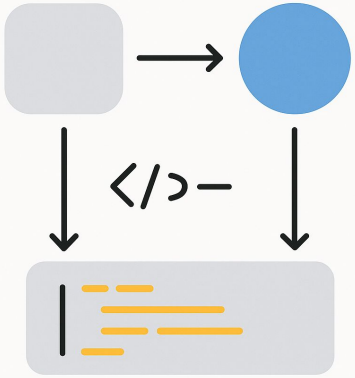
SDSLH
A Decoupled, Protocol-Driven Framework for Enhanced Rexx Syntax Highlighting Across Editors



Adrian Sutherland
REXX Symposium, Vienna

Why Syntax Highlighting Matters

Syntax highlighting is essential for modern code editors and IDEs.

It's not just aesthetic; it significantly boosts developer productivity and code quality.

Enhances code comprehension by visually structuring text.

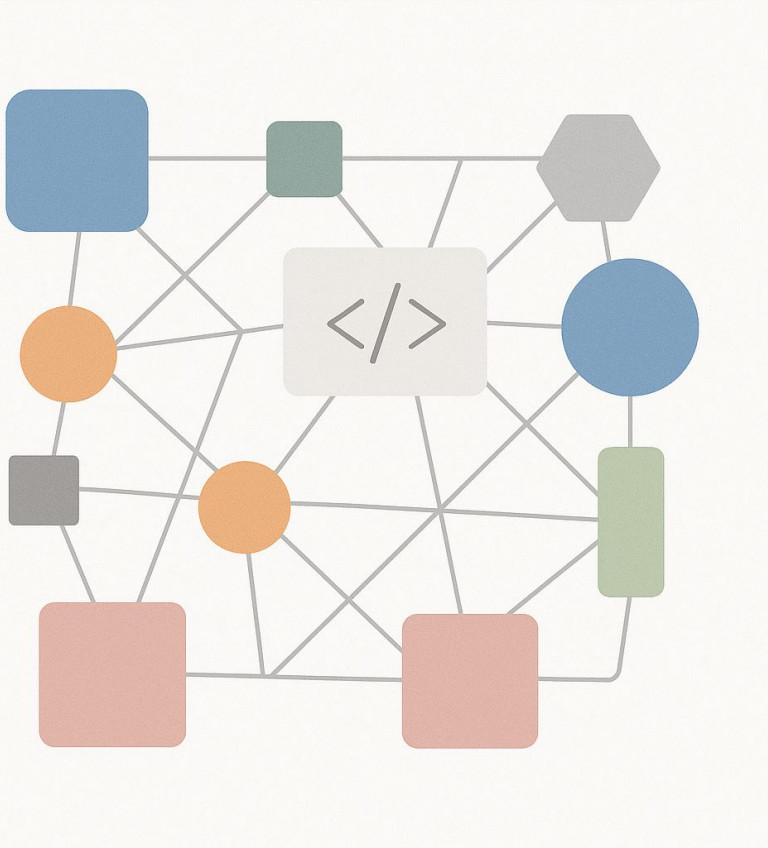
Aids faster navigation and quicker identification of code elements.

Reduces cognitive load.

Helps detect syntax errors early, improving maintainability and reducing debugging time.

Transforms plain text into a more accessible representation of code logic

The Challenge: Diverse Ecosystems



Implementing accurate, performant, and maintainable syntax highlighting is challenging.

Challenges amplified by complex grammars, numerous dialects, or domain-specific extensions.

Significant burden: reimplementing highlighting logic for each editor platform with proprietary APIs.

Many Language Engineers become disappointed that their language processor does not really support highlighting - **typically too laggy**

Rexx Ecosystem Challenges

Rexx, with its decades of evolution, exemplifies these challenges.

Multiple variants and implementations (Classic, ooRexx, NetRexx).

Diversity combined with wide array of platforms and editors creates a complex tooling ecosystem.

