Slovo: A Typed Structural Language for Tool-Visible Native Programs

ⰔⰎⰑⰂⰑ

Sanjin Gumbarevic
hermeticum_lab@protonmail.com

Publication release: `1.0.0-beta.1`

Technical behavior baseline: language surface through `1.0.0-beta`; tooling and install workflow through `1.0.0-beta.1`

Date: 2026-05-22

Evidence source: paired local Slovo/Glagol monorepo verification and benchmark reruns from a local checkout; beta.1 release-gate verification from the public monorepo

Maturity: beta

Abstract

Slovo (ⰔⰎⰑⰂⰑ) is a typed structural programming language whose source form is already a tree. It borrows the visual regularity of Lisp-family syntax, but it is not a Lisp dialect. Slovo is designed around a statically checked source tree, canonical formatting, explicit types, explicit failure through `option` and `result`, lexical `unsafe`, and native compilation through the Glagol compiler to LLVM IR and hosted executables.

The current publication release, `1.0.0-beta.1`, keeps the first real general-purpose beta language baseline from `1.0.0-beta` and records the first post-beta tooling/install hardening update. The beta baseline includes the completed `u32` / `u64` unsigned scope, the staged stdlib breadth that makes ordinary command-line programs practical, and the current nine-kernel benchmark suite. This paper records the current beta technical state, the difference between Slovo and Lisp-family languages, the benchmark methodology, the beta.1 tooling update, and the remaining path from beta to stable.

1. Scope

This document is a technical state paper for the current beta baseline. It summarizes the behavior represented by the paired local Slovo and Glagol workspaces, with `1.0.0-beta` as the current language-surface baseline and `1.0.0-beta.1` as the current publication/tooling baseline.

The support rule remains strict:

- a feature is supported only when Slovo specifies it
- Glagol must parse it, lower it, type-check it, and either emit behavior or reject it with a structured diagnostic
- formatter behavior, examples, tests, release notes, and review documents must agree
- partial parser recognition or speculative examples do not count as support

Historical `exp-*` releases remain experimental alpha maturity. The current publication accompanies `1.0.0-beta.1`.

2. Design Thesis

The name Slovo is rendered in Glagolitic as ⰔⰎⰑⰂⰑ. This spelling is part of the language identity and should render correctly in both web and PDF publications.

Slovo's core claim is:

```
written tree
-> parsed tree
-> checked tree
-> lowered tree
```

```
-> LLVM IR
-> native executable
```

Most mainstream languages hide structure behind precedence, grammar exceptions, and multiple stylistic spellings. Slovo exposes structure directly with parenthesized prefix forms:

```
(fn add ((a i32) (b i32)) -> i32
  (+ a b))
```

The goal is not minimalism for its own sake. The goal is continuity between the written program, the compiler's typed representation, diagnostics, formatter output, and generated code.

3. How Slovo Differs From Lisp

Slovo is Lisp-shaped, but not a Lisp in the usual technical sense.

Similarities:

- source is parenthesized and prefix-oriented
- programs are visually tree-shaped
- tools can inspect forms without complex precedence parsing
- the syntax is regular enough for reliable generation and repair

Important differences:

- Slovo is statically typed at the language boundary. Current source uses explicit signatures such as `i32`, `i64`, `f64`, `bool`, `string`, concrete `option` / `result` families, fixed arrays, and concrete vector families.
- Slovo does not promote a macro system as the center of the language. Convenience must lower into one specified typed core and formatter behavior.
- Slovo source is not presented as ordinary runtime list data. The source tree is a compiler contract, not a promise that programs manipulate themselves as lists.
- Slovo has no reader-conditionals, package-time dynamic namespace semantics, or hosted REPL-first execution model.
- Slovo rejects hidden numeric conversion. Numeric widening and narrowing are explicit standard-runtime calls or result-returning conversions.
- Slovo uses `option` and `result` instead of null and exceptions as the intended ordinary absence/failure model.
- Slovo reserves raw memory and low-level operations behind lexical `unsafe`.
- Slovo is compiled by Glagol through a visible pipeline to LLVM IR and a hosted native executable path.
- Slovo treats canonical formatting, diagnostics, test metadata, artifact manifests, and benchmark publication discipline as part of the language contract.

