# C to Symbolic-Maths Converter: Requirements Document

## Overview & Goals

### 1.1 Purpose

Although neural networks have produced a step-change in artificial intelligence, they could go further. They are very good at learning patterns in things; especially in word and mathematical configurations. However, they have difficulty going much beyond what the human already knows and can do. This is basically due to the inability to optimise the neural network. It’s just a black box, and the mathematics is not a true representation of the mathematics; it’s just number weights in nodes in an interconnected network.

So ideally, we need something to be able to represent objects and patterns in true symbolic mathematics, so we can apply optimisation to the mathematics. And go far beyond what we can with an unoptimized neural network.

Where do we start then? Where can we tap into the mathematics of things and start optimising them. The obvious place to start is with code; extracting the way code performs its calculations into symbolic mathematical form, optimising it, then converting it back into code to see if there’s an improvement.

We can then expand on this to start drawing in the blocks that make up a possible android brain, such as image processing, audio and language processing, simulation modelling, scheduling and planning. All things that could be used to start automating everything. Not that generative AI can’t help us with this at the moment, but it still has certain limitations. Applying optimised mathematics is the beginning of overcoming these limitations. And the development of a code to symbolic mathematics converter is the start of that journey.

### 1.2 Stakeholders

I intend to eventually make the code available on a public repository like GitHub, so anyone can help by contributing to the project. But to start with, it will just be my own work.

And until that time, I will be using it for my own development and understanding purposes.

### 1.3 Success Criteria

* **MVP Success:** Able to parse small C11 source files (≤500 LOC) and export equivalent SymPy expressions (via srepr JSON) without crashes.
* **Round-trip Fidelity:** For a golden corpus of C examples, pipeline C → IR → SymPy → IR → C preserves semantics (ignoring formatting).
* **Basic Types & Expressions:** Support for integers, floats, variables, arithmetic ops, assignments, conditionals, and functions with parameters/returns.
* **Diagnostics:** Syntax errors and unsupported constructs produce clear file:line:col diagnostics.
* **Performance:** End-to-end conversion of 10 KLOC in under 1 second on a laptop-class machine.
* **Extensibility:** IR schema is versioned; new constructs can be added without breaking old tests.

## 2. Scope & Non-Goals

### 2.1 In-Scope (v1: C-only)

* **Language version:** ISO **C11** baseline; path to C17.
* **Coverage:** translation units, declarations, definitions, types (builtins, qualifiers, pointers, arrays, function types, structs/unions, enums, typedefs), expressions, statements/control flow, function pointers, storage classes, linkage.
* **Bidirectional conversion:** C **⇄** SymPy representation (export & ingest).
* **Build integration:** compile database (compile\_commands.json), out-of-tree headers supported via include paths.
* **Diagnostics:** file/line/column with macro-expansion origins (metadata).

### 2.2 Non-Goals for v1 (planned for v1.x or later)

* GNU/Clang **extensions** (unless explicitly whitelisted), \_Atomic**,** \_Generic, and complex \_Thread\_local semantics.
* Full **preprocessor** structure preservation (macros are expanded; only origin metadata is kept).
* **ABI layout** computation (bit/byte offsets) beyond declarative field info.
* **Undefined behavior** modeling or formal verification.
* **Concurrency primitives** beyond syntactic parsing (no memory-model reasoning).
* **C++** features (classes, templates, methods, namespaces) — deferred to a later phase.

## 3. Functional Requirements

### Assumptions

* Language target: **ISO C11** (with a path to C17); GNU extensions **out-of-scope for v1**, unless explicitly listed.
* Input is processed **after preprocessing** (via the compilation database), but source locations are preserved for diagnostics.
* The converter is **bidirectional**: C ⇄ SymPy.

