



All the Way Down

MetaCarp, a self-hosting compiler for Carp

Veit Heller, Erik Svedäng

October 2026

cyberwitchery lab

experience report · version 1.0

abstract

MetaCarp is a compiler for Carp written in Carp. It compiles its own source, and successive generations emit byte-identical C. Creating it required the language to express the machinery its Haskell implementation had supplied: binding identities, compile-time evaluation, type inference, specialization, and ownership.

This report also follows the compiler into a second use: as a library behind long-lived sessions. A session keeps Core checked between requests, accepts transactional source edits, and answers semantic queries for editors and notebooks, and an LLVM backend adds native execution that keeps compiled definitions between cells.

On an M1 Max, MetaCarp emits its own source in a median 9 seconds, against the reference compiler's 296 seconds. Separate small-cell workloads measure 31 milliseconds for warm inference and 40 milliseconds for warm native compilation and execution.

The experience suggests that self-hosting is useful as a method of language investigation: the compiler becomes a demanding program whose failures reveal a language's blind spots.

keywords MetaCarp; self-hosting compilers; persistent sessions; ownership; macro expansion; just-in-time compilation

the idea

The compiler as a library enables a new class of interactions.

1 The language as specification

Carp is a statically typed Lisp that compiles to C and manages memory without a garbage collector. Its surface resembles Clojure, while its runtime discipline is closer to an ML with ownership, borrowing, and explicit destruction (Carp contributors 2026b). Its reference compiler is written in Haskell, where recursive data, persistent maps, and garbage collection are readily available. A Carp implementation has to account for the ownership of those same compiler data structures. A syntax tree must have an owner. A captured environment must remain available for as long as a closure can call it. Reimplementing the compiler brings these guarantees and invariants into one unusually demanding, structurally complex program.

MetaCarp is an independent implementation organized around phase libraries. It reads source, evaluates macros, resolves names, infers types, specializes polymorphic functions, plans cleanup, and produces executable code. Its default backend emits C, while an alternative LLVM backend consumes the same lowered program. The implementation described and measured here is revision 956aec4, from 11 October 2026 (Heller et al. 2026). The measurements in Section 5 were made the same day using a clean checkout of that revision.

The first result is a closed bootstrap. Compiler bootstraps are conventionally described as stages (McKeeman et al. 1970; Rust Project Developers 2026). Let C_0 be the reference compiler and S the MetaCarp source. Including the C compilation and linking step in the notation, the chain is

$$C_1 = C_0(S), \quad C_2 = C_1(S), \quad C_3 = C_2(S).$$

C_1 is built by reference Carp. Self-hosting closes when C_1 builds C_2 . The next build checks repeatability: the C emitted by C_1 and C_2 for the same compiler source is byte-identical. The comparison is between translation units, with source ordering, dependencies, and the target environment held fixed. C_3 itself is never compiled; the check compares the C it would be built from.



Figure 1. The bootstrap chain. Reference Carp produces generation 1, which produces generation 2, which repeats the build. The C translation units emitted in the last two steps are byte-identical.

Artifact equality is semantically limited. Thompson showed how a compiler can reproduce behavior absent from its visible source, diverse double compilation provides a different test of source and executable correspondence (Thompson 1984; Wheeler 2009). MetaCarp’s fixed point is a determinism result. Reference Carp remains the root of the chain, alongside clang, runtime headers, and pinned library sources.¹

The second result concerns what happens after one compilation. An editor needs the type of a local binding and the source range that caused an error. A notebook needs an

¹Reducing that seed would be a further bootstrapping project, currently not on the roadmap (Bootstrappable Builds contributors 2026).

environment in which definitions can be replaced, with failed edits leaving the previous environment usable. Thus, a compiler session must retain enough information to answer those requests without re-loading Core, the standard library assumed to always be loaded. This workload depends on assumptions about identity, inference, and lifetime that a fresh process per compilation can leave implicit.

The report follows these assumptions through the implementation. We believe that this is the more useful account of self-hosting. The completed build is meaningful, but so are the language decisions it makes observable and the programs built on it.

2 The compiler as a set of libraries

The driver supplies source text and filesystem policy, while the phase libraries own language semantics. The phases fall into two groups. Checking turns a source registry into an expanded surface module, a resolved core, and recorded type information; declaration validation and the derivation of generated functions belong to it. Code generation turns the checked program into concrete function instances and an ownership plan, then lowers it to a shared backend representation that can be rendered as C or emitted through LLVM. Failures carry the name of the rejecting phase as well as source information.

Two libraries support clients across these phases. `carp-source` defines caller-owned source identities and byte spans. `carp-session` retains selected compiler results between requests. The optional native execution host sits above that session boundary.

2.1 Crossing a phase boundary

`carp-surface` preserves syntax and source spans. `carp-ir` represents resolved expressions and binding identities. `carp-types` implements monotypes, schemes, substitutions, lifetimes, and unification without depending on syntax or C. The inference phase follows the familiar Hindley-Milner shape of predeclaration, instantiation, unification, and generalization (Damas and Milner 1982), while leaving ownership decisions to a later pass. An overloaded name represents a choice constraint: a scheme can carry an open choice to its use sites, where argument types determine the final unification.

Resolution gives runtime bindings identities that later phases can retain. This matters when imports or shadowing give the same names to different bindings. Type identity has a related distinction: reference lifetimes matter to checking, but lifetime-only differences must not produce duplicate runtime layouts or monomorphized bodies.

