Scheme-langserver: Treat Scheme Code Editing as the First-Class Concern

WANG, Zheng
ufo5260987423@163.com
Institute of Scientific and Technical Information of China
Haidian District, Beijing, P. R. China

Abstract

Programmers have become increasingly reliant on modern IDEs and editors due to their support for advanced language features such as auto-completion, go-to-definition, and reference, which significantly reduce the mental burden during programming. However, existing implementations for Scheme language programming predominantly focus on REPL-driven development, treating language features as secondary byproducts of code execution. To address this gap, Scheme-langserver, an LSP (Language Server Protocol) implementation targeting R6RS-based Scheme programmers, treats code editing as a first-class concern. By leveraging abstract interpretation, basic partial evaluation, and lightweight type inference, it provides robust tooling support comparable to mainstream IDEs while preserving Scheme's functional paradigm. The main purpose of this paper is to explain how Scheme-langserver works, its internal design, and how it differs from other tools and methods.

1 Introduction

1.1 Background of Scheme Code Editing

The Scheme programming language has been shaped by a series of standardized reports, notably R5RS (1998) [1], R6RS (2007) [6], and R7RS (2013) [5], which emphasize minimalism and extensibility. These standards prioritize a small core syntax (e.g., 23 syntactic forms in R5RS) while enabling macro systems for domain-specific extensions. This design philosophy aligns with Scheme's historical role in computer science education. For instance, the seminal textbook Structure and Interpretation of Computer Programs (SICP) leverages R5RS Scheme's simplicity to teach recursion, higher-order functions, and meta-circular interpreters through immediate REPL feedback. The language's REPL-driven workflow allows students to iteratively test code fragments (e.g., (map square '(1 2 3))), fostering an empirical understanding of abstraction layers from lambda calculus to compiler design.

ELS'25, May 12–13 2025, Zurich, Switzerland

https://doi.org/10.5281/zenodo.15384882

Scheme development traditionally revolves around the Read-Eval-Print Loop (REPL), where code is incrementally evaluated and refined, or it's named exploratory programming. Developers are used to loading files dynamically, enabling runtime updates without restarting the environment, quickly executing a small piece of complete code, and concerning themselves with what the execution says. This workflow is profoundly enhanced by Emacs and its Geiser mode [3], which seamlessly bridges code editing and REPL interaction. Geiser enables bidirectional communication with a live Scheme process, allowing developers to send S-expressions directly from the editor buffer to the REPL for instant evaluation. It provides namespace-aware autocompletion, symbol definition jumps, and inline macro expansion visualization, all while preserving the REPL's dynamic context. By integrating keystroke-driven workflows with structural editing tools like Paredit, Emacs transforms the REPL from a standalone prompt into an interactive programming workspace, which aligns perfectly with Scheme's ethos of linguistic abstraction and exploratory coding.

1.2 Problem & Challenge

REPL philosophy requires complete and syntactically valid expressions for evaluation. This design assumes programmers work with self-contained code snippets, where each input is a logically atomic unit. However, modern code editing inherently involves incomplete or transitional states, such as half-written functions, dangling parentheses, or placeholder variables. For example, a developer might type `(define (factorial n)` but pause to refine the base case, leaving the expression unresolved. The REPL cannot process such fragments, forcing programmers to either manually construct temporary completions (e.g., adding a dummy `0)`) to test partial logic, which pollutes the REPL's namespace, or frequently switch contexts between editing and evaluation, disrupting mind flow.

Many tools attempt to bridge this gap through hybrid approaches. Emacs Geiser mode leverages the REPL's runtime state by capturing REPL's stack trace. With approximately "live" editing, it's able to highlight the limited diagnostic issues inline. Tree-sitter extracts all identifiers from the buffer ignoring scoping rules, and offers them as completion candidates. This works for trivial cases but risks mismatches. Especially when comparing to Java/Python's mature ecosystems, it seems the fundamental tension in the REPL-centric development model and the static-analysis-driven workflows

haven't seriously treated programmers' modern IDE senses yet.