Existing Rexx tools often have limitations in highlighting capabilities.

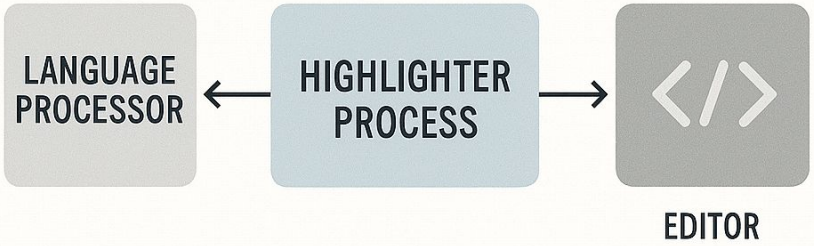
Issues include incorrect highlighting with themes, mixed-case keywords, incomplete error detection, and lack of support for includes/embedded languages.

Interpreter diversity (Regina, ooREXX, Netrexx, Regina, cREXX, BREXX) adds complexity.

Certain editors (like THE, modeled after XEDIT/KEDIT) are historically significant.

Developing distinct plugins for each editor-variant combination is a substantial overhead

Introducing SDSLH: The Decoupled Approach



Simple **DSL** **S**yntax **H**ighlighter (SDSLH) framework addresses cross-editor highlighting complexities.

Novel decoupled architecture.

Language-specific highlighter process communicates with editors via a standardised, lightweight protocol.

Aims for consistent, high-quality, efficient highlighting across multiple editors.

Offers specific applicability and benefits for the Rexx ecosystem.

Architecture Overview & Motivation

Core concept: separate complex parsing (highlighter) from editor display/interaction (editor).

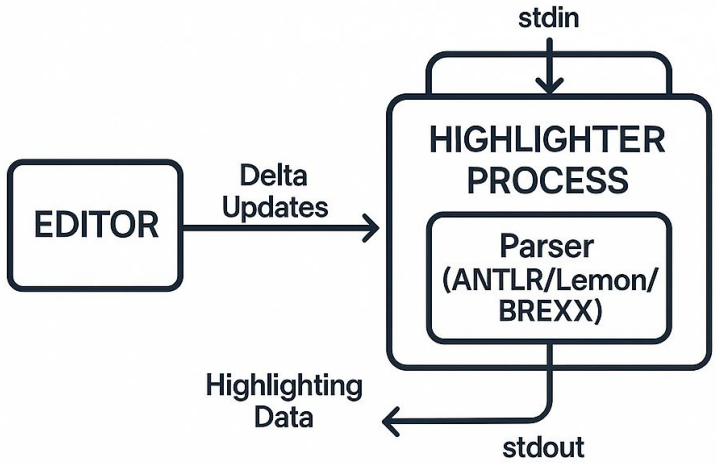
Highlighter process runs dedicated for each session/file type.

Editor (client) launches process and communicates via stdin/stdout.

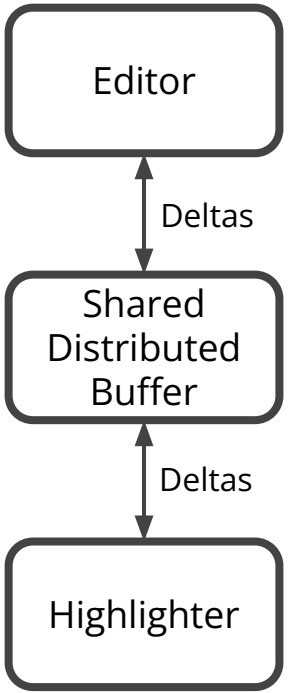
Highlighter (server) processes code and returns token info.

Motivation:

- Leverage existing parsers (Bespoke, ANTLR, Lemon, reusing existing parsers like TUTOR, Regina, BREXX).
- Simplify editor integration via a single protocol.
- Improve quality & consistency through centralized, deeper analysis.
- Efficiency via delta updates for incremental changes.