The command-line driver reads files, assembles the standard-library source registry, resolves relative loads in the reference compiler's order (working directory, loading file's directory, then Core), installs git references into Carp's shared cache when necessary, writes generated C, invokes clang, and optionally runs the result.²

Keeping those effects outside the phase libraries means the same compiler path can accept an in-memory source registry in tests or another tool, such as an IDE. Target architecture and operating system are detected by the compiled driver, and the bootstrap runs

²Canonical paths deduplicate different spellings of the same file.

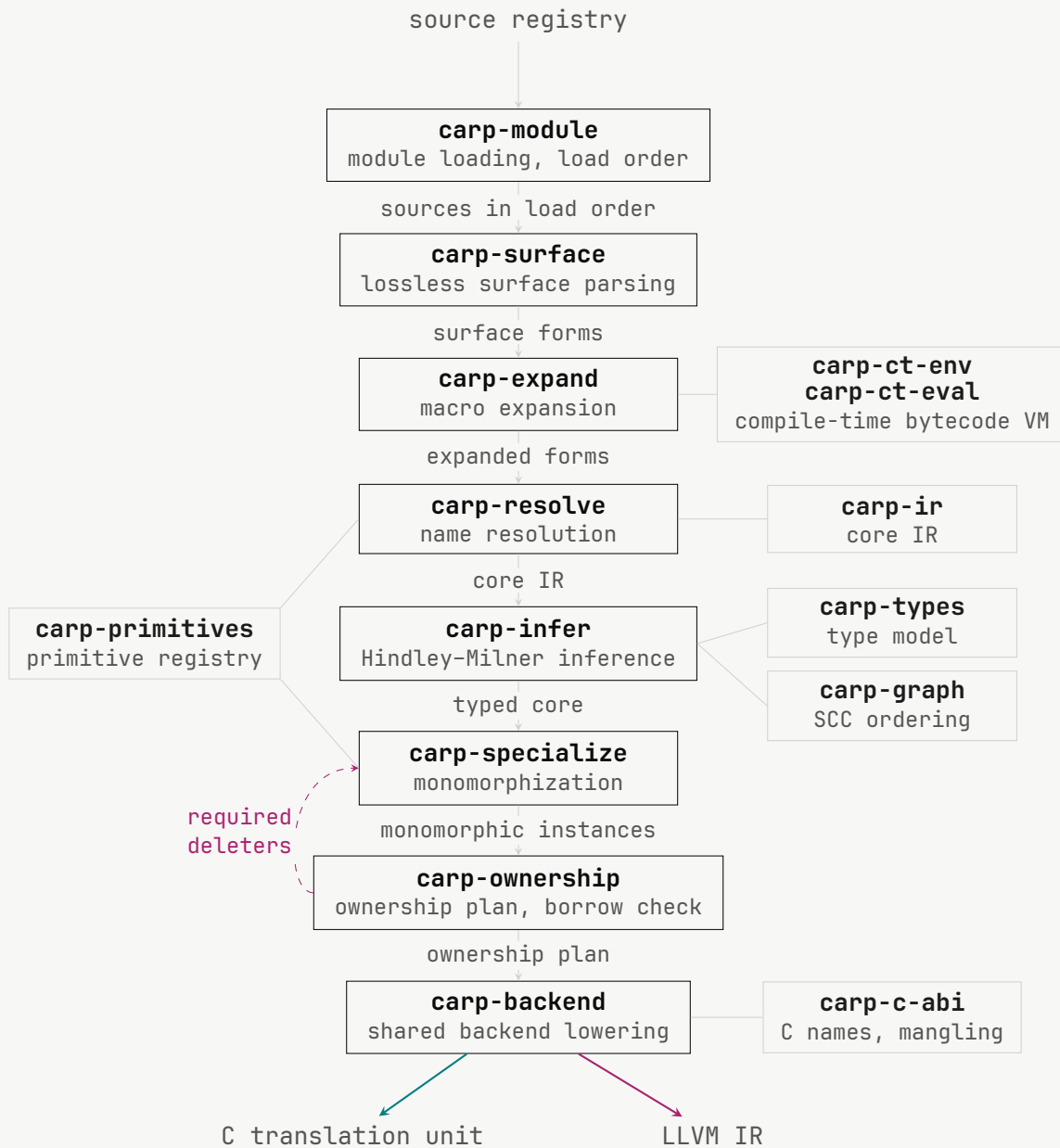


Figure 2. The batch pipeline and its supporting libraries. Both emitters consume the same lowered program. The optional LLVM emitter lives in `carp-llvm-backend`, the C renderer lives in `carp-backend`. The dashed edge returns required deleters to specialization.

on both macOS and Linux. Include paths, package acquisition, toolchain selection, and process execution therefore remain driver policy rather than language semantics.

2.2 The compiler as a service

An interactive client has to both re-check the program it already knows and check the fragment that just changed whenever it invokes the compiler. A session divides that work between an immutable base and a replaceable overlay. The base is whatever the host loads when it creates the session: `Session.create` loads Core alone, while `Session.create-from-module` accepts any module the host has already loaded, such as Core with third-party libraries or a whole project. The session checks the base once. No commit checks it again. Its definitions keep the bindings they resolved to, and its expansion snapshot is shared with each fresh overlay through reference counting and copied only once the overlay commits a definition. User source inputs form the overlay above it.

This speeds up compilation for any program that exists on top of a standard library or a set of libraries, which is to say most programs.