The core challenge to implement Java/Python-like IDE support for Scheme programming is to overcome the limits of general static analysis. For example, Scheme's hygienic macro system allows users to redefine syntax for control flow, variable declarations, and even package management. And Geiser's Chez Scheme support only directly involves dynamic dependencies and imports corresponding symbols by calling (environment-symbols (interaction-environment)). It ignores incomplete programs because they can't be executed and it truely gets some information involving macro expansion. This is also the most interesting thing in Scheme, and if we imagine a state-of-the-art IDE having full support for Scheme, we may notice macro programming is editing both of IDE behavior and the program itself.

1.3 Scheme-langserver as A Solution

Scheme-langserver[1] is a Language Server Protocol (LSP) [4] implementation designed to bring modern IDE features, such as autocompletion, go-to-definition, and type-inference-based diagnostics(Fig. 1) to R6RS-compliant Scheme programmers. Further more, unlike traditional REPL-driven tools like Geiser, which rely on runtime evaluation, Scheme-langserver adopts a hybrid approach combining static analysis, partial evaluation, and abstract interpretation to approximate Scheme's dynamic semantics without requiring full code execution, and to try to bridge the gap among every "live" code editing by maintaining mind flow with static analysis information over and over again. This is actually to treat code editing instead of REPL as the first-class concern by avoiding distraction caused by further information requirements.

For example, giving a code file Listing 1 having dependency on (scheme-langserver util natural-order-compare) in Fig. 1, building upon general experience, any code editing like switching (rnrs) between R6RS and R7RS, deleting a character, or hovering specific words, may raise various LSP requests and Scheme-langserver is supposed to over the limit of REPL and to respond accrodingly. These responses may require resolving global identifiers via references in different files(e.g., line 8 refers natural-order-compare in Fig. 1), completing local identifiers (e.g., line 7 claims parameter string-list), updating program dependencies(e.g., line 2 switches (rnrs) between R6RS [6] and R7RS), hovering type inference diagnostic information as in Fig. 1, or even canceling working task by involving peephole optimization, to skip current request and work on next request.

Listing 1: Editing May Involve Different Scheme-langserver Behaviors through LSP

```
1 (library (scheme-langserver example)
2   (import
3     (rnrs)
4     (scheme-langserver util natural-order-compare)
5   (export sort-in-natural-order)
6   (define sort-in-natural-order
7     (lambda (string-list)
8       (sort natural-order-compare string-list))))
```

In the rest of this paper, we present a systematic analysis of how Scheme-langserver processes incomplete source code to deliver actionable editor feedback. Section 2 details the server's architecture, focusing on its asynchronous request-handling for minimizing latency during client-server communication. Section 3 introduces a data-flow graph model that unifies three core techniques, abstract interpretation for approximating runtime state, partial evaluation for resolving static code fragments, and heuristic type inference for programming experience improvement, while critically examining their limitations, particularly in handling hygienic macros and mutable states. Finally, Section 4 synthesizes our findings, emphasizes the differences in subsequent development among Scheme-langserver and its counterparts, and calls for community collaboration to address open challenges in formalizing Scheme's semantics for static tooling.

2 Engineering Affairs

2.1 Architecture & LSP Lifecycle

As illustrated in Fig. 2, Scheme-langserver's established an architecture for processing LSP requests spans 3 core domains: virtual file system, request queue optimization, and indexer/analyzer integration. The workflow adheres to the LSP specification [4], wherein the server operates as an independent process communicating with editors via JSON-RPC over standard I/O. The request lifecycle follows red arrows in Fig. 2: first, raw JSON-RPC payloads are parsed into structured request objects, bypassing editor-specific I/O limitations; second, requests enter a FIFO queue of suspendable tasks; third, initialize or resolve in-memory buffers asynchronously, via a unified virtual file system abstraction;finally, trigger response.

This section elaborates on above components excluding the indexer/analyzer, which is analyzed in Section 3.