| ID | Title | Description | Input | Output | Constraints |
| --- | --- | --- | --- | --- | --- |
| FR-001 | Parse translation unit | Parse one or more C source files (and headers) into a C **AST** using Clang/libclang or LibTooling. | C files + compile flags/includes | In-memory C AST | Must respect compile\_commands.json; report syntax errors with file:line:col |
| FR-002 | Build symbol table | Construct scopes for files, functions, blocks; resolve identifiers, typedefs, tags (struct/enum). | C AST | Symbol table metadata | Shadowing rules of C11; extern/linkage recorded |
| FR-003 | AST → IR lowering | Lower the rich C AST to a **language-agnostic IR** capturing types, declarations, statements, exprs. | C AST | IR (in memory) | IR must be serializable and stable (versioned schema) |
| FR-004 | IR serialization (export) | Serialize IR to **JSON** (primary) for downstream tools and for SymPy emission. | IR | JSON | Deterministic ordering for diffs |
| FR-005 | IR deserialization (import) | Read JSON back into IR with full fidelity. | JSON | IR | Validate schema, reject unknown versions gracefully |
| FR-006 | **IR → SymPy emitter** | Emit **SymPy** objects representing expressions, types-as-symbols, and lambda forms for functions. | IR | **Python module (.py)** constructing SymPy objects **and/or** JSON with srepr strings | Require **Python ≥3.10** and **SymPy ≥1.12** for .py form |
| FR-007 | **SymPy → IR ingestor** | Parse a **SymPy** module (or JSON srepr) back into IR (round-trip path). | SymPy .py module or JSON srepr | IR | Static analysis of .py limited to a supported subset (no arbitrary exec); prefer JSON srepr for safety |
| FR-008 | C code generator | Generate C source from IR: declarations, function definitions, expressions, control flow. | IR | C source text | Pretty-printed; idempotent on IR where possible |
| FR-009 | Types support | Support C base types, qualifiers (const/volatile), pointers, arrays, function types, typedef, enums, structs/unions. | IR lowering/emitter paths | IR type nodes & **SymPy symbols/types metadata** | Bitfield metadata captured; flexible array members allowed |
| FR-010 | Expressions support | Literals, unary/binary ops, assignment, compound assign, comma, conditional ?:, casts, address/deref, array subscripts, member access (./->), function calls. | C AST | IR & **SymPy expressions** | Operator precedence preserved semantically |
| FR-011 | Statements & control flow | Blocks, decl-stmts, if/else, switch+case/default, loops (for/while/do), break/continue, return, labels & goto (v1.x optional). | C AST | IR & **SymPy** control forms (Lambda, piecewise, sequenced terms) | goto lowered conservatively or flagged if v1 defers |
| FR-012 | Functions | Declarations/definitions, storage class (extern/static), inline spec, parameters (incl. varargs), prototypes. | C AST | IR function nodes & **SymPy Lambda** | Varargs recorded as metadata; SymPy side may model them as argument tuples |
| FR-013 | Struct/union/enum declarations | Tag declarations/definitions, field layouts (names/types), enum constants. | C AST | IR aggregate/enum nodes & **SymPy symbols/tuples** | Layout not computed; only declarative info |
| FR-014 | Pointers to functions | Parse and represent function pointer types and uses (call through pointer). | C AST | IR types/Call nodes & **SymPy symbols** | SymPy call uses function-valued symbols |
| FR-015 | Preprocessor handling | Operate after preprocessing; optionally record macro-expansion origins for diagnostics. | Compiler pipeline | Location/macro metadata | No macro-level transformation in v1 |
| FR-016 | Diagnostics & recovery | Continue on unsupported constructs; emit warnings with precise locations; insert IR **hole** nodes. | Any phase | Diagnostic stream | Non-fatal on single-file errors when possible |
| FR-017 | CLI & library APIs | Provide a CLI for batch files and a library API to embed the converter. | Command-line args / API calls | Exit code, JSON, files | CLI supports --emit=[ir,json,sympy,c]; stdout/stderr separation |
| FR-018 | Round-trip tests | Ensure semantic round-trip for supported subset: C → IR → **SymPy** → IR → C. | Test corpus | Pass/fail results | Allow benign formatting diffs |
| FR-019 | Performance baseline | Process a 10 KLOC C project within target budget (1 second). | Source corpus | Timing metrics | Single-thread v1; note memory ceiling |
| FR-020 | Security & sandboxing | Do not execute user code; treat inputs as untrusted text; safe file IO only. | Files | N/A | For .py SymPy, run in **no-exec parse** mode or prefer JSON srepr |

### Notes

* **IR is still necessary** in a C-only v1 to decouple Clang’s AST shape from your symbolic model and backends (exporters/generators). It also gives you a stable schema for tests and future C++ support.
* For the **symbolic DSL**, keep effects minimal (e.g., Pure vs. State for writes, and IO/Extern for calls to unknown functions) and evolve later.