A client submits source as inputs: pieces of text with an identity the client chooses, such as a notebook cell or a file. A cell usually holds one definition and a file many. Every definition belongs to the input it came from, and the session replaces inputs, not fragments of them. Each input occupies one place in the replay sequence, and committing it again replaces all of its definitions together. Editing one definition by name is refused when that definition came from a larger input, because it would change part of the text the client submitted without the rest.

For example (Figure 3), a notebook cell holding `(defn square [x] (* x x))` is committed with `upsert`: one form, identified by its name. A file `geometry.carp` holding `(deftype Point [x Int y Int])` and a function `norm` over points is committed with `upsert-input` as a single input. Committing the edited file again replaces both definitions together, and a definition that has vanished from the file is dropped from the session. Replacing `norm` by name with `upsert` is refused, because `norm` belongs to the file's input; the client resubmits the file instead. A query cell such as `(norm (Point.init 3 4))` can then be inferred against the committed definitions without being installed.

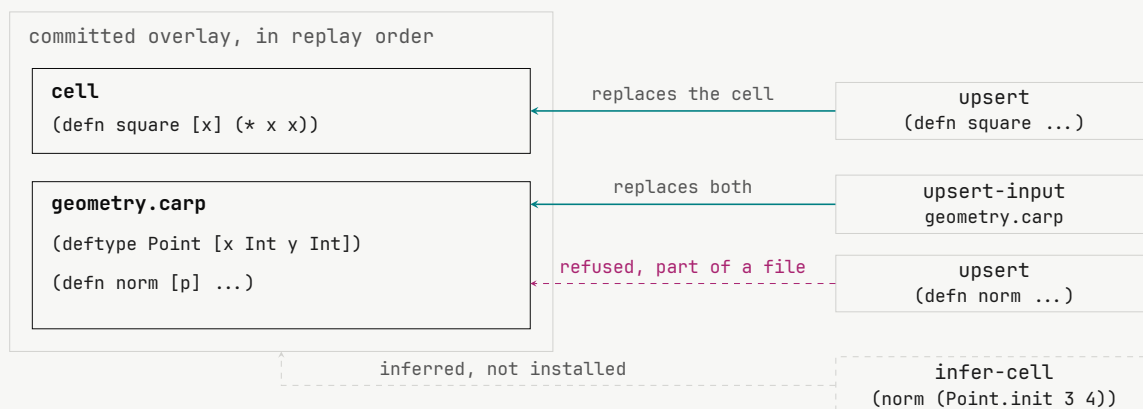


Figure 3. Inputs in a session. The cell and the file are separate inputs. Committing the file again replaces both of its definitions; replacing `norm` by name is refused because it would edit part of the file. A query cell is inferred against the committed definitions and leaves them unchanged.

An edit constructs a candidate overlay by replaying the committed inputs against the checked base. The candidate must pass checking before it replaces the current overlay. On failure, the preceding state remains usable. A successful commit does not certify ownership: code generation, and with it the ownership check, is deferred until a client requests an ownership report or compiles code. Reset drops user definitions and their caches while retaining the base.

Resolved references identify dependencies. Replacing a definition rebinds its dependents to the latest committed meaning of the name; this also applies when an overlay definition shadows a base definition. Compatible replacements can retain their global identity. Macros, types, interfaces, and implementations use broader invalidation because they can change expansion or dispatch beyond ordinary runtime reference edges.

This is persistent compilation with transactional replay. The invalidation report tells a client which definitions changed meaning, but a commit still replays the overlay. It is not a claim of selective re-inference proportional only to the edited definition. It should instead be taken as a heuristic that, when in doubt, chooses more liberal recompilation over shorter compilation times.

The cost of a commit therefore grows with the overlay, not with the base. Every commit replays all committed inputs, so a large program kept in the overlay pays for its size on each commit, while the same program loaded into the base is checked once and afterwards only read. A client can place stable libraries in the base and keep the code under edit in the overlay.

Transient operations infer or expand forms, step through expansion, report ownership, complete names, and retrieve documentation without installing the input. They return structured values and diagnostics.³

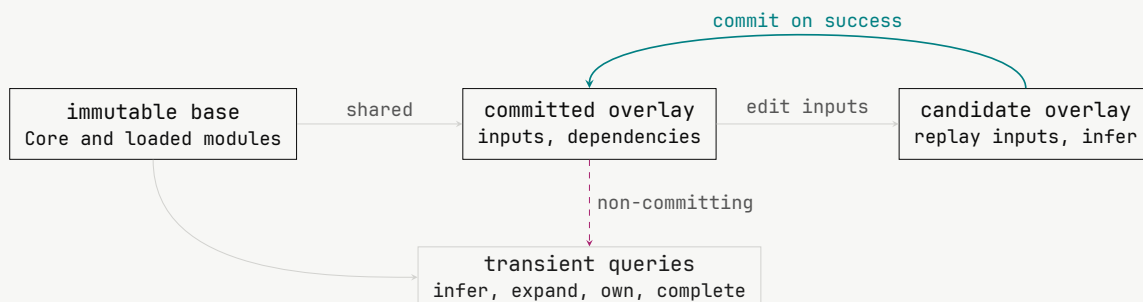


Figure 4. The semantic session. Candidate inputs replay against the checked base before commit. Transient queries use the committed environment without installing their input. Native code has a separate lifetime, described below.