2.2 Virtual File System

The virtual file system (VFS) maintains a real-time representation of the project's lexical and syntactic state, instead of the complete semantic state, or the environment, required by the REPL runtime. During initialization (Fig. 2), the VFS constructs a hierarchical file-node graph rooted at the project directory, with each node mapping to a document that synchronizes source code. Upon receiving client updates (e.g., textDocument/didChange), the VFS invalidates affected index-node subtrees and regenerates them using Chez Scheme's default reader[2] coupled with positional annotation (line/column spans, parent-child dependencies). Combined with a fault-tolerant mechanism, the VFS isolates syntactic or lexical discontinuities into independent

[1] https://github.com/ufo5260987423/scheme-langserver, https://codeberg.com/ufo5260987423/scheme-langserver, https://gitee.com/ufo5260987423/scheme-langserver

util > natural-order-compare.sls > ...

```
1  (library (scheme-langserver util natural-order
2      (export natural-order-compare)
3      (import (rnrs))
4
5  (define natural-order-compare
6      (case-lambda
7          [(string-a string-b) (natural-order-co
8          [(string-a string-b index-a index-b)
9              (let ([length-a (string-length str
10                 [length-b (string-length s
11             (if (or (>= index-a length-a)
12                 (>= index-b length-b))
13             (<= length-a length-b)
14             (let ([char-a (string-ref string-a index-a)]
15                 [char-b (string-ref string-b index-b)])
16                 (if (char=? char-a char-b)
17                     (natural-order-compare string-a string-b (+ 1 index-a) (+ 1 index-b))
18                     (char<? char-a char-b)))))]))
19 )
```

string-a

Type Inference

something?

[identifier-reference string?]

Definition

```
(case-lambda
        [(string-a string-b) (natural-order-compare string-a string-b 0 0)]
        [(string-a string-b index-a index-b)
```

Figure 1: Scheme-langserver: Hover to Display Type Inference and Definition.