Detailed Architecture: Process Model & Buffer



Process Model: Editor launches highlighter process on file open. Communication over stdin/stdout. Editor manages lifecycle.

Shared Buffer Concept: Both editor and highlighter maintain a synchronized copy of the document buffer.

Editor sends delta updates to the highlighter.

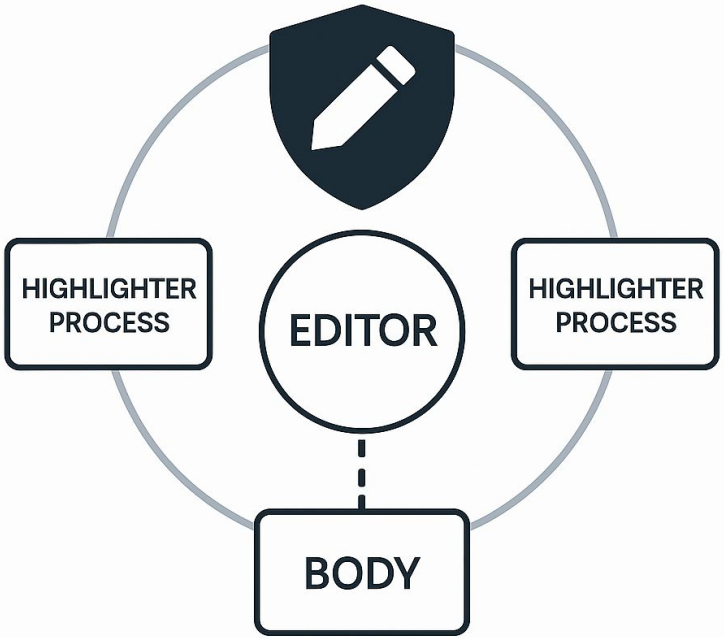
Highlighter applies deltas to stay consistent.

Synchronization using version numbers or timestamps.

Detailed Architecture: Deltas & Emergency Highlighting

- Delta Updates: Cornerstone of efficiency.
- Editor sends only incremental changes (Transactions).
  - Bundled into DELTA messages with position and content details.
  - Minimizes data transfer.
  - Difference detection (e.g., Myers Diff) for large pastes.
- Emergency Highlighting: Editor applies quick local rules immediately on edit.
  - Provides instant visual feedback.
- Replaced by accurate data from SDSLH process when received.
  - Reconciliation handles user edits during request-response cycle.

Detailed Architecture: Robustness & Generalizability



Robustness: External process failure doesn't crash the editor.

Editor is insulated and can fall back to emergency highlighting.

Contrasts with tightly integrated plugins.

Generalizability: Architecture is language-agnostic.

Protocol and editor integration patterns reusable for other languages by substituting backend.

The SDSLH Protocol: Lightweight & Text-Based

Communication governed by the SDSLH protocol.

Rationale: Simplicity for C/C++ backend implementation.

Plain text parsing using basic string manipulation.

HTTP-like structure: headers and optional body.

Well-understood and extensible format.

Message Format:

- <Command> <Resource> SDSLH/1.0
- Header: Value pairs
- Blank line separates headers from body
- Content-Length: <length> (mandatory)
- `` (payload)

SDSLH Protocol: Key Commands & Headers

Key Commands

- INIT: Establish session, send initial state.
- DELTA: Transmit incremental changes (transactions).
- HIGHLIGHT: Request/provide token data.
- ACK: Acknowledge message processing.
- ERROR: Communicate parsing or protocol errors.
- PING: Keep-alive mechanism.

Essential Headers

- Content-Length: Body size in bytes.
- Content-Type: Body format (e.g., application/rexx-tokens-v1).
- Range: Character or line range.
- Timestamp / Version: Synchronization and ordering.

Text-based trade-offs: potentially less performant than binary, but simpler C (and other languages like REXX and JavaScript) implementation.