2.3 Types that outlive their solver

Inference records types for expressions, calls, and bindings alongside a final substitution. A recorded type can still contain a variable whose meaning is supplied by that substitution. In batch compilation, the records and their substitution flow together, and specialization applies the substitution when it reads a type.

³Queries may populate caches or advance internal identity allocators; non-committing means that they leave the user's definitions unchanged.

A session combines results from separate runs: the base, committed inputs, and a transient cell, such as a query that is inferred but never committed. One substitution field cannot stand for all of them, so the session applies each run's substitution to that run's recorded types before combining them. The combined module carries an empty substitution. Legitimate polymorphic variables remain; what disappears is any dependence on a discarded solver result.⁴

Omitting this step fails too late. A cell can make reachable a polymorphic definition that no earlier cell instantiated. Its recorded types survive, but specialization meets variables it can no longer resolve. Tests whose base held only simple functions do not reach this case.

Normalization costs work proportional to the program it covers. The base is immutable, so its normalized analysis is computed once and retained.⁵ `Session.prepare-emit` pays the cost explicitly, otherwise the first emission does. Committing an input invalidates the combined compilation view but keeps the normalized base.

2.4 Keeping executable code between cells

`Session.emit-cell` returns a C translation unit. `Session.lower-cell` stops earlier, at the shared `BackendModule`. The latter gives the LLVM host the same lowered program that the C renderer receives.

The session library itself has no LLVM dependency.

The native host retains one ORC LLJIT and a thread-safe LLVM context (LLVM Project 2026). Each cell adds a module containing newly reached definitions, with fresh names for cell roots and initialization functions. Already published functions and globals are referenced by symbol. Their storage survives subsequent cells, so a mutable global can be observed after the cell that first initialized it has finished.

Two compiler caches support this path. A resident semantic view lets a transient cell append and retract its own intermediate forms and recorded types. A specialization cache lets later cells lower only newly reached instances. C templates are compiled into a shared-library shim and reused until a cell requires another template specialization. Shim construction follows LLVM emission, which discovers the required invocation helpers and ABI wrappers. A warm cell that needs no new templates therefore doesn't pay for any clang invocations.

Publication and source editing have distinct lifetimes. Per-cell resource trackers allow the host to remove a module if root-symbol lookup fails. Committing a definition edit conservatively clears published native code and its specialization state while retaining the LLJIT, target state, and template shim. This is in-process reuse.⁶

Source provenance reaches the backend through a shared line map. The LLVM emitter uses it for DWARF locations, while C emission can attach source-line directives. The same

⁴Generalized schemes already serve as the reusable environment for later inference, so this normalization concerns only the recorded facts consumed by specialization.

⁵Normalization builds an indexed view of the substitution once and applies it to each recorded type, avoiding a linear scan for each variable lookup.

⁶A persistent object cache and a cached `build-cell` operation remain unimplemented.

identities that attach an editor annotation to a binding can therefore connect generated instructions to source.

2.5 Source becomes data, twice over

A Lisp reader turns text into forms. Macro expansion then treats some of those forms as data manipulated by a program and expects new forms in return. Carp's macros and dynamic functions run in this second language, separate from the statically typed runtime program.

`carp-ct-env` provides frames and mutable cells with integer identities. A closure retains its defining-frame ID, a call allocates a child frame, `set!` updates a shared cell. This preserves mutation through captured environments without making each closure own a separate copy of the bindings.

The evaluator contains a compiler of its own. `CtLower` classifies syntax into a compile-time intermediate representation. `CtCompile` turns it into bytecode, including jumps, frame operations, calls, and indexed tables for constants and names. `CtEval` runs that code with a program counter and value stack. A macro body is compiled when the macro is defined, repeated calls reuse it. Dynamically supplied forms, including the argument of `eval`, are compiled when encountered.

This arrangement removes repeated interpretation of a body's syntax, but retaining name lookups introduces another contract. A free-name cache is keyed by the body's defining frame and the environment's binding epoch. Lowering must distinguish locals from free names, and changes to definitions or imports must advance the epoch. Parameter and local bindings do not advance it.⁷

`carp-expand` processes top-level forms in source order, installs macro and dynamic definitions, imports compile-time frames through `use`, and recursively expands macro results. Compile-time-only declarations and loader markers disappear before runtime resolution. Expansion remains unhygienic, matching Carp, and generated names are resolved as written (Carp contributors 2026a).⁸

The distinction between these two languages is useful beyond implementation. A macro can fail while manipulating a list, or it can successfully return syntax that later fails runtime type inference. Those are different errors, with different information available to explain them.

2.6 Specialization meets ownership

After inference, `carp-specialize` starts from concrete top-level and named demands. It resolves interface calls to implementations, settles any overload inference left open by the concrete call type, and walks each new body for further global calls. An instance is keyed by stable binding ID and concrete function type, deduplicated by runtime representation.

⁷Epochs come from a process-wide counter so diverging session snapshots cannot accidentally reuse a cache entry for different environments.

⁸A hygienic macro system remains proposed, but is currently not on the roadmap until the Carp community has reached a consensus.

A repeated request for the same binding and representation reuses the existing instance, including a recursive call at that type.

Ownership is computed over this concrete program. Types are classified from declared behavior rather than spelling: references are borrowed, a type with a `delete` implementation is owned unless a valid `blit` marks it unmanaged, the rest are unmanaged. Function values, which may carry a closure environment, and heap arrays are always owned, and a static array owns what its elements own.