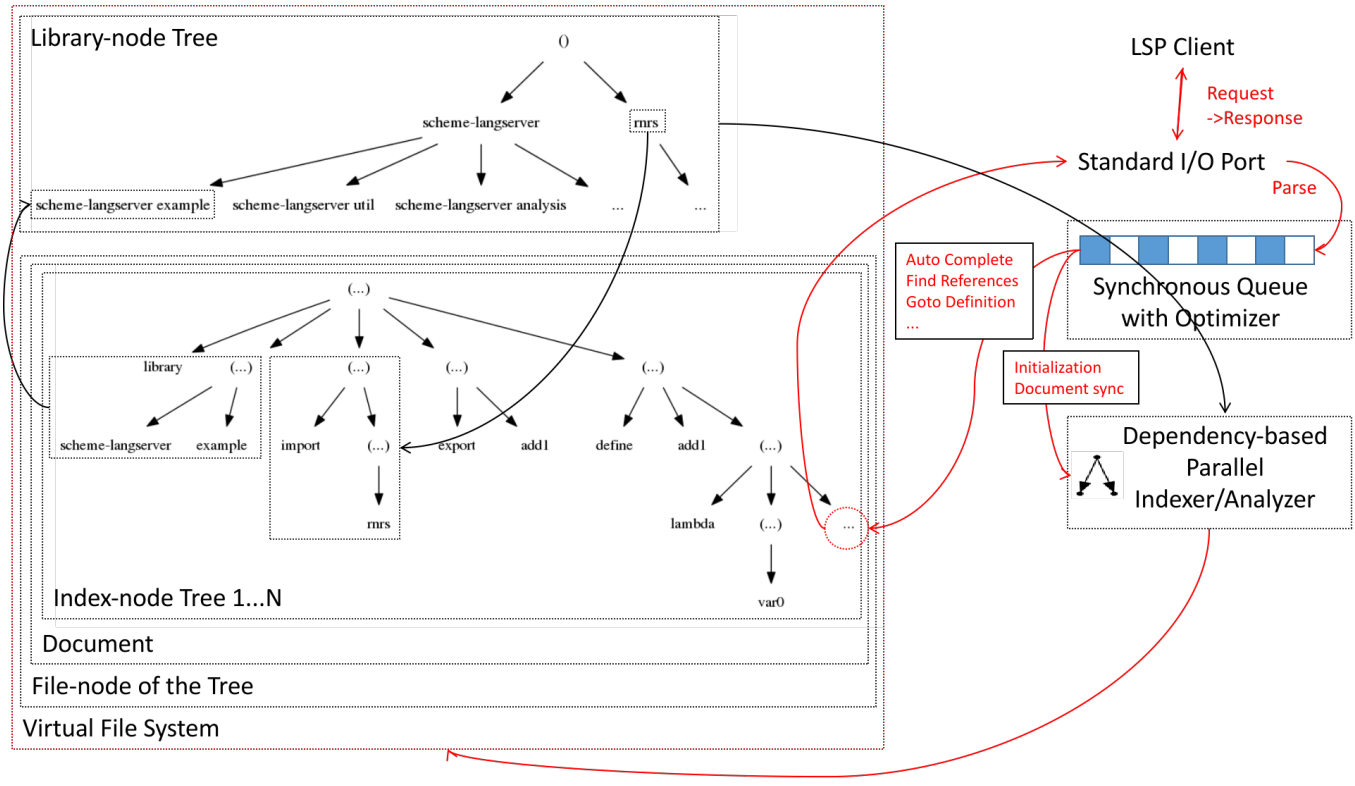

Figure 2: How Scheme-langserver Responds LSP Requests

AST fragments using parent-child dependency boundaries and prevents corruption propagation. As detailed on the left-hand side of Fig. 2, these index-node fragments interconnect via library-node dependencies, forming a cross-file semantic graph.

Outside of the above graph, the key to understanding is LSP's [4] textual delta model: clients transmit character-level edits rather than syntactic unit modifications. For Scheme this means while modern editors commonly enforce parenthesis balancing, LSP does not guarantee atomic S-expression updates. Consequently, a critical constraint arises from partial index-node updates and must instead reconstruct the entire sandbox with necessary fault tolerance, which is a trade-off ensuring syntactic validity at the cost of latency. Fortunately, the textDocument/didChange request doesn't interleave other requests requiring semantic completeness frequently, and additional mechanisms like request queue optimization make such latency tolerable.

The VFS's novelty lies in its bidirectional binding of incomplete code states to LSP workflows. As visualized in Fig. 2's red dashed elements, it enforces rule-based associations between raw text edits (red arrows) and analyzed AST fragments (black arrows), enabling context-aware diagnostics even during transitional code states.

2.3 Request Queue Optimization & Threaded mechanism

Scheme-langserver facilitates a request queue to decouple the request lifecycle into two distinct phases: scheduling and execution. As for scheduling, it applies peephole optimization to asynchronously analyze API requests. More specifically, instead of efficiency or syntax, the request optimization now focuses on cancelation, which means unresponsive requests can be canceled. Here within queue context, Scheme-langserver wraps each request with a suspendable task in order to check whether they're canceled during processing.

Upon dequeuing a request, a dedicated worker thread assumes responsibility for the VFS operations, which including document synchronization, dependency resolution, in-memory buffer management and making response. LSP-specific semantics (e.g., textDocument/didChange) invoke concurrent primitives such as thread pools for parallel I/O and batched AST traversals, enabling non-blocking responsiveness during resource-intensive tasks like cross-module symbol lookup.

This architecture ensures atomicity for critical paths (e.g., updating global symbol tables) while permitting asynchronous execution of auxiliary operations (detailed in Section 3).

2.4 Testing and Accident Reproduction

The testing infrastructure of Scheme-langserver is structured to validate both control and data flows, as visualized by the red and black data pathways in Fig. 2. The system currently maintains 154 comprehensive test cases spanning abstract syntax tree and thread safe structure manipulations, JSON-RPC serialization/deserialization, virtual file system operations and dependency resolution. This granular test coverage unprecedented among Scheme language LSP implementations, ensures behavioral consistency across edge cases like partial S-expression evaluations and concurrent document updates.

A pivotal innovation lies in its deterministic failure reproduction mechanism. By logging time-stamped request sequences alongside project state snapshots, developers can replay failure scenarios via log-driven execution: By re-injecting captured client requests into a server instance and, programmers can interrupting execution at specific and freeze external dependencies (e.g., filesystem calls) to maintain causal consistency. This capability not only accelerates debugging but also lowers the barrier for community contributions—users can submit reproducible test cases as pull requests, effectively crowdsourcing quality assurance.

3 Analyze Incomplete Source Code

3.1 Framework

Incomplete code introduces Scheme-langserver a halting problem, as arbitrary transient editor states may defy rigorous formal analysis under Scheme's dynamic semantics, preventing the system from explicitly determining termination or endless continuation. More specifically, deterministic execution requires a constrained framework as in Fig. 3: First, LSP initialization request triggers a dependency matrix to partition file-nodes into isolated batches; Second within each batch, files maintain mutual independence; Third, the analyzer sequentially processes batches while importing dependencies exclusively from prior completed batches; Fourth, within contexts explicitly bounded by preceding analysis phases, lexical bindings (e.g., lambda parameters, let-bound variables) become resolvable; Finally, type inference proceeds with well-defined semantic boundaries.

