Change:** Implement an extensible plugin system for handling semantic actions (=> { ... }) and projections. This decouples the parser from any single execution strategy.
* **Implementation:**
  1. **Action Handler Interface:** Define a public interface, ISemanticActionHandler, with a method like void Execute(ActionContext context).
  2. **Plugin Registration:** The StepParserEngine will expose a method, RegisterActionHandler(string name, ISemanticActionHandler handler), allowing users to register custom handlers.
  3. **GrammarLoader Role:** When parsing a grammar, GrammarLoader will not interpret action code. It will simply store the raw action text and an optional handler name within the ProductionRule or ContextProjection object. The grammar syntax could be => { HandlerName: "action text" }. If no handler name is specified, a default handler is assumed.
  4. **ActionContext Object:** An ActionContext class will be created to pass state to handlers. It will be the sole API for actions and will contain:
     + The list of matched child nodes (for accessing $1, $2, etc.).
     + The ParseContext (providing access to SymbolTable and ContextStack).
     + The CognitiveGraphBuilder for manipulating the semantic graph.
     + The ICodeLocation of the matched rule.
     + The raw action text to be executed.
  5. **StepParser Role:** During a reduction, if the matched ProductionRule has an associated action, the StepParser will look up the registered ISemanticActionHandler by name (or use the default) and invoke its Execute method, passing the fully populated ActionContext.
  6. **Default Roslyn Plugin:** The system will ship with a default, built-in handler named "csharp\_script" that uses the **Roslyn Scripting API**. This handler will execute the action text as C# code, providing the behavior described in the previous draft, but now as an optional, pluggable component rather than a hardcoded dependency.
* **Affected Files:** GrammarLoader.cs, StepParser.cs, StepParserEngine.cs, and new files for ISemanticActionHandler, ActionContext, and the default RoslynSemanticActionHandler.

## 6. CognitiveGraph Integration Enhancements

### 6.1. Efficient Node Location and Selection

* **Current State:** StepParserEngine.TryFindNodeAtLocation is a non-functional placeholder. The CognitiveGraph library lacks a built-in spatial index for efficient location-based lookups.
* **Change:** Defer to the CognitiveGraph library to provide this functionality. The StepParserEngine will be a consumer of this feature, not its implementer.
* **Implementation (StepParserEngine):**
  1. **Line/Column Mapping:** The StepParserEngine will be responsible for creating and managing a line-offset map for the source file, allowing for efficient conversion between line/column locations and byte offsets.
  2. **BuildSpatialIndex Invocation:** After a parse is complete and the CognitiveGraph is built, the StepParserEngine will call a new method on the CognitiveGraphBuilder, BuildSpatialIndex(), before finalizing the graph buffer.
  3. TryFindNodeAtLocation Implementation: This method will now:  
     a. Convert the input ICodeLocation to a byte offset using its line-offset map.  
     b. Call the new CognitiveGraph.FindNodesAt(byteOffset) method, which will use its internal interval tree to perform the lookup.  
     c. Filter the results to find the most specific (smallest) node that contains the location.
  4. **SelectFrom... Implementation:** This method will use the same FindNodesAt API and filtering logic to handle selections and ambiguities.
* **Affected Files:** StepParser.cs, StepLexer.cs. A new file for the line-offset map utility is required. *Changes to the CognitiveGraph library are specified in a separate document.*