A flow-sensitive pass threads moves and borrow origins through evaluation order, joins branch states, and rejects escaping local borrows, uses after moves, and borrows whose owners have moved. Its output is a side table over specialized expressions. It schedules deletion for unconsumed owned locals and parameters, discarded temporaries, overwritten `set!` values, elements overwritten by `StaticArray.set!`, unconsumed by-value match bindings, and constructor payloads discarded by wildcard patterns. Reference matches borrow and schedule no deletion. The backends consume this plan. Each emitter also frees the previous value of a field overwritten in place, at the write site.

There is one circular dependency, colored magenta in Figure 2. The ownership plan may discover that a concrete managed type needs a deleter. That deleter is itself polymorphic Carp code that must be specialized. The new instance may contain owned values and therefore reveal another required deleter. These additional instances must therefore be checked before lowering can begin.

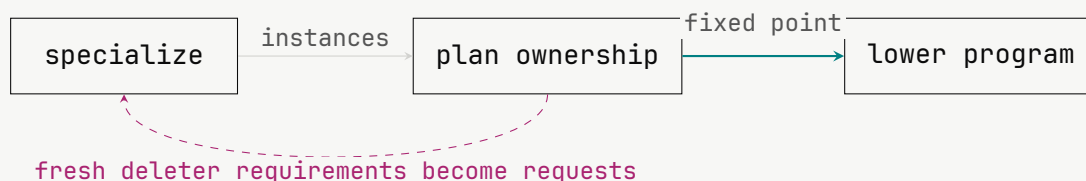


Figure 5. The specialization-ownership fixed point. Specialization produces monomorphic instances; the ownership plan for those instances may require deleters that do not exist yet, which become specialization requests for the next pass. Lowering starts once a pass requires nothing new. `carp-compiler` owns the loop. Deduplication closes it when the reachable requirement types form a finite set.

`carp-compiler`, the library that composes the phases, owns the loop in Figure 5. Each pass adds previously unseen requirements, deduplicated by runtime representation. Lowering begins when ownership requests nothing new. This terminates when the reachable concrete requirement types form a finite set. A finite source program does not guarantee that condition: non-regular recursive generic types can keep producing new instances. The implementation has no iteration cap or general termination check, and its invariant notes record a divergent example (Heller 2026b).

The bootstrap demonstrates closure for a substantial program within this boundary. The architectural benefit is that the cycle is visible: ownership reports requirements, specialization supplies them, and `carp-compiler` decides when both have finished.

3 Agreements between phases

The reference implementation distributes several agreements across its phases. Names, lifetimes, overloads, generated functions, and C layout each carry assumptions into later phases. MetaCarp states them explicitly, as the following cases show.

3.1 Identity outranks spelling

Carp allows modules, imports, shadowing, generated accessors, interface implementations, and multiple concrete instances of one polymorphic function. A source name alone cannot identify which binding a later phase means.

The new pipeline therefore makes resolver-issued IDs authoritative. Derived functions are inserted into the same core representation with fresh IDs before the main inference pass. Specialization keys contain binding identity as well as type. Ownership locations contain specialization context as well as expression ID, because the same polymorphic source body may be planned differently at two concrete instances. C names are derived from these identities in one shared ABI layer and never serve as identity themselves.

Table 1 shows the result for a small program, in which a function `twice` is used at `Int` and at `String`, and a local `n` is shadowed by a second `n` in an inner `let`. The outer `n` keeps its spelling; the inner one is emitted with its binding ID.

binding	type	C name
<code>twice</code>	<code>(Fn [Int] (Pair Int Int))</code>	<code>C5_twice__F1_1_IZ15_N4_Pair2_1_I1_I</code>
<code>twice</code>	<code>(Fn [String] (Pair String String))</code>	<code>C5_twice__F1_1_SZ15_N4_Pair2_1_S1_S</code>
<code>outer n</code>	<code>Int</code>	<code>n</code>
<code>inner n</code>	<code>Int</code>	<code>L1_n__x45_1614807157</code>

Table 1. Bindings and their C names in a small program, as emitted by MetaCarp at 956aec4. The two instances of `twice` are one binding at two types; the two locals share a spelling but are different bindings. The inner local’s name carries its binding ID, with the minus sign escaped as `_x45_`.

These identities let the compiler distinguish transformations that preserve a binding from those that create one. Runtime representation keys separately identify instances that can share emitted code despite lifetime differences.

Persistence adds a second identity problem. Resolver IDs allocated while checking one source fragment cannot be treated as stable external handles if the next fragment starts its own allocation sequence. A session therefore carries one monotonic node allocator across requests, and new global bindings continue from the highest identity already in its snapshot. The caller separately assigns each source input an identity; locations within it are half-open UTF-8 byte ranges. Provenance indexes connect definitions, bindings, and expression nodes back to those ranges. Syntax introduced by a macro retains an anchor at the invocation that produced it.

Provenance lets a client attach an inferred type or ownership action to the expression it describes. The external identity is the caller’s source ID and byte range; internal node IDs need not survive a session restart. This keeps an editor’s document identity separate from the compiler’s allocation order.

3.2 Generated code is not special

Carp derives operations such as copy, delete, accessors, and string conversion for user types. An expedient compiler can emit these directly in the backend. The self-hosting compiler instead generates ordinary core definitions before the main inference pass wherever possible. They then pass through checking and code generation by the same path as user definitions.