This section discusses how the indexer/analyzer degrades the halting problem into a solvable problem by addressing three Scheme-specific challenges with the above framework, including contextual ambiguity in library resolution, heuristic pattern matching for incomplete code, and indecisive type inference. Furthermore, this section will also discuss how to process self-defined macros.

3.2 Dependency Analysis

Unlike Python or Java, Scheme's R6RS standard specifies a runtime-dependent library mechanism, which means `library` and `import` in Listing 1 don't certainly perform library dependencies under code editing scenario and detailed analysis may cause computational complexity. To mitigate this, Scheme-langserver imposes a static import graph assumption: it extracts R6RS library identifiers and dependencies by analyzing only the outermost S-expressions (e.g., `(library ...)` or `(import ...)` forms) during initialization. So that the framework in Fig. 3, its parallelization efficiency hinges on partitioning the import such dependency graph into maximally independent batches, a problem reducible to finding the minimum number of feedback vertex sets (FVS) to break cycles. Since FVS computation is NP-hard, Scheme-langserver adopts a linear-time heuristic algorithm:

- Degree-1 Pruning: Iteratively removes vertexes with in/out-degree less than 1, as they cannot participate in cycles.

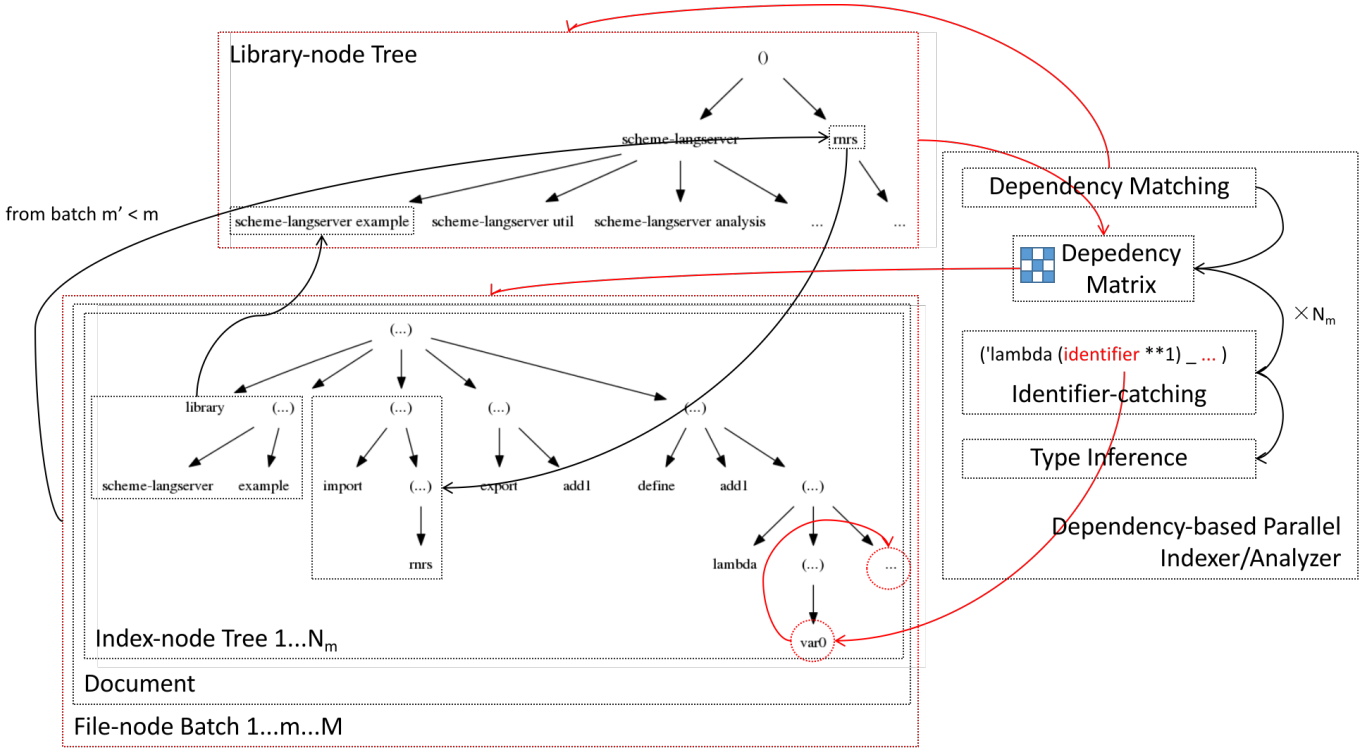

Figure 3: How Scheme-langserver Indexes/Analyzes Parallelly.