Compared with Common Lisp, Slovo is much less dynamic and does not have CLOS, conditions, a mature macro system, or an interactive image-centered tradition. Compared with Clojure, Slovo is not hosted on the JVM, does not center persistent data structures today, and does not rely on dynamic vars or runtime sequence abstractions. Compared with Scheme, Slovo is not aiming at a small untyped lambda-calculus core. Slovo's current direction is closer to a tree-shaped systems language with a small checked core.

4. Current Language State

At the current published beta baseline, Slovo can express and Glagol can compile small local projects using:

- modules, explicit imports and exports, local packages, and workspace membership
- functions and top-level tests
- `let`, `var`, `set`, `if`, and `while`
- current concrete `match` behavior over the promoted option/result/enum lanes
- `i32`, `i64`, `u32`, `u64`, finite `f64`, `bool`, immutable `string`, and internal `unit`
- explicit numeric widening plus selected checked narrowing conversions
- integer remainder and integer bitwise operations on promoted integer lanes

- payloadless enums plus unary direct payload variants over `i32`, `i64`, `f64`, `bool`, and `string`, plus unary payload variants over current known non-recursive struct types
- structs with direct scalar, immutable string, current enum, numeric, current concrete option/result, current concrete vector, current known non-recursive struct fields, and direct fixed immutable array fields
- fixed immutable arrays over `i32`, `i64`, `f64`, `bool`, and `string` with positive literal lengths and checked `i32` indexing
- concrete runtime-owned vector families `(vec i32)`, `(vec i64)`, `(vec f64)`, `(vec bool)`, and `(vec string)` with source-authored standard facades
- string concatenation, string length, string equality, and concrete parse and format helpers
- basic IO, stderr output, stdin result flow, process arguments, CLI facades, environment lookup, text file read/write, randomness, monotonic time, sleep, and scalar C FFI
- lexical `unsafe` boundaries for reserved raw operations
- staged source-authored standard-library modules under `std/`

The staged source library currently includes:

```
std/cli.slo
std/env.slo
std/fs.slo
std/io.slo
std/math.slo
std/num.slo
std/option.slo
std/process.slo
std/random.slo
std/result.slo
std/string.slo
std/time.slo
std/vec_bool.slo
std/vec_f64.slo
std/vec_i32.slo
std/vec_i64.slo
std/vec_string.slo
```

These modules are beta standard-library source facades. They are explicit-import APIs with compatibility discipline, but they are not yet frozen stable `1.0` standard-library APIs. Glagol supports explicit standard-source imports, installed `share/slovo/std` discovery, `lib/std` discovery from a checkout, and `SLOVO_STD_PATH`; Slovo does not yet claim stable implicit standard imports or a stable package ecosystem.

5. Current Toolchain State

Glagol is the first compiler for Slovo. The supported compiler path is:

```
.slo source
-> tokens
-> S-expression tree
-> AST
-> typed AST
-> LLVM IR text
-> Clang + runtime/runtime.c
-> native executable
```

The toolchain currently provides:

- `glagol check`

- `glagol fmt`, `fmt --check`, and `fmt --write`
- `glagol test`
- `glagol build`
- `glagol doc`
- JSON diagnostics
- textual artifact manifests
- lowering inspection
- native executable output through hosted LLVM/Clang integration
- `glagol run` for build-and-execute workflows
- `glagol clean` for generated `.slovo/build` artifacts
- `glagol new --template binary|library|workspace`
- `scripts/install.sh` with installed `bin/glagol`, `share/slovo/std`, and `share/slovo/runtime/runtime.c`
- benchmark scaffolds for Slovo, C, Rust, Python, Clojure, and Common Lisp/SBCL comparisons
- repo-local release-gate and document-render scripts for publication artifacts

The compiler implementation is intentionally conservative. When a source form is not specified, the desired behavior is a structured diagnostic rather than a panic or invalid LLVM.