Flexibility for future enhancements via new headers/content types.

SDSLH Protocol: Data Structures

Message body format depends on Content-Type.

Highlight Tokens

- Sequence of tokens in HIGHLIGHT responses.
- Includes type (KEYWORD, STRING_LITERAL, COMMENT, etc.), start/end position.
- Generic types for editor mapping to styles.

AST Markers

- Zero-length tokens like TREE_UP, TREE_DOWN for structural info.
- Indicate entry/exit from syntactic constructs.
- Allows editors to reconstruct AST representation, e.g. for code folding

Error Tokens

- Special ERROR token marks error location.
- Accompanied by structured error details (severity, code, message).
- Editor displays diagnostics.

Support Tool for Language Engineers

- Tool (Tree Walker) to convert an AST “back” into a Parse Tree needed for syntax highlighting

SDSLH Protocol: AST to Parse Tree tool

Constructing the CB_ParseTree

To support accurate syntax highlighting from Abstract Syntax Trees (ASTs), these functions help language engineers construct a CB_ParseTree that preserves full token order and structure.

Requirements:

- **Source Order** Tokens must follow original source order, which may differ from AST traversal
- **Completeness** All tokens (incl. whitespace and delimiters) must be present to maintain correct line/column info
- **Reorganization** Some tokens may need repositioning between subtrees (e.g., moving whitespace outside constructs)

Since ASTs often omit non-essential tokens, these helper functions manage the complexity of rebuilding a complete token stream for highlighting and analysis.

Protocol Workflow

Initialization (INIT): Editor launches process, sends initial state; highlighter initializes buffer.

Editing (DELTA): Editor captures edits as Transactions, updates local buffer, sends DELTA messages with timestamps/versions.

Emergency Highlighting: Editor applies quick local rules immediately.

Highlight Request (HIGHLIGHT): Editor requests updated highlighting for a range.

Processing: Highlighter reads DELTA, applies changes, parses affected region, generates tokens.

Highlight Response (HIGHLIGHT): Highlighter formats token data, sends response (minimal range).

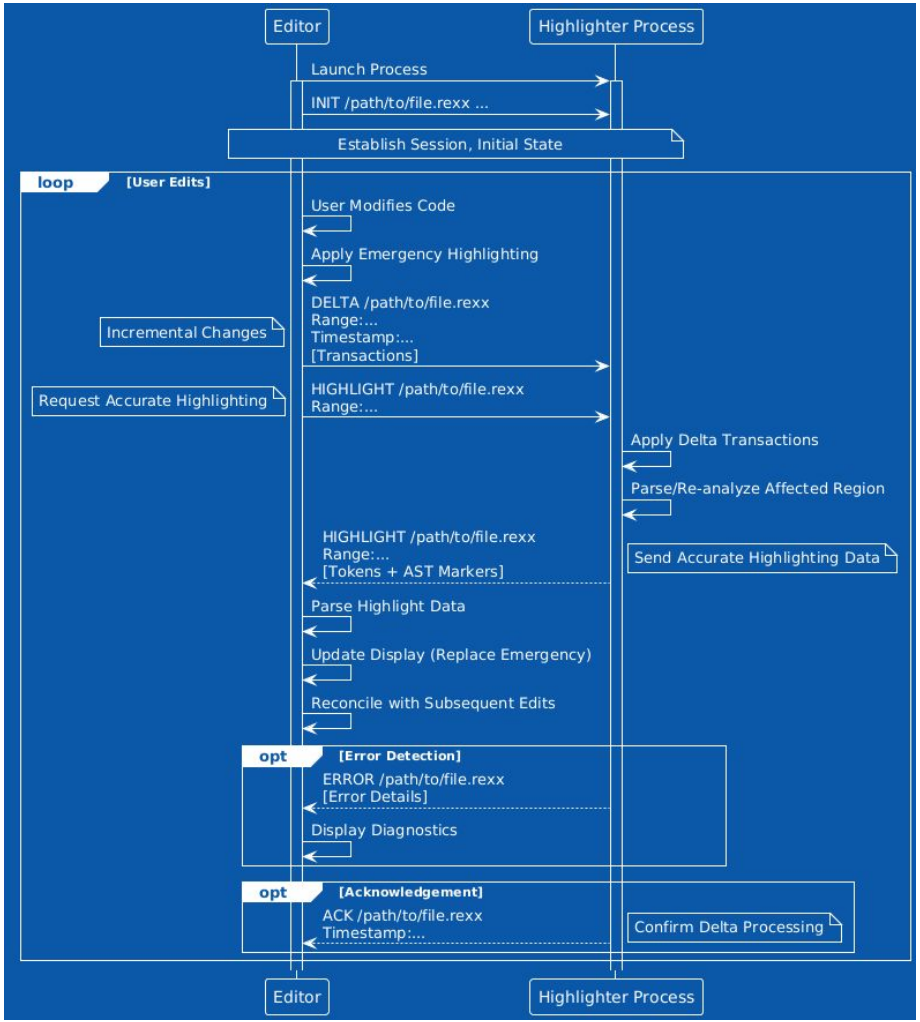
Acknowledgement (ACK): Optional ACK confirms DELTA processing.