- Virtual Super Vertex Aggregation: Groups remaining cyclic sub-graphs into synthetic supernodes, treating them as atomic units during batch allocation.

Considering cyclic dependency caused by code editing where module A imports B, B imports C, and C imports A, the above algorithm would first aggregate A, B, and C into a super vertex and then process the super vertex sequentially after all acyclic dependencies. While static import graph assumption is adequate for such typical editing scenarios, where incremental changes rarely introduce deep cycles and the algorithm sacrifices optimal parallelism (super vertexes are processed serially) to avoid combinatorial explosion, a more serious scenario is the cycles caused by non-outermost S-expressions editing that may intentionally entangles codebases.

Exampling with SRFI-103, it allows programmers to locate files containing libraries and to import them in runtime. This flexibility introduces runtime-dependent symbol resolution, where identical code files may yield divergent semantic interpretations based on evaluation order or macro-generated bindings. But taking a view of the halting problem, it indicates a limitation shared with LSP implementations for Scheme language, for that dynamic libraries may cause premature termination leading to failure of binding structure, or for careless handling may cause endless recursion. The latter one is much more unacceptable to Scheme-langserver and involves abstract interpretation and partial evaluation.

3.3 Abstract Interpretation & Partial Evaluation

Most LSP requests necessitate Scheme-langserver to establish precise binding structures, where abstract interpretation and partial evaluation serve as foundational mechanisms for preventing infinite recursion during semantic analysis. As codified in Table 1's mapping between R6RS semantics in the abstract domain and R6RS declaration patterns for partial evaluation, the analysis engine employs a critical invariant: When runtime resolution with fixed dependencies explicitly assigns specific semantics to a symbol per Table 1's schema - irrespective of nominal congruence with R6RS standard identifiers (e.g., `(import (rename (rnrs)(define define1)))`) or syntactic completeness of outer S-expressions - the system guarantees partial evaluation execution without calling to analyzer itself, while bypassing full macro-expansion. This strategy directly captures semantic claims about raw S-expressions, always an index-node in VFS, through concrete domain projections, effectively decoupling syntactic analysis from dynamic evaluation phases.

Besides, its efficiency is decided mainly by a two-phase process:

Table 1: Semantics in Abstract Domain Trigger Identifier-capturing in Concrete Domain

Semantic Name in R6RS Standard	Partial Evaluation Template
case-lambda	(case-lambda clause ...)
define	(define var expr)
define	(define var)
define	(define (var0 var1 ...) body1 body2 ...)
define	(define (var0 . varr) body1 body2 ...)
define	(define (var0 var1 var2 ... . varr) body1 body2 ...)
define-condition-type	(define-condition-type name parent constructor pred field ...)
define-enumeration	(define-enumeration name (symbol ...) constructor)
define-ftype	(define-ftype ftype-name ftype)
define-ftype	(define-ftype (ftype-name ftype) ...)
define-property	(define-property id key expr)
define-record	(define-record name (fld1 ...) ((fld2 init) ...) (opt ...))
define-record	(define-record name parent (fld1 ...) ((fld2 init) ...) (opt ...))
define-record-type	(define-record-type record-name clause ...)
define-record-type	(define-record-type (record-name constructor pred) clause ...)
define-structure	(define-structure (name id1 ...) ((id2 expr) ...))
define-syntax	(define-syntax keyword expr)
define-top-level-syntax	(define-top-level-syntax symbol obj)
define-top-level-syntax	(define-top-level-syntax symbol obj env)
define-top-level-value	(define-top-level-value symbol obj)
define-top-level-value	(define-top-level-value symbol obj env)
fluid-let	(fluid-let ((var expr) ...) body1 body2 ...)
fluid-let-syntax	(fluid-let-syntax ((keyword expr) ...) form1 form2 ...)
identifier-syntax	(identifier-syntax tmpl)
identifier-syntax	(identifier-syntax (id1 tmpl1) ((set! id2 e2) tmpl2))
lambda	(lambda formals body1 body2 ...)
let	(let ((var expr) ...) body1 body2 ...)
let	(let name ((var expr) ...) body1 body2 ...)
let*	(let* ((var expr) ...) body1 body2 ...)
let*-values	(let*-values ((formals expr) ...) body1 body2 ...)
let-syntax	(let-syntax ((keyword expr) ...) form1 form2 ...)
let-values	(let-values ((formals expr) ...) body1 body2 ...)
letrec	(letrec ((var expr) ...) body1 body2 ...)
letrec*	(letrec* ((var expr) ...) body1 body2 ...)
letrec-syntax	(letrec-syntax ((keyword expr) ...) form1 form2 ...)
syntax	(syntax template)
syntax-case	(syntax-case expr (literal ...) clause ...)
syntax-rules	(syntax-rules (literal ...) clause ...)
with-syntax	(with-syntax ((pattern expr) ...) body1 body2 ...)