Generated functions therefore receive the same checking and dispatch as written functions. A derived deleter must obey its ownership convention. This also makes its mistakes ordinary compiler errors, located in generated code whose provenance must still lead back to a user declaration.

Runtime primitives and external C layouts remain registered at the foreign boundary. With LLVM, aggregate arguments and results cross that boundary through generated pointer-based wrappers. Clang handles the C ABI within the shim, LLVM-generated callers pass addresses. Both emitters consume the same ownership plan, but each places a few frees itself, such as the old value at an in-place field write and a temporary closure's environment. A memory fixture checks the field-write case under both drivers: each run must end with balanced allocations, and the two runs must print the same output.

Parameterized containers make this distinction concrete. Closing generics and the specialization fixed point together discover the element operations required by each concrete instance, including memberwise copies and deep deletion. Array operations are consequently generated for the actual nested type, and Box, an ordinary recursive Core type, is specialized like any other generic definition, rather than falling back to a shallow C copy or a container-only free. Compiler data structures depend on this: without it, a superficially correct generated program can leak or retain interior values.

3.3 The lifetime of compiler data

The compiler builds large recursive trees and many short-lived collections. Several representations of the same program can coexist during a phase transition. Their lifetime affects peak memory even when every allocation is eventually freed.

Lowering moves some deletes from scope exit to just after a binding's last use along a declaration chain. The ownership plan keeps scope lifetime when the value may be aliased, conditionally consumed, or held in a lifted temporary supporting a borrow. This is a restricted liveness optimization, with conservative fallbacks. It is not a complete control-flow liveness analysis.

Correct cleanup also has to survive lowering. Closure environments own their captured values, discarded managed values still need deletion, and a field write must free the value it displaces. These requirements reach across the type classifier, ownership plan, and backend. A correct plan alone is insufficient if lowering loses one of its scheduled operations.

4 Compatibility as a development method

The compatibility runner mirrors programs from the reference Carp repository. Positive cases must compile, test programs must run to a zero exit status, and programs with an expected output must reproduce it. Negative cases must be rejected.⁹

The runner moves with reference Carp and, as of October 13, 2026, mirrors 165 cases. There are no accepted gaps, and the reference memory suite runs without exceptions.¹⁰

The runner exercises the full Core graph through checking, code generation, and linking with the runtime headers. Phase-local tests check the intermediate contracts. A separate seven-program corpus covers macros, quasiquotation, generated symbols, dynamic evaluation, syntax sugar, and numeric promotion. Both compilers build and run each program, and their observable output must match. This checks expansion-related behavior through execution, it does not compare expanded syntax trees.

Continuous integration runs the self-hosting path on macOS and Linux. It also checks the self-built compiler with macOS's leak detector while compiling a Map test, with a default budget of zero detected leaked bytes. A separate AddressSanitizer job instruments generated programs in the reference suite. A third job builds the LLVM driver, runs its tests, and requires a memory fixture to balance and produce the same output under both drivers. The leak check covers compiler allocation during one workload; the sanitizer and LLVM jobs check generated execution.

The evidence remains selected. Some mirrored cases test only successful building, and a behavior absent from the reference tests is easy to miss. Session tests add repeated resolution and mutation in one process, which a cold compatibility run cannot exercise.

5 Performance and reuse

Compiler throughput, interactive latency, and generated-program speed answer different questions. Loading Core once changes the cost of the next cell, but does not by itself make the generated program faster. We measure these paths separately, including the first native cell and subsequent cells in one host.

The benchmark reproduced here ran on 11 October 2026, on an Apple M1 Max MacBook Pro with 32 GB of memory, macOS 15.8.1, and Apple clang 17.0.0. It used MetaCarp revision 956aec4 and reference Carp revision 1d36c69. The reference executable was built with GHC 9.6.7. MetaCarp and the session harnesses were rebuilt with `carp -b --optimize`, the native host used LLVM 23.1.3. The machine was in normal use during the run. Binary hashes, library revisions, commands, load averages, and raw measurements are retained in the supplementary materials.

Ordinary cold emissions use five runs; compiler emission and bootstrap use three; generated programs, inference sessions, and JIT sessions use seven. Paired compiler and runtime measurements alternate which implementation runs first. Each cold compilation is a fresh process, but filesystem caches are not flushed. The raw data preserves wall time, status, maximum resident set size, and macOS's peak memory footprint for every

⁹Diagnostic wording can be reported for review but does not determine whether a case passes.

¹⁰Graphical SDL programs and documentation generation are outside the runner.

process. The table uses the latter as its physical-memory measure. Runtime results are accepted only after the two executables produce identical output. All 94 measured processes completed successfully.

5.1 Cold compilation and generated code

Cold emission loads Core and produces a C translation unit for a small recursive example, a JSON parser, n-body, Mandelbrot, or MetaCarp itself. The separate bootstrap measurement includes C compilation and linking to produce C_1 .

source	median time			median peak footprint		
	ref.	Meta	ratio	ref.	Meta	ratio
mutual recursion	1.05 s	0.39 s	0.38×	153 MiB	59 MiB	0.39×
JSON parser	1.33 s	0.45 s	0.34×	158 MiB	62 MiB	0.39×
n-body	1.07 s	0.42 s	0.40×	154 MiB	61 MiB	0.40×
Mandelbrot	1.05 s	0.40 s	0.39×	153 MiB	59 MiB	0.39×
MetaCarp	295.89 s	8.68 s	0.03×	1259 MiB	440 MiB	0.35×

Table 2. 11 October 2026 cold source-to-C compilation. Ratios are MetaCarp/reference, lower is better. Each process loads Core. Ordinary sources use five runs and the compiler source uses three. “Peak footprint” is macOS’s physical-memory footprint, not the separate maximum-RSS counter retained in the raw data.