Integration & Reconciliation: Editor reads responses, updates display, reconciles with subsequent local edits.

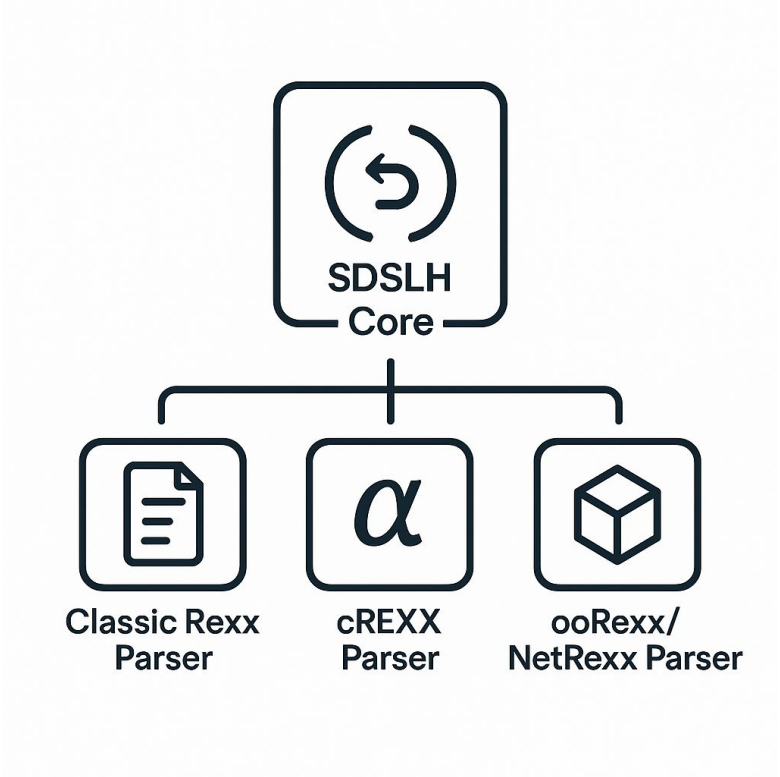
Error Handling (ERROR): Highlighter includes ERROR tokens/messages; editor displays diagnostics.

Synchronization: Timestamps/versions correlate changes and handle out-of-order messages.

Robustness: ERROR command and PING for health checks; timeouts for unresponsiveness



SDSLH for the Rexx Ecosystem: Pluggable Backends



Decoupled architecture is key to supporting Rexx diversity.

Different parser implementations can be distinct highlighter processes or modules.

