

# Slovo Language Manifest: A Typed Structural Source Contract

Sanjin Gumbarevic  
[hermeticum\\_lab@protonmail.com](mailto:hermeticum_lab@protonmail.com)

*The tree is the language.*

**Slovo** (Словѣнскъ) is a programming language designed for humans, tools, compilers, LLMs, and machines.

Its name means **word**.

Its Glagolitic spelling is Словѣнскъ.

Slovo is built on a simple belief:

*Source code should already be structure.*

A Slovo program should be easy to read, easy to generate, easy to check, easy to transform, easy to repair, and easy to compile.

Slovo is not designed to be clever.

Slovo is designed to be clear.

---

## Purpose

Slovo exists for an age where programs are written, read, checked, transformed, and repaired by both humans and machines.

It is designed for:

- humans who want clarity
- tools that need structure
- compilers that need precision
- machines that need efficient code

LLMs are treated as tools that generate, repair, and explain code.

Slovo should allow a human, an artificial intelligence, a static analyzer, and a compiler to inspect the same program structure, and allow the machine to execute code derived from that structure.

The goal is not only to write programs.

The goal is to write meaning in a form that can become machine code.

---

## Core Idea

Most programming languages hide their structure behind syntax.

Slovo exposes structure directly.

In Slovo, source code is tree-shaped.

A form looks like this:

```
(name argument argument argument)
```

Example:

```
(+ 2 3)
```

A function is also a form:

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

The source is close to the AST.

The AST is close to the compiler.

The compiler is close to the machine.

***The tree is the language.***

---

## What Makes Slovo Different

Slovo is not primarily a Lisp, a scripting language, a macro system, or an academic intermediate representation.

Slovo is a typed structural language aimed at systems programming, where the written program, the checked program, and the lowered program remain visibly related.

Its central bet is that a language can make source code look like a tree without giving up native compilation, safety, tooling, or practical systems programming.

Slovo is built around continuity of structure:

```
written tree
↓
parsed tree
↓
checked tree
↓
lowered tree
↓
LLVM IR
↓
machine code
```

The written form should not be a disguise.

The written form should be the beginning of the compiler's truth.

---

## The Four Readers

Slovo serves four readers:

- the human
- the tool
- the compiler
- the machine

LLMs are treated as tools that generate, repair, and explain code.

### The Human

A human should be able to read Slovo without guessing.

The language should avoid hidden behavior, ambiguous grammar, precedence traps, and unnecessary stylistic choices.

### The Tool

A tool should be able to parse, format, inspect, transform, repair, test, and explain Slovo code reliably.

Tooling is part of the language.

## The Compiler

A Slovo compiler should not need heroic parsing tricks.

The grammar should be small.

The core language should be smaller.

Every advanced feature should lower into a stable typed core.

## The Machine

Slovo should not be only symbolic.

It should compile efficiently.

The primary compilation target is **LLVM IR**.

---

## Canonical Form, Not Crudeness

Slovo values consistency over personal style.

There should not be five competing ways to increment a value.

Not this:

```
x++
x += 1
inc x
x = x + 1
```

But this canonical core meaning:

```
(set x (+ x 1))
```

A Slovo formatter is part of the language.

Formatting is not decoration.

Formatting is shared understanding.

Surface conveniences may exist if they lower clearly into one canonical core form and the formatter makes their structure obvious.

The rule is not that Slovo must be minimal at all costs.

The rule is that Slovo must not let convenience hide meaning.

---

## Explicitness

Slovo does not guess silently.

A value of one type does not secretly become another type.

This should not be allowed:

```
(+ 1 2.5)
```

The conversion must be explicit:

```
(+ (cast f64 1) 2.5)
```

Explicitness makes programs easier for humans to review, easier for tools to repair, and easier for compilers to trust.

---

## Safety and Memory

In safe Slovo code:

- values are initialized before use
- nullable absence is represented by `option`
- recoverable failure is represented by `result`
- casts are explicit
- array and slice access is bounds-checked
- raw memory operations require `unsafe`
- unsafe code is lexically visible
- uninitialized memory is not observable from safe code

Safety does not mean the program is correct.

Safety means ordinary Slovo code avoids classes of machine-level undefined behavior unless the programmer explicitly enters `unsafe`.

Early Slovo should begin with a simple memory model:

```
v0:  
- managed heap values for safe code  
- raw pointers only in unsafe  
- slices are bounds-checked  
- option/result instead of null/exceptions  
- explicit allocation APIs  
- no concurrency memory model yet
```

Later Slovo may evolve toward:

```
v1+:  
- arenas / regions  
- ownership or affine resources if needed  
- FFI memory contracts  
- layout annotations
```

Memory must never be hidden magic.

Allocation, ownership, aliasing, and lifetime should become more visible as code moves closer to the machine.

---

## The Lowering Principle

Every surface feature must explain how it lowers.

If a feature cannot be described as a transformation into the typed core, it does not belong in the language yet.

A feature is not accepted because it is expressive.

A feature is accepted when its meaning remains visible after lowering.

Lowering is not an implementation detail.

Lowering is part of the language contract.

---

## LLVM as the Main Target

Slovo aims at LLVM.

The intended compiler pipeline is:

```
Slovo source
↓
S-expression parser
↓
surface AST
↓
desugaring
↓
typed core AST
↓
Slovo IR
↓
LLVM IR
↓
native machine code
```

LLVM is not only a backend target.

It is a constraint that forces Slovo to define types, control flow, memory, and calling conventions precisely.

---

## Tooling Is Part of the Language

The parser, formatter, type checker, diagnostic format, package layout, test runner, and lowering inspector are part of Slovo's promise.

They are not optional accessories.

A Slovo implementation that accepts code the formatter cannot canonicalize is incomplete.

A Slovo implementation that reports errors only as unstructured text is incomplete.

A Slovo implementation that cannot show how a surface form lowers is incomplete.

The toolchain should make the tree visible.

---

## Non-Goals

Slovo is not trying to:

- replace Python for quick scripting
- be maximally terse
- support every programming paradigm
- become a macro playground in early versions
- hide machine costs
- optimize for clever one-liners
- compete with Lisp on dynamic metaprogramming
- compete with C on minimal implementation size
- compete with Rust on advanced borrow-checking before Slovo's core is stable

A language becomes stronger when it knows what it is not trying to be.

---

## Slovo and Lisp

Slovo honors Lisp.

It inherits the belief that simple symbolic forms can express deep systems.

But Slovo is not Lisp renamed.

Lisp traditionally treats code-as-data as a source of expressive power.

Slovo treats code-as-tree as a source of compile-time discipline.

Lisp asks:

*What can the programmer express?*

Slovo asks:

*What can the human, tool, compiler, and machine all agree on?*

Slovo takes Lisp's structural beauty and aims it at modern compilation.

## The Logo

The Glagolitic spelling of Slovo is:

ഭക്ഷണം

It is written letter by letter as *slovo*.

It should be drawn simply, with respect for its origin.

The symbol should be used with cultural humility, not as exotic ornament.

A good Slovo logo should feel like an old letter that learned to compile.

## Closing Declaration

Slovo begins with a simple belief:

A programming language can be symbolic without being vague.

It can be low-level without being hostile.

It can be AI-friendly without being gimmicky.

It can be old in spirit and new in purpose.

It can treat code as words, words as structure, and structure as something the machine can faithfully execute.

**Slovo is a language where source is structure.**

**The tree is the language.**