## 7. Refactoring Operations (StepParserEngine.cs) Implementation

* **Current State:** The refactoring operations (ExtractVariable, InlineVariable, Rename, FindUsages) in StepParserEngine.cs are non-functional placeholders using hardcoded values.
* **Change:** Implement the full logic for these operations, leveraging the newly available CognitiveGraph spatial index and the SymbolTable for semantic analysis.
* **Implementation:**
  1. **Common Logic:**
     + All refactoring operations will start by using the now-functional TryFindNodeAtLocation to get the primary SymbolNode at the specified ICodeLocation. If no node is found, the operation fails gracefully.
  2. **FindUsages:**
     + This will be the foundational operation. It identifies the symbol from the target node.
     + It will then query the ParseContext.SymbolTable and the CognitiveGraph to find the symbol's declaration site and all reference sites within the correct scope.
     + It returns a List<ICodeLocation> for all found usages.
  3. **Rename:**
     + Calls FindUsages to get all locations (declaration and references) for the symbol at the given location.
     + It then generates a CodeChange for each location, replacing the old symbol name with the new one.
     + It will perform a scope check to ensure the new name does not conflict with an existing symbol in a way that would change the code's meaning (shadowing).
  4. **ExtractVariable:**
     + Identifies the boundaries of the expression from the target SymbolNode.
     + Determines the correct scope to insert the new variable declaration (e.g., the start of the current statement block).
     + Generates two CodeChanges: one to insert the new variable declaration (e.g., var newVar = [original expression];) and one to replace the original expression with the new variable's name.
  5. **InlineVariable:**
     + From the target SymbolNode (which is a variable usage), it finds the variable's declaration.
     + It checks if the variable is safe to inline (e.g., it is initialized with a constant or simple expression and is not modified).
     + If safe, it calls FindUsages to locate all references.
     + It generates CodeChanges: one to delete the original variable declaration and one replace for each usage, substituting the variable with its initializer expression.
* **Affected Files:** StepParserEngine.cs.

## 8. Testing Strategy

* **Current State:** The test suites (DevelApp.StepLexer.Tests and DevelApp.StepParser.Tests) have placeholders and incomplete tests that intentionally avoid full parsing to prevent hangs, indicating low confidence and coverage.
* **Change:** Implement a comprehensive, multi-layered testing strategy to ensure correctness, robustness, and performance across the entire system.
* **Implementation:**
  1. **Unit Tests:**
     + **Lexer:** Create focused tests for the eNFA regex engine, PCRE2 feature parsing, ambiguity splitting, ZeroCopyStringView operations, and Unicode handling.
     + **Parser:** Test GLR state merging with ambiguous grammars (e.g., "dangling else"), left-recursion handling, precedence rules, and context-sensitive rule activation.
     + **GrammarLoader:** Test parsing of valid and malformed .grammar files, correct merging for Extends, meta-parsing with Inherits, and external grammar importing.
     + **CognitiveGraph & Refactoring:** Test the interval tree index, FindNodesAt queries, and each refactoring operation (Extract, Inline, Rename, FindUsages) on various code structures.
  2. **Integration Tests:**
     + Create end-to-end tests that load complex grammars and parse corresponding source files.
     + Verify that the final CognitiveGraph is structured as expected and that semantic actions were correctly triggered and executed via the plugin engine.
  3. **Performance Benchmarks:**
     + Utilize BenchmarkDotNet to create a performance testing suite.
     + Measure and track lexer/parser throughput (tokens/sec, lines/sec).
     + Measure and track memory allocation using [MemoryDiagnoser] to validate the zero-copy architecture.
     + Benchmark the performance of CognitiveGraph queries, especially spatial index lookups.
  4. **Regression Testing:**
     + Create a dedicated suite of regression tests based on previously discovered bugs (especially parser hangs/loops) and known difficult-to-parse constructs. This suite will be run on every commit to prevent regressions.
* **Affected Files:** All .Tests projects will be significantly expanded. New performance benchmark projects may be added.

## 9. Review Status

* [x] **Section 3.1: Core Lexical Analysis Engine** - **Approved**
* [x] **Section 3.2: PCRE2 Feature Implementation** - **Approved**
* [x] **Section 3.3: Ambiguity Resolution Logic** - **Approved**
* [x] **Section 3.4: Enhanced Unicode Support** - **Approved**
* [x] **Section 3.5: Lexer-Parser Interaction** - **Approved**
* [x] **Section 4: DevelApp.StepParser Enhancements** - **Approved**
* [x] **Section 5.1: Grammar Composition** - **Approved**
* [x] **Section 5.2: Pluggable Semantic Action Engine** - **Approved**
* [x] **Section 6.1: Node Location and Selection** - **Approved**
* [x] **Section 7: Refactoring Operations Implementation** - **Approved**
* [x] **Section 8: Testing Strategy** - **Approved**

Document Status: FINAL

Reviewer:

Date: 2025-10-22