On the four smaller inputs, MetaCarp took 0.34–0.40 times as long as reference Carp, a reduction of 60–66 percent. Its median physical footprint was 59–62 MiB against 153–158 MiB. Loading and checking Core accounts for work shared by all four cold compilations.

For the compiler source, MetaCarp emitted itself in a median 8.68 seconds, against 295.89 seconds for the reference, a time ratio of 0.03, or a 97.1 percent reduction. Its median peak footprint was 440 MiB, against 1,259 MiB, a ratio of 0.35, or a 65.0 percent reduction. Reference Carp’s complete optimized bootstrap, including C compilation and linking, took a median 321.78 seconds with a 1,258 MiB footprint.

The self-built generation repeated compiler emission in 7.09 seconds with a 412 MiB footprint, producing C byte-identical to generation 1’s output. Its time was 18.3 percent shorter than the reference-built generation’s. The memory advantage depends on the counter: maximum RSS during compiler emission was 1,223 MiB for reference-built MetaCarp and 1,268 MiB for reference Carp, a much smaller difference than their physical footprints.

Demand-driven reachability also reduced the emitted C (Table 3). Generated-program speed was measured independently for n-body and Mandelbrot, with both sides compiled using `clang -O3 -D NDEBUG`. Their runtime ranges overlapped, and every output comparison passed. The large reduction in compiler time therefore does not imply a corresponding improvement in generated-program throughput.

This result is explainable by the fact that MetaCarp itself does not perform any specialized optimizations yet, although optimization passes are proposed as future work.

source	emitted C		median run time	
	ref.	Meta	ref.	Meta
mutual recursion	372 KiB	4 KiB	-	-
JSON parser	480 KiB	82 KiB	-	-
n-body	406 KiB	25 KiB	2.79 s	2.79 s
Mandelbrot	380 KiB	9 KiB	14.91 s	14.81 s
MetaCarp	10,985 KiB	6,220 KiB	-	-

Table 3. 11 October 2026 emitted C and generated programs. Emitted C is the translation unit from cold emission. Run time is the median of seven runs of the program built with `clang -O3 -D NDEBUG`, measured for n-body and Mandelbrot only.

5.2 Inference and execution in a warm session

The session harnesses create a fresh Core base in each of seven processes (Heller 2026d; Heller 2026c). Within each process, inference repeats one 43-byte cell 100 times. Native execution uses a different, array-based integer expression. The JIT harness times one first cell, then 20 warm JIT cells and five C-path cells. Warm-cell values are medians of per-process means, not individual-cell latency percentiles.¹¹

session measure	median
cold Core creation	429 ms
100 warm inference queries	3099 ms
warm inference per cell	31 ms
process peak footprint	111 MiB

Table 4. 11 October 2026 inference benchmark. Each of seven fresh processes creates one session, then infers the same 43-byte cell 100 times without committing it. The footprint covers the whole process.

Creating the checked Core base took a median 429 milliseconds. Once resident, checking the cell 100 times took 3,099 milliseconds, or 31 milliseconds per cell, with no failures across 700 queries. Per-process means ranged from 30.91 to 31.13 milliseconds. The median process peak footprint was 111 MiB. This workload measures `Session.infer-cell`, without code generation, linking, or execution.

execution measure	median
first JIT cell	306 ms
warm JIT execution per cell	40 ms
C emission, clang, execution per cell	334 ms

Table 5. 11 October 2026 native execution benchmark, across seven fresh processes. The first JIT cell excludes Core creation. Warm JIT and C-path values summarize means over 20 and five cells respectively.

¹¹All internal timers use monotonic clocks.

The first native cell took a median 306 milliseconds, including template-shim compilation but excluding Core and JIT-host creation. Warm JIT cells took 40 milliseconds each, including compilation and execution, with process means ranging from 39.7 to 40.5 milliseconds. C emission, clang, and execution took 334 milliseconds per cell, about 8.3 times as long. That lane runs after the JIT lane and invokes clang without an optimization flag. It compares interactive paths, not optimized generated-program throughput. The median process peak footprint was 120 MiB; maximum RSS was 352 MiB. No failures were reported across 147 JIT cells and 35 C-path cells.

For these small cells, both inference and native execution fit within tens of milliseconds once warm. The two harnesses use different expressions, so their times cannot be subtracted to estimate code-generation overhead. Nor do these measurements establish latency for large overlays, invalidating edits, or new template specializations.

6 Clients of the compiler

The `gt4carp` bridge is the first substantial client of `carp-session` (Heller 2026a). It uses warm semantic queries and provenance for notebook annotations, completion, documentation, and views of a type's members. These requests exercise a different interface from the command-line compiler: the useful result is often a type or source range, rather than an executable.

An experimental CLAP audio plugin exercises the native side. Its narrow entry point compiles a named (`Fn [Float] Float`) function and returns a callable address. Successful compilations publish separate generations, so earlier addresses remain callable if a later compilation fails. The plugin instance selects the active generation. Compiled generations remain resident until the plugin instance is destroyed, so repeated replacement also grows retained code. This is a small experiment in replacing running code, not a general live-update mechanism (Heller et al. 2026).