- AST traversal with index-nodes: Resolves identifiers using innermost lexical scope (e.g., let-bound shadows top-level-bound).
- Cross-context validation backtrack ancestor index-nodes or even dependency code documents: Validates the route and construct an evidence chain against dependencies to reject invalid references (e.g., accessing `var0` outside its `lambda` must fail in Listing 1).

3.4 Type Inference

The addition of type inference in Scheme-langserver began with observations of Typed Racket [8], in which additional type annotations truely explain how programmers thought. However, annotating as in Listing 2 line 1 `(inst cons Symbol Integer)` may deprive programmers' pleasure in Scheme language's flexibility. We wonder whether Scheme language could conserve line 2 type information without annotations.

Listing 2: Typed Racket REPL: Such Cases Deprive Programmers' Pleasure in Scheme Language's Flexibility

```
1 > (map (inst cons Symbol Integer) '(a b c d) '(1 2
        3 4))
2 - : (Listof (Pairof Symbol Integer))
3 '((a . 1) (b . 2) (c . 3) (d . 4))
```

An idea is to leverage semantic-driven type hints derived from R6RS [6] to aid code comprehension. Considering another example in Listing 3, procedure + can accept its parameter fuzzy as number? represented with the same predictor. In order to formalize this kind of information, Scheme-langserver annotates standard procedures/syntaxes with a hand-made DSL (Domain Specific Language) like (number? <- (number? ...)) for +, in which <- indicates a function type, the left is the return type, the right is the parameter types and "..." is a shortcut omitting repeated annotations. So, combined with type variables denoting index-nodes and such annotations, the system constructs a constraint graph with abstract interpretation and partial evaluation.

LSP hover and auto-completion requests execute type inference through constraint solving bound to specific index-nodes' type variables. Within the Hindley-Milner framework [7], this manifests as a triangular substitution mechanism that achieves type unification while propagating contextual type information bidirectionally across S-expressions. Notably, this process necessitates interprocedural analysis integration as type constraints propagate beyond local lexical boundaries through identifier caller and callee. This hybrid approach ensures principal typing properties are preserved while accommodating Scheme's dynamic evaluation characteristics.

Listing 3: Parameter fuzzy has Type number?

```
1 (lambda (fuzzy) (+ 1 fuzzy))
```

Of course, this system can't be that powerful like in many other main stream typed languages, and following are some special tips:

- Conservative Assumptions: Type inference always processes executable code, no matter what code editing is performing. This means type inference in Scheme-langserver always plays an assistance role by giving useful information instead of restrictions. Especially when it completes identifiers, type-matching ranks completion items higher instead of filtering them out. Also such types can't diagnose any code.
- No Runtime Validation: Types reflect static code structure, ignoring dynamic mutations (e.g., set!).
- Macro Blind Spots: Unexpanded macros are treated as syntactic literals, deferring type resolution to runtime.

