nC

LANGUAGE DOCUMENTATION

nC Language Guide

Language design, native toolchain setup and the path from source to the machine layer.

Bootstrap edition V81
24 October 2026
github.com/Arizt00/nc

01 Why nC exists
02 Reading the project
03 Setting up the toolchain
04 The first program
05 Places and functions
06 Memory and services
07 The machine layer
08 Working during bootstrap

Why nC exists

A program runs on a concrete machine. Between source and execution, the toolchain chooses data layouts, calling conventions, memory operations, linkage and operating-system services. nC aims to make those decisions expressible within one language and traceable through its compiler.

The language exposes several levels. Ordinary source describes functions, data and control flow. The root `[@]` establishes foundational contracts. `[@machine]` exposes typed instructions and control transfers. `primitive[@]` specifies lower-level material operations. Transitions between these surfaces must retain the information needed to validate each operation.

Toolchain sovereignty means being able to inspect and build the compiler and its dependencies. The compiler written in nC and its successive bootstrap generations serve that objective. Seeds, runtime providers and external tools remain explicit inputs while the build depends on them.

V81 records reconstruction and execution on specific x86-64 profiles. Global Stage3 promotion remains open. Each support claim is limited to its inputs, target and recorded checks.

nC is currently relevant to compiler development, data representation, ABI work and native code generation. Start with small programs and retain the compiler digest used to build them. Memory has manual/raw semantics: the program controls allocation, address validity and release.

Reading the project

The language is **nC**; the compiler is **ncc**. Ordinary source uses `.n`. Machine modules also use `.nm`, and primitive examples use `.ncp`.

The native compiler exposes analysis and emission operations under `compiler`. It participates in bootstrap generations B and C. The retained Stage0 provides the development CLI and routes that the candidate does not yet implement. V81 has incomplete command-line help; use the kit README to identify supported commands.

Component	Responsibility
.n source	Functions, typed places, control flow and imports
Root [@]	Foundational contracts and continuity with material layers
[@machine]	Typed instructions, registers, labels and terminators
primitive[@]	Lower-level material contracts
Runtime	Services and providers required by a program
MCT and backend	Materialization, object formats, ABI and targets

Begin with a complete program, run semantic analysis, then inspect its IR. Native execution additionally requires a target and the corresponding providers. A hardware example may be valid for analysis without being executable on the host machine.

The directories `examples/00_hello`, `examples/01_basics`, `examples/11_machine` and `examples/12_primitive` group examples by subject. Milestones such as V81 identify engineering work; they are not stable language-version numbers.

Setting up the toolchain

Download the V81 native kit from the repository's Releases page. It contains the compiler, runtime provider, examples and SHA-256 checksums. This workflow does not require Rust, Cargo or LLVM.

In PowerShell, from the extracted Windows kit:

```powershell
.\ncc.exe self-test
.\ncc.exe compiler emit-executable hello.n `
  --target x86_64-pc-windows-msvc --out hello.exe
.\hello.exe
```

On Linux x86-64, keep the provider beside the compiler. The kit was tested on Ubuntu 24.04 under WSL. Its imports require glibc 2.34 or later.

```bash
chmod +x ncc
./ncc self-test
./ncc compiler emit-executable hello.n \
  --target x86_64-unknown-linux-gnu --out hello
chmod +x hello
LD_LIBRARY_PATH="$PWD${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}" ./hello
```

Expected output: `Hello, nC public beta`. Expected exit code: `0`. The Stage0 commands `check`, `machine` and `--help` are not available through the same interfaces in this kit.

Compiler and runtime identity

Retain the SHA-256 digest of both artifacts. The compiler determines emitted code; the provider resolves services imported by that code. Replacing either changes the reproduction conditions even when the source is unchanged.

The Linux compiler has RUNPATH `$ORIGIN`. The generated example needs `LD_LIBRARY_PATH` to locate its provider. On Windows, keep `ncruntime.dll` beside the executables. Check exported symbols and ABI before mixing providers from different generations.

Stage0 development is documented in `docs/install.md`. Building it still requires Rust/Cargo, a verified native seed and a compatible archiver. The examples in this guide invoke distributed executables directly.

The first program

The repository retains this source in `examples/00_hello/hello.n`:

```nc
use <io:output>;

helloCompacto {
    to main {
        println("Hello, nC public beta");
    }

    fa:
        return true;
}
```

The import selects the `io` output service. `helloCompacto` names the container. `to main` declares the compact entry point; `fa:` supplies the container final. Braces delimit the container and execution block.

From the extracted Linux or WSL kit:

```bash
./ncc compiler sema hello.n
./ncc compiler emit-ir hello.n
```

In PowerShell:

```powershell
.\ncc.exe compiler sema hello.n
.\ncc.exe compiler emit-ir hello.n
```

`compiler sema` checks the program. `compiler emit-ir` exposes its intermediate representation. Both commands should exit with code 0. IR output describes the program; execution requires emitting and launching a target image as shown in the toolchain setup.