## 4. Non-Functional Requirements

* **Performance:**
  + Parse & lower a 10 KLOC C project in ≤ 500 ms on a modern laptop (target; adjust per hardware).
  + Symbolic export/import adds ≤ 20% overhead over parse+lower.
* **Scalability:**
  + Successfully process projects with ≥ 200 KLOC within ≤ 2 GB RAM (single process).
* **Reliability & Robustness:**
  + Non-fatal handling of unsupported constructs via diagnosed holes in IR; exit non-zero only on fatal parse errors.
  + Deterministic output (stable ordering and IDs) for identical inputs.
* **Extensibility:**
  + IR schema is versioned; unknown fields are ignored with warnings; new node kinds pluggable without core rewrites.
* **Portability & Dependencies:**
  + Builds on Linux/macOS/Windows; toolchain pinned to Clang/LLVM X.Y (to be chosen) with minimal optional deps.
  + SymPy path: For emission/ingestion in .py form, require Python ≥3.10 and SymPy ≥1.12. For safer pipelines, prefer \*\*JSON \*\*``, which does not require executing Python.
* **Security:**
  + Treat input as untrusted text; no code execution, dynamic linking, or eval.
  + Path sanitization and safe file IO; sandboxing guidance for CI.
* **Observability:**
  + Structured logs (JSONL) for pipeline stages; --trace flag emits stage timings and counts.
* **Usability:**
  + Clear CLI with --emit selection, --warnings-as-errors, and machine-readable diagnostics format.

## 5. High-Level Architecture

### 5.1 Components

* **C Front-End (Clang/LibTooling or libclang)** — parses C into a rich AST and supplies source locations. Uses **Visitor pattern** to traverse AST and feed into IR.
* **Intermediate Representation (IR)** — language-agnostic typed tree for decls/stmts/exprs. Constructed via an **IR Factory/Builder**.
* **SymPy Emitter(s)** — IR → SymPy (Python objects or JSON srepr). Designed via **Strategy pattern** so different emitters can be swapped.
* **SymPy Ingestor(s)** — SymPy (JSON/.py) → IR (round-trip). Also a Strategy.
* **C Code Generator** — IR → pretty-printed C source. Another emitter Strategy.
* **CLI/Driver Layer** — orchestrates stages; provides a **Facade** over the library API. Implements a simple **Pipeline**: Parse → Lower → Emit.

### 5.2 Data Flow Diagram

C Source → C Front-End (AST) → IR Lowering → SymPy Emitter → SymPy module / srepr JSON  
SymPy module / srepr JSON → SymPy Ingestor → IR → C Code Generator → C Source

### 5.3 Interfaces & Formats

* **Inputs:** Files, compile database, include paths; CLI flags / library calls.
* **Outputs:** JSON IR, SymPy artifacts (.py or srepr), generated C source; diagnostics.
* **CLI switches (initial):** --emit=[ir|sympy|c], --sympy-format=[py|srepr], --in <files>, --compile-db <path>, --std=c11, -I <inc>, --trace, --warnings-as-errors.

### 5.4 Component Responsibilities

* **C Front-End**
  + *Responsibilities:* Parse translation units, build AST, collect types/decls/bodies, provide source locations.
  + *Pattern:* **Visitor** traverses AST and hands off to IR builder.
  + *Dependencies:* Clang/LLVM.
* **IR Lowering**
  + *Responsibilities:* Convert AST → IR nodes (Decl/Stmt/Expr/Type), attach symbol links.
  + *Pattern:* **Builder/Factory** creates consistent IR nodes.
  + *Extensibility:* Versioned schema, new node kinds via visitors.
* **Emitters (JSON, SymPy, C)**
  + *Responsibilities:* Map IR into chosen output.
  + *Pattern:* **Strategy** — each emitter implements a common interface.
  + *Extensibility:* Plug in new backends (e.g., another math system).
* **SymPy Ingestors**
  + *Responsibilities:* Parse SymPy constructs back to IR.
  + *Pattern:* **Strategy**.
* **CLI/Driver**
  + *Responsibilities:* Orchestrate stages; parse CLI args; manage diagnostics; direct output.
  + *Pattern:* **Facade + Pipeline** — CLI presents a simple entrypoint hiding library complexity, chaining stages linearly.

### 5.5 Minimal Data Schemas (sketch)

* **IR (JSON) — example:**

{  
 "ir\_version": "1.0",  
 "decls": [  
 {"kind":"func", "name":"add", "type":"int(int,int)",  
 "params":[{"name":"a","type":"int"},{"name":"b","type":"int"}],  
 "body": {"kind":"return", "expr":{"kind":"binop","op":"+",  
 "lhs":{"kind":"var","name":"a"},  
 "rhs":{"kind":"var","name":"b"}}  
 }}  
 ]  
}

* **SymPy (two interchange options)**
  1. **Python module (.py)**

# sym\_add.py (emitted)  
from sympy import symbols, Lambda, Add, Integer  
# declare typed-ish symbols via metadata sidecar (not shown)  
a, b = symbols('a b', integer=True)  
add = Lambda((a, b), Add(a, b))  
exports = {"add": add}

1. **JSON with srepr strings** (preferred for safe round-trip)