Example 1: Classic Rexx: Leverage BREXX (or Regina) parser components for high-fidelity highlighting.

Example 2: cREXX: Adapt C/Lemon backend for nuances of cREXX (and RXAS)

Example 3: ooRexx/NetRexx: Reuse of existing parsers (maybe TUTOR), or distinct grammars (e.g. ANTLR).

Feasibility PoC (and working example) of BREXX reuse is a key next step for the C backend

SDSLH for the Rexx Ecosystem: Integration with THE

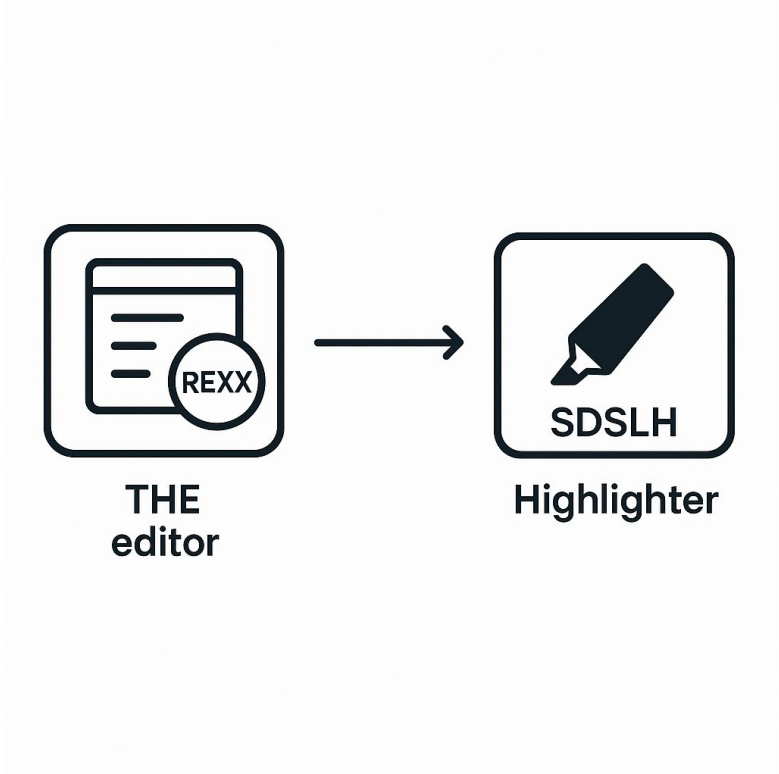
THE (The Hessling Editor) is significant in the Rexx world (XEDIT heritage, Rexx macros).

Proposed Integration: A SDSLH plugin can link against REXX (or other languages like RXAS).

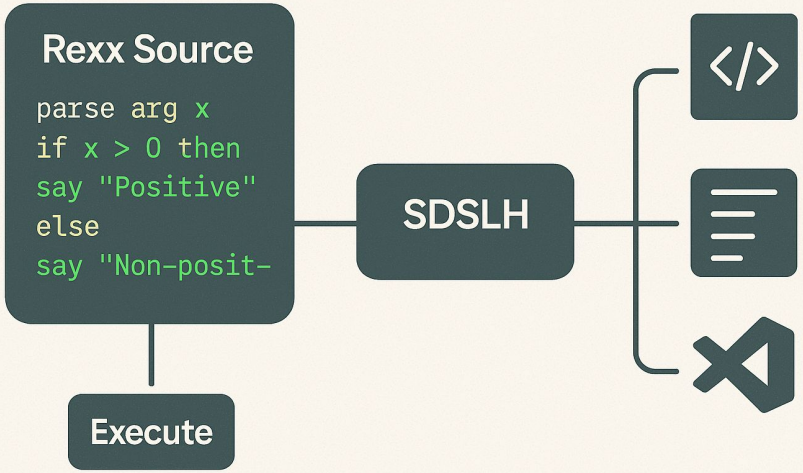
Initially a MVP, but potentially a generic THE capability.