These clients make different demands on the same compiler. The notebook needs answers attached to changing text, the audio plugin needs executable code with a controlled lifetime. Their transport, presentation, and execution policies remain outside the compiler libraries. The common requirement is that the compiler's intermediate knowledge can be used without replaying its command-line workflow.

7 Limits of this implementation

MetaCarp follows a moving revision of reference Carp and its Core library. The default driver and assurance pipeline cover macOS and Linux hosting. Windows, explicit cross-target compilation, and equivalence with the reference REPL are outside that scope. The LLVM backend expands the available emission paths, but does not by itself establish portability to every LLVM target.

The last-use cleanup optimization is conservative. A delete stays at scope exit when its binding may be consumed on some control-flow path, may be aliased, or is a lifted tempo-

rary supporting a borrow. A set! into a binding whose value may have been moved on any path is rejected.

Session mutation replays committed inputs, and compile-time dependencies use broad invalidation. More precise dependency tracking could reduce edit cost. Native-code reuse is currently tied to one host lifetime, and committed edits discard published code conservatively. Concurrent mutation, process isolation, and remote-value management remain client concerns.

This report supplies implementation evidence and measurements, not a soundness proof. Ownership checking, foreign calls, and the two emitters can still contain defects. The fixed point establishes reproducibility of the compiler translation in its checked environment. The compatibility tests establish agreement on their selected programs. Neither result extends to all programs the language can express.

8 What self-hosting enabled

Self-hosting makes several of Carp’s assumptions available for inspection. Binding identity is data carried through phases. Ownership requirements are inputs to specialization. The dynamic compile-time language has its own intermediate representation and bytecode.

Persistent sessions require these representations to mean something outside the compilation that produced them. The substitution problem is the clearest example: a recorded type could survive in memory while losing the information needed to interpret it. Making that boundary explicit also marks the work that can be reused. The native host extends the same concern to the lifetime of generated code.

Alternative implementations have served related purposes in other self-hosting and debootstrapping work (Courant et al. 2022). For Carp, the second implementation provides a place to test language decisions against a substantial program, then make the resulting machinery available to other programs in the language. We think its value lies in that continued use. Each new client asks which compiler decisions can be retained and which must be recomputed.

References

- Bootstrappable Builds contributors (2026). *Bootstrappable Builds*. URL: <https://www.bootstrappable.org/> (visited on 07/21/2026).
- Carp contributors (2026a). *Macros*. URL: <https://carp-lang.github.io/carp-docs/Macros.html> (visited on 07/21/2026).
- (2026b). *The Carp Language*. URL: <https://carp-lang.github.io/carp-docs/LanguageGuide.html> (visited on 07/21/2026).
- Courant, Nathanaëlle, Julien Lepiller, and Gabriel Scherer (2022). “Debootstrapping without Archeology: Stacked Implementations in Camlboot”. In: *The Art, Science, and Engineering of Programming* 6.3, 13:1–13:26. DOI: [10.22152/programming-journal.org/2022/6/13](https://doi.org/10.22152/programming-journal.org/2022/6/13).
- Damas, Luís and Robin Milner (1982). “Principal Type-Schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 207–212. DOI: [10.1145/582153.582176](https://doi.org/10.1145/582153.582176).

- Heller, Veit (2026a). *gt4carp: A Glamorous Toolkit Bridge for Carp*. URL: <https://github.com/carpentry-org/gt4carp> (visited on 08/01/2026).
- (2026b). *MetaCarp Implementation Invariants*. URL: <https://github.com/cyberwitchery/metacarp/blob/956aec4/docs/invariants.md> (visited on 10/11/2026).
- (2026c). *MetaCarp LLVM Backend and Session JIT*. URL: <https://github.com/cyberwitchery/metacarp/blob/956aec4/carp-llvm-backend/README.md> (visited on 10/11/2026).
- (2026d). *MetaCarp Persistent Sessions*. URL: <https://github.com/cyberwitchery/metacarp/blob/956aec4/carp-session/README.md> (visited on 10/11/2026).
- Heller, Veit, sqrew, Christian Weilbach, and Erik Svedäng (2026). *MetaCarp: A Self-Hosting Compiler for Carp*. Implementation snapshot, 11 October 2026. URL: <https://github.com/cyberwitchery/metacarp/tree/956aec44ff60ea46190096acd92fc1e2ef032e3f> (visited on 10/11/2026).
- LLVM Project (2026). *ORC Design and Implementation*. URL: <https://llvm.org/docs/ORCv2.html> (visited on 09/17/2026).
- McKeeman, William Marshall, James J. Horning, and David B. Wortman (1970). *A Compiler Generator*. Englewood Cliffs, NJ: Prentice-Hall.
- Rust Project Developers (2026). *What Bootstrapping Does*. URL: <https://rustc-dev-guide.rust-lang.org/building/bootstrapping/what-bootstrapping-does.html> (visited on 07/21/2026).
- Thompson, Ken (1984). “Reflections on Trusting Trust”. In: *Communications of the ACM* 27.8, pp. 761–763. DOI: [10.1145/358198.358210](https://doi.org/10.1145/358198.358210).
- Wheeler, David A. (2009). “Fully Countering Trusting Trust through Diverse Double-Compiling”. PhD thesis. George Mason University. DOI: [10.48550/arXiv.1004.5534](https://doi.org/10.48550/arXiv.1004.5534). arXiv: [1004.5534](https://arxiv.org/abs/1004.5534).