{  
 "sympy\_version": "1.12",  
 "items": {  
 "add": "Lambda((a, b), Add(a, b))",  
 "symbols": {"a": "Symbol('a')", "b": "Symbol('b')"}  
 },  
 "metadata": {"types": {"a": "int", "b": "int"}}  
}

### 5.6 Typical Sequences

* **C → SymPy**: read compile DB → parse with Front-End → Lower to IR → Validate IR → Emit **SymPy** (.py or srepr) → Write file/STDOUT.
* **SymPy → C**: read **SymPy** (srepr JSON preferred) → Ingest to IR → Validate IR → Generate C → Format → Write file/STDOUT.
* **C → Symbolic**: read compile DB → parse with Front-End → Lower to IR → Validate IR → Emit Symbolic → Write file/STDOUT.
* **Symbolic → C**: read Symbolic → Ingest to IR → Validate IR → Generate C → Format → Write file/STDOUT.

### 5.7 Error Handling & Diagnostics

* Each stage returns a result + diagnostics list {severity, code, message, file, line, col}.
* Unsupported constructs insert **IR holes**: {kind:"hole", reason:"unsupported\_goto"} and emit WARN.
* --warnings-as-errors escalates WARN → ERROR.
* Fatal parse errors stop pipeline after Front-End; later stages may proceed with holes if policy allows.

### 5.8 Performance & Memory Model (high level)

* Single-threaded v1; stream-friendly JSON writing; arena allocators for IR nodes.
* Optional --parallel in v1.x to process TUs independently and merge.

### 5.9 Deployment & Integration

* **CLI binary** for batch use; **C API** (stable) and **C++ API** (header-only façade) for embedding.
* CI examples for CMake + Ninja; guidance for Bazel.
* Packages: Homebrew (macOS), apt/winget (later).

### 5.10 Extensibility Strategy

* Versioned IR (ir\_version), capability flags in symbolic modules.
* Plugin hook points: custom emitters, custom ingestors, IR passes (normalization, effect inference).
* Forward-compatible field allowances (unknown fields ignored with warning).

## 6. Testing & Validation

* **Unit Tests:**
  + Cover IR node creation, serialization/deserialization, emitter/ingestor round-trips, and code generation.
  + Include edge cases: empty functions, nested blocks, unusual but legal type declarations.
* **Round-Trip Tests:**
  + For each supported construct, verify semantic fidelity through the pipeline: C → IR → SymPy → IR → C.
  + Use a golden corpus of small C snippets with expected SymPy outputs.
  + Compare generated C to normalized source (ignoring formatting).
* **Error Handling Tests:**
  + Feed in unsupported constructs (e.g., goto, GCC extensions) and assert diagnostics are produced with correct severity and location.
  + Ensure hole nodes are inserted and pipeline continues.
* **Performance Benchmarks:**
  + Establish baseline with 10 KLOC corpus, measure parse+lower+emit time.
  + Track memory usage < 2 GB at 200 KLOC.
  + Automated regression checks in CI to detect performance degradation.
* **Integration Tests:**
  + Run converter on small open-source C projects (e.g., Lua, zlib) to ensure robustness in real-world code.
  + Verify emitted SymPy artifacts import cleanly in Python and basic evaluation succeeds.
* **Continuous Integration:**
  + All tests run in CI on Linux, macOS, Windows.
  + Matrix covers Clang versions, Python 3.10–3.12 with SymPy ≥1.12.

## 7. Roadmap & Milestones

| Phase | Description | Deliverables | Target Date |
| --- | --- | --- | --- |
| Phase 1 | Prototype arithmetic-only converter | CLI prototype, AST→IR lowering for literals/vars/binops, SymPy emitter (exprs) | [Date] |
| Phase 2 | Add control flow & functions | IR for statements/blocks, SymPy Lambda, round-trip for simple functions | [Date] |
| Phase 3 | Full type & aggregate support | Struct/union/enum IR, SymPy representation, codegen for aggregates | [Date] |
| Phase 4 | Robust testing & optimization | Comprehensive test corpus, performance benchmarks, CI across platforms | [Date] |
| Phase 5 | Optional extensions | Function pointers, varargs, advanced control flow, CLI polish | [Date] |
|  |  |  |  |

For a solo MVP:

1. **Must-haves:** FR-001 to FR-008 (parse → IR → SymPy → IR → C).
2. **Nice to have early:** FR-009, FR-010, FR-012 (basic types, expressions, functions).
3. **Later:** control flow (FR-011), structs/unions (FR-013), diagnostics polish (FR-016).
4. **Much later/optional:** function pointers, preprocessor metadata, perf/security hardening, CLI polish.