User ecosystem also leverage THE's macro system capabilities.

Proof-of-concept with THE will be a faster to deliver demo target over Netbrains / VSCode etc.



SDSLH for the Rexx Ecosystem: Handling Rexx Nuances



Full parser backend handles Rexx's specific features accurately.

Correctly handles free-form and case-insensitivity.

Distinguishes keywords vs. variables through contextual analysis.

Identifies and highlights compound variables/stems (e.g., myStem.index.value).

Accurate tokenization of comments, strings, numbers, operators.

Identifies INTERPRET keyword and expression (though full dynamic highlighting is hard).

Leveraging REAL parsers mean syntax highlighting is “gold standard”

Implementation Strategy: Backends

Reuse

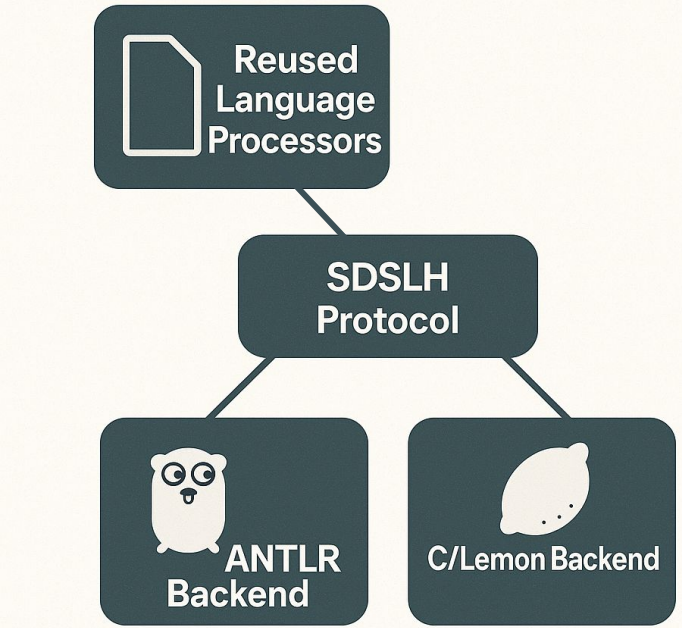
- Rationale: Leverage existing proven impentations
- Process: Take existing Parsers and integrate to the SDSLH client library. This may mean improving resyncing after error detection to allow the highlighting to continue after a syntax error

Go/ANTLR Backend

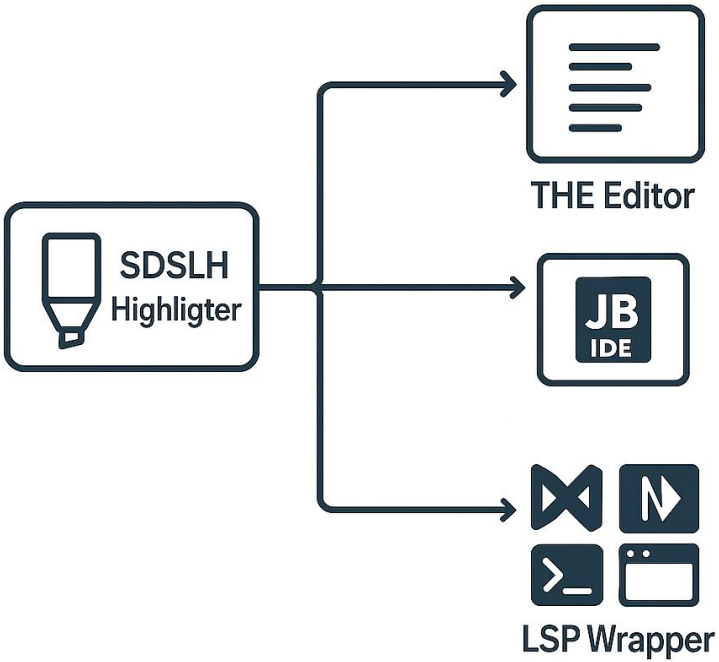
- Rationale: Performance, concurrency, excellent tooling in Go; **powerful** ANTLR parser generation.
- Process: Define ANTLR grammars, generate Go parser code, implement SDSLH protocol logic (read DELTA, parse, format/send HIGHLIGHT).