6. Benchmark Method

The benchmark suite measures local-machine behavior only. It is a regression and comparison harness, not a public performance claim.

Environment:

Field	Value
Host	`Linux 6.17.10-100.fc41.x86_64 x86_64 GNU/Linux`
Glagol	`glagol 1.0.0-beta` benchmark baseline
Python	`Python 3.13.9`
C compiler	`clang version 19.1.7 (Fedora 19.1.7-5.fc41)`
Rust	`rustc 1.77.2 (25ef9e3d8 2024-04-09)`
Clojure	`1.11.2`
Common Lisp	`SBCL 2.5.9-1.fc41`

Build and runtime paths compared:

Implementation	Build/runtime path
Slovo	`glagol build <benchmark>` -> generated LLVM -> host `clang -O2` linking `runtime/runtime.c`
C	`clang -O2 -std=c11` on the local scaffold
Rust	`rustc -C opt-level=3 -C debuginfo=0` on the local scaffold
Python	`python3` running the local scaffold
Clojure	`clojure` running the local scaffold; timings include JVM and Clojure startup
Common Lisp	`sbcl --script` running the local scaffold; timings include SBCL startup

Timing semantics:

- The runner builds each implementation once before timing. The reported benchmark numbers are execution timings, not compile-time timings.
- `cold-process` launches a fresh process per sample with the base loop count. It measures process startup plus one normal benchmark run.
- `hot-loop` also launches a fresh process per sample, but with the amplified loop count `10000000`; the reported normalized median divides the timed total by `10` to compare with the base `1000000` loop count.

Benchmark kernels:

- `math-loop`: scalar arithmetic accumulation
- `branch-loop`: scalar branching and accumulation
- `parse-loop`: repeated decimal parsing with checksum validation
- `array-index-loop`: checked fixed-array indexing and scalar accumulation
- `string-eq-loop`: exact string content equality reduced to an `i32` checksum
- `array-struct-field-loop`: checked fixed-array access through direct struct fields plus scalar accumulation
- `enum-struct-payload-loop`: unary enum payload `match` plus bound non-recursive struct field access reduced to a deterministic checksum
- `vec-i32-index-loop`: runtime-owned `i32` vector indexing plus scalar accumulation
- `vec-string-eq-loop`: runtime-owned string vector indexing plus exact string equality reduced to an `i32` checksum

Comparison boundaries:

- `math-loop` and `branch-loop` compare structurally similar loop bodies across all implementations.
- `parse-loop` keeps the same input text and checksum, but not the same parser implementation. Slovo uses `std.string.parse_i32_result`, C uses `strtol`, Rust uses `text.parse::<i32>()`, Python uses `int`, Clojure uses `Integer/parseInt`, and Common Lisp uses `parse-integer`.
- `array-index-loop` should stay on immutable fixed-array indexing plus scalar accumulation only. The Slovo lane should use the promoted fixed-array surface, not vector helpers or newer container experiments.
- `string-eq-loop` measures exact content equality only. It should use fixed ASCII data and an explicit checksum reduction, not pointer identity, normalization, locale handling, regex engines, or concat-heavy allocation.
- `array-struct-field-loop` should stay on the exp-120 surface only: direct fixed-array struct fields, existing immutable struct flow, checked indexing, and scalar accumulation. It should not widen into array mutation, nested arrays, or unrelated container experiments.
- `enum-struct-payload-loop` should stay on the exp-121 surface only: unary enum payload variants over current known non-recursive struct types, existing immutable enum flow, `match` payload binding, and bound-field access. It should not widen into direct array/vec/option/result payloads, recursive payload graphs, or mutation.
- `vec-i32-index-loop` should stay on the current promoted runtime-owned `(vec i32)` lane only: immutable vector creation, checked vector indexing, and scalar accumulation. It should not widen into mutation or unrelated collection experiments.
- `vec-string-eq-loop` should stay on the current promoted runtime-owned `(vec string)` lane only: immutable vector creation, checked vector indexing, and exact string equality. It should not widen into regex, normalization, locale handling, or concat-heavy allocation experiments.
- Because Rust is timed at `opt-level=3` while Slovo and C are timed through `clang -O2`, the suite is a useful local regression/comparison harness, not a strict same-flags compiler shootout.