3.5 Macro Analysis

Scheme-langserver intentionally avoids built-in analysis of user-defined macros due to the inherent challenges of reconciling Scheme's hygienic macro system with static tooling expectations. Hygienic macros, while powerful, introduce semantic unpredictability at edit-time, macro expansions may generate context-sensitive identifiers (e.g., via gensym) or redefine control flow in ways that defy static inference without full evaluation. For example, a macro like (define-syntax loop (syntax-rules ()((_)(loop)))...) could create unbounded recursion during expansion, stalling analysis.

To balance flexibility and performance, the system currently adopts manual rule paradigm. By defining custom pattern-matching templates and registering on abstract interpretation, programmers can easily specify whether to treat macros as syntactic literals, identifier declaration or type annotation.

4 Conclusion

Scheme-langserver represents a pragmatic attempt to reconcile Scheme's dynamic, macro-centric philosophy with the static tooling expectations of modern development workflows. While its current architecture achieves usable code completion and cross-module navigation for R6RS-compliant code, significant opportunities remain to align the tool more closely with Scheme's full expressive power. Our roadmap prioritizes 2 interconnected goals:

- User-Defined Macro Rule Integration: A forthcoming macro transformer interface will allow programmers to declaratively specify how custom macros should be interpreted during static analysis. For instance, users could define expansion templates for a (for-loop (i 0 10) ...) macro, enabling identifier resolution within loop bodies without full evaluation. Besides this, the system will also enable macro expansion analysis to automatically catch identifier declarations.
- Backward Compatibility and Maintenance: All enhancements will preserve existing capabilities, including parallel AST traversal, heuristic type inference, and virtual file system optimizations.

We must emphasize that Scheme-langserver exhibits unique architectural virtues warranting deep community engagement to unlock its full potential, as it profoundly prioritizes modern programming assistance features' impact on developer experience through meticulously engineered and modularized code infrastructure compared to alternatives like Geiser, Scheme-lsp-server, and Racket-langserver, whose implementations lack equivalent systematic integration of real-time semantic validation, version-controlled dependency tracking, and IDE-grade code intelligence services that Scheme-langserver's ordered abstraction layers inherently enable for future extensibility. The Scheme community's participation is critical to realizing this vision. We invite contributors to test the system with real-world code bases and issue bugs and codify expansion rules for common libraries (e.g., SRFIs).

By bridging Scheme's REPL-driven heritage with modern tooling paradigms, scheme-langserver aims to empower developers without compromising the language's ethos of minimalism meets expressivity. This effort underscores a broader truth: even in an era dominated by static typing, dynamically typed languages can thrive through adaptive tooling and community-driven innovation.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, and M. Wand. Revised5 report on the algorithmic language scheme. Higher-Order and Symbolic Computation, 11(1), 1998.
[2] R. Kent Dybvig. The development of chez scheme. In John H. Reppy and Julia Lawall, editors, Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming,

WANG, Zheng„

ICFP 2006, Portland, Oregon, USA, September 16-21, 2006, pages 1–12. ACM, 2006.

[3] jao. Emacs geiser, 2023. URL https://gitlab.com/emacs-geiser. [Online; accessed 26-May-2023].

[4] Microsoft. Language server protocol, 2023. URL https://microsoft.github.io/language-server-protocol/. [Online; accessed 26-May-2023].

[5] Alex Shinn, John Cowan, Arthur A Gleckler, et al. Revised 7 report on the algorithmic language scheme. Scheme Language Steering Committee, Rep. R7RS, 2013.

[6] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robert Bruce Findler, and Jacob Matthews. Revised6 Report on the Algorithmic Language Scheme. Cambridge University Press, 2010. ISBN 978-0-521-19399-3.

[7] Martin Sulzmann. A general type inference framework for hindley/milner style systems. In Herbert Kuchen and Kazunori Ueda, editors, Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings, volume 2024 of Lecture Notes in Computer Science, pages 248–263. Springer, 2001. doi: 10.1007/3-540-44716-4_16. URL https://doi.org/10.1007/3-540-44716-4_16.

[8] Sam Tobin-Hochstadt. Tutorial: Typed racket. In Christian Queinnec and Manuel Serrano, editors, Proceedings of ELS 2013 - 6th European Lisp Symposium, Madrid, Spain, June 3-4, 2013, page 26. ELSAA, 2013. URL https://european-lisp-symposium.org/static/proceedings/2013.pdf#page=32.