C/Lemon Backend

- Rationale: Minimalism, portability, ease of integration in C; compact Lemon LALR(1) parser; simple protocol handling avoids complex libs.
- Process: Define Lemon grammar, generate C parser, implement SDSLH protocol logic (manage I/O, parse messages, invoke parser, generate output).



Implementation Strategy: Editor Integration Approaches



Direct Protocol Support

Editor/plugin implements SDSLH client directly (e.g., via external process APIs). THE is a candidate.

JetBrains Plugin

Challenge: Conforming to JetBrains PSI/AST architecture. SDSLH token stream doesn't directly match.

Solution: Plugin manages SDSLH process, parses responses (including AST markers), reconstructs PSI-like structure. Requires careful asynchronous handling.

LSP Wrapper

Purpose: Grant compatibility with large ecosystem of LSP-capable editors (VS Code, Neovim, Sublime Text, Eclipse). Avoids bespoke plugins per editor.

Implementation: Standalone app acts as LSP server to editor, SDSLH client to highlighter. Translates LSP requests to SDSLH commands and vice-versa. Manages state synchronization.

Project Status: Work-In-Progress

SDSLH is in design and early implementation stages.

Core architecture and protocol defined.

MVP/PoC Library, and demo Parser and Editor in development.

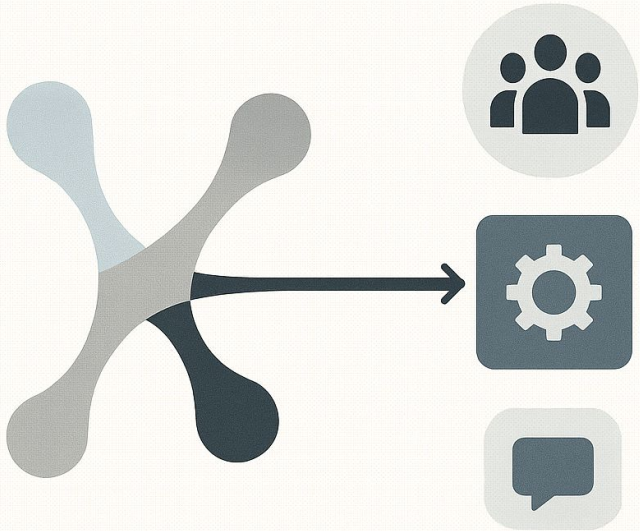
Full implementations of backends and integrations will start once MVP Library is finalised

Symposium Scope

- **Live demo milestone was missed!**
- Presentation focuses on design merits, protocol details, workflow, potential benefits for Rexx.

Presenting WIP for early feedback and collaboration.

Future Directions & Call for Collaboration



Future Enhancements

- Complete backend implementations (Go/ANTLR, C/Lemon, TUTOR).
- Develop and release editor integrations (JetBrains plugin, LSP wrapper, THE direct).
- Expand features beyond basic highlighting (code completion, linting, detailed error reporting).
- Broaden language support (additional Rexx variants, other DSLs).
- Foster community: open-source protocol and reference implementations.
- Engage with Rexx community (RexxLA) for feedback and contributions.

SDSLH design holds potential for improving Rexx tooling.

Project focus is on simple (but comprehensive) protocols, and promoting implementation and robustness.

Call for Collaboration

- Discussions on design, Rexx requirements, challenges.
- Collaboration can accelerate development and ensure effectiveness for Rexx programmers.

Q&A and Thank You!

Questions?

Adrian Sutherland

adrian@sutherlandonline.org