Hot-loop commands:

```
python3 benchmarks/math-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/branch-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/parse-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-index-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/string-eq-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
```

```
compiler/target/debug/glagol
python3 benchmarks/array-struct-field-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/enum-struct-payload-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-i32-index-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-string-eq-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
```

Cold-process commands:

```
python3 benchmarks/math-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/branch-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/parse-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-index-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/string-eq-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-struct-field-loop/run.py --mode cold-process --repeats 3 --warmups 1 --
glagol compiler/target/debug/glagol
python3 benchmarks/enum-struct-payload-loop/run.py --mode cold-process --repeats 3 --warmups 1 --
glagol compiler/target/debug/glagol
python3 benchmarks/vec-i32-index-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-string-eq-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
```

7. Benchmark Results

The benchmark rows below remain the full-suite `1.0.0-beta` publication baseline. `1.0.0-beta.1` changes tooling and install workflow only; it does not claim changed benchmark performance.

The exp-123 publication baseline widened the paired same-machine result set from seven rows to nine by adding two owned-vector kernels:

- `vec-i32-index-loop`
- `vec-string-eq-loop`

Hot-loop normalized median time, in milliseconds per one million iterations:

Benchmark	Slovo	C	Rust	Python	Clojure	Common Lisp/SBCL
math-loop	1.121	1.121	1.138	111.092	241.220	1.753
branch-loop	2.014	2.012	2.032	114.016	241.469	4.624
parse-loop	6.456	16.233	7.169	134.465	264.669	20.108
array-index-loop	1.103	1.109	1.128	96.649	298.388	3.379
string-eq-loop	4.332	4.092	2.279	120.453	288.617	11.128
array-struct-field-loop	1.139	1.116	1.129	110.854	277.466	3.663

enum-struct-payload-loop	4.304	1.512	1.880	302.252	310.066	5.297
vec-i32-index-loop	1.328	1.103	1.131	111.153	272.914	2.231
vec-string-eq-loop	5.210	4.122	3.471	122.826	302.817	10.431

Cold-process median time, in milliseconds per benchmark run:

Benchmark	Slovo	C	Rust	Python	Clojure	Common Lisp/SBCL
math-loop	1.625	1.675	1.765	121.014	2808.812	9.435
branch-loop	2.563	2.517	2.682	130.790	2674.146	15.027
parse-loop	6.942	16.749	7.857	149.594	2835.421	27.750
array-index-loop	1.599	1.606	1.807	107.150	2812.157	17.589
string-eq-loop	4.826	4.756	2.938	135.748	2892.359	21.504
array-struct-field-loop	1.670	1.612	1.837	115.371	2823.026	13.411
enum-struct-payload-loop	4.934	2.000	1.783	291.850	2516.815	27.555
vec-i32-index-loop	3.047	2.851	1.776	112.950	2911.603	10.017
vec-string-eq-loop	5.427	4.575	4.081	134.914	2567.482	18.950

Interpretation boundaries:

- `math-loop`, `branch-loop`, and `array-index-loop` isolate different parts of the current lowered execution path: arithmetic, branching, and checked array indexing.
- `array-struct-field-loop` is the composite-data extension of that array lane. It isolates exp-120 direct fixed-array struct fields plus checked field-path indexing without claiming new array semantics.
- `enum-struct-payload-loop` is the composite-data extension of the current enum lane. It isolates exp-121 unary non-recursive struct payloads plus existing `match` payload binding and bound-field access without claiming new ADT semantics.
- `vec-i32-index-loop` is the owned-collection extension of the integer array lane. It isolates the promoted runtime-owned `(vec i32)` surface without claiming generic collections, mutation, or broader container semantics.
- `vec-string-eq-loop` is the owned-collection extension of the string array lane. It isolates the promoted runtime-owned `(vec string)` surface without claiming broader string-library or collection semantics.
- `parse-loop` and `string-eq-loop` exercise richer library/runtime behavior than the pure scalar kernels and should be interpreted as end-to-end local implementation comparisons, not only backend loop-lowering comparisons.
- Cold-process timings include executable or host startup plus one benchmark run; they are not compile-time numbers.
- Hot-loop timings are startup-amortized process timings, not an in-process high-resolution benchmark harness.
- Clojure and Common Lisp remain useful as two distinct Lisp-family comparison points rather than one generic "Lisp" bucket.
- Clojure startup dominates the cold-process mode and still heavily influences the hot-loop process timing, which is why its results are much larger than the native lanes in this harness.