Places and functions

`value >> int <- 0;` declares a typed place and initializes its bits. `>>` separates the name and type; `<-` writes a value. A declaration without an initializer does not guarantee initialized memory.

```nc
counter {
    fn:
        public:
            run() -> bool {
                value >> int <- 0;
                while (value < 3) {
                    value <- value + 1;
                }
                return value == 3;
            }

    ref main ->> run;

    fa:
        return true;
}
```

`fn:` groups functions. `public:` controls visibility, and `ref main ->> run;` selects the entry point. The name `run` has no implicit entry-point privilege. Its `bool` result is produced by the final comparison.

Parameters use declarations such as `value >> int`. Blocks and `if`, `else` and `while` express control flow. Transformations must preserve the evaluation order defined by the source.

Keep examples small enough to verify their result. Add a condition, function or import only after the preceding program works. When a diagnostic appears, retain the smallest source that reproduces it.

The current parser and fixtures define the accepted compact syntax. Older documents and migration tools may retain historical forms; their presence in the repository does not establish current compiler support.

Memory and services

Ordinary nC source has manual/raw memory semantics. Type checking does not add automatic release, ownership tracking or pointer-lifetime validation. The program must satisfy the material preconditions of each operation.

The spellings `T*`, `p<T>` and `ptr<T>` converge on typed pointer representations. Memory examples use operations such as `rd`, `wr` and `off`. Check allocation size, alignment, access type and address lifetime before dereferencing or computing an offset.

A pointer parameter does not implicitly carry allocation length. A function returning allocated storage does not arrange its eventual release. Read each memory API together with its examples and target provider.

The runtime supplies services such as output, files, strings and operating-system access. Availability depends on the compiler, provider and profile. Frontend recognition of a signature does not establish that its symbol can be resolved during linking.

Project modules use compact imports such as:

```
nc
use <project:worker>;
```

Resolution must preserve unit identity and dependencies. Diagnose linkage failures through the project graph and a minimal reproduction. Objects and caches under `.ncc` are derived artifacts, not source files for publication.

The machine layer

`[@machine]` participates in the same typed system. Registers, labels, instructions and terminators carry contracts that must remain coherent through analysis and IR. `examples/11_machine/checked_add_overflow_machine.n` demonstrates addition followed by an overflow branch.

This route requires Stage0. From the repository root on Linux or WSL, invoke the already-built executable directly. On Windows, use `.\target\debug\ncc.exe` with the same arguments. These entry points are not implemented by the V81 native kit.

```bash
./target/debug/ncc check \
  examples/11_machine/checked_add_overflow_machine.n
./target/debug/ncc emit-ir \
  examples/11_machine/checked_add_overflow_machine.n
```

Object-contract `.nm` modules require the machine CLI:

```bash
./target/debug/ncc machine emit-ir \
  examples/11_machine/object_elf64.nm
```

A branch needs a valid destination. A terminator closes an execution path, and the IR must retain that control-flow decision. Accepting an unterminated block where the contract requires termination moves an inconsistency into the backend.

Inspect examples containing `halt`, privileged registers or ISA-specific instructions before attempting execution. They require a compatible target and execution context. Parser acceptance alone does not establish host executability.

`primitive[@]` exposes a different material level. Its relationship to machine instructions, memory and linkage must remain explicit. Duplicating semantics across surfaces makes those contracts harder to verify.

Working during bootstrap

Record the complete command, host, target, executable digest, provider, minimal source and diagnostic for each reproduction. When comparing generations, retain their providers too: a runtime difference can resemble a compiler regression.

Positive fixtures establish accepted behavior and results. Negative fixtures establish rejection and diagnostics. A surface or semantic change must keep parser, AST, semantic analysis, IR, fixtures and documentation aligned.

The V81 dossier records byte-identical B2/C2 compilers within each Windows and Linux/WSL x86-64 host, alongside executed programs, runtime contracts and adversarial checks. Those results apply to the documented profiles; they do not close execution support for every declared architecture.

A useful contribution identifies one behavior: an incorrect example, a missing diagnostic, an unclear contract or a reproducible regression. Include the expected result and the smallest source that demonstrates the issue. Syntax proposals must explain how their meaning survives subsequent compiler phases.

Source references

- `AGENTS.md`: continuity rules and completion criteria.
- `docs/syntax.md`: compact syntax and manual memory.
- `crates/ncc-stage0-bridge/build.rs`: seed requirements.
- `examples/00_hello/hello.n` and `examples/11_machine`.
- `docs/NCC_MODULES_V81_DOSSIER.md` and `docs/NCC_MODULES_V81_REPRO.md`.

The architecture reference describes the pipeline and interpretation of bootstrap evidence. Release notes and installation documentation carry the operational instructions for each published kit.