Threats to validity:

- all timings are from one local machine on one date
- implementations are intentionally small benchmark equivalents, not idiomatic full applications
- Slovo, C, and Rust compile to native executables while Clojure is JVM-hosted and Python is interpreter-hosted
- hot-loop mode is still process timing, not a stable in-process microbenchmark protocol
- `string-eq-loop` is only valid when every lane uses explicit content equality rather than identity shortcuts
- the two composite-data kernels are valid only when they stay within the already promoted exp-120 and exp-121 surfaces rather than broadening semantics in benchmark-only ways

- no benchmark threshold is part of the language contract

8. Why The Benchmark Matters

The useful result is not a blanket claim that Slovo is faster than mature native languages. The narrower result is:

- the benchmark suite now exercises arithmetic, branching, parsing, checked fixed-array indexing, exact string equality, direct fixed-array struct-field access, and unary enum payload matching over non-recursive struct payloads
- the current hosted native path is now measured across both scalar and newer promoted fixed-array, composite-data, and string behaviors rather than only older scalar kernels
- deterministic checksums prevent accidental benchmark invalidation
- the suite separates startup-inclusive and startup-amortized questions
- future optimization work can be measured without changing the language contract

9. What Is Missing Before Stable

Slovo is now beta, but it should not be called stable until beta feedback, compatibility policy, and conformance coverage have hardened the public contract.

Major remaining gaps before `1.0.0`:

- richer strings and collections beyond the current concrete fixed-array and concrete vector families
- broader data modeling, especially container composition and more general ADT ergonomics
- reusable abstractions and generics
- unsigned and narrower integer families, `f32`, broader casts, and mixed numeric policy
- stable standard-library APIs and compatibility guarantees
- clearer package dependency behavior and version policy
- richer host errors and broader file/process/time APIs
- editor integration, conformance suites, and compatibility gates
- semantic versioning and deprecation policy
- a clear separation between stable and experimental features

10. Path Beyond `1.0.0-beta.1`

The beta threshold is now real. The next work should treat `1.0.0-beta` as the language compatibility-governed baseline, and `1.0.0-beta.1` as the first tooling/install hardening point, then move deliberately toward stable general-purpose status.

Recommended sequence:

1. Collection and container breadth: improve strings, vectors, arrays, and current container composition without losing beta compatibility discipline.
2. Data modeling breadth: expand ADTs, error modeling, match behavior, and composite value ergonomics.
3. Numeric completeness: refine `f32` policy, additional integer families, and broader explicit parse/format/conversion coverage.
4. Standard-library stabilization: move from staged source facades to clearer compatibility-governed APIs and documentation.
5. Package discipline: harden local packages, path dependencies, and artifact manifests before any registry work.
6. Tooling hardening: strengthen diagnostics, generated docs, conformance fixtures, benchmark publication discipline, and release-gate automation.
7. Stable freeze: publish `1.0.0` only after beta feedback, compatibility policy, and migration/deprecation rules are proven across multiple releases.

Benchmark expansion is useful because it exercises more of the promoted surface. It is not itself a beta claim.

11. Conclusion

Slovo is now a serious beta language with a native compiler and a broad enough source-authored standard-library surface to support ordinary local command-line tools and libraries. Its strongest technical identity is not that it looks like Lisp. Its identity is

that the source tree remains visible and accountable from writing through diagnostics, formatting, checking, lowering, benchmarking, and native execution.

The path ahead is no longer “reach beta.” It is to keep the beta contract honest while hardening compatibility, libraries, packages, and tooling toward `1.0.0`.