LANGUAGE MANUAL · NCSCRIPT V1

The ncScript reference manual.

ncScript — called NCIP, the NexaCore Interpreted Program — is the capability-gated, Rust-derived scripting language of NexaCore OS. Source files use the `.ncs` extension. This manual is the complete working reference. The normative specification is NCIP-ncScript-030; the reference implementation is the `nexacore-script` crate, and this manual documents both — marking, throughout, where the two differ today.

How to read this manual

The manual is in four parts. **Part I** is the language: its lexical structure, values, types, expressions, control flow, pattern matching, and error handling. **Part II** is the standard library, function by function. **Part III** covers the capability model, resource limits, and the safety and privacy properties that follow from them. **Part IV** is reference material: the complete grammar, the diagnostics and runtime-error catalogues, the host embedding API, a Rust comparison, a cookbook, and a glossary.

Read the chapters in order the first time; afterwards the manual is a reference — every chapter stands alone, and the standard-library and error chapters are lookup tables.

Status tags

ncScript is one language, but a given feature may run in three different places today. Each example and many sections carry a tag:

- `PLAYGROUND` — runs in the in-browser tutorial and is exercised by its examples.
- `REFERENCE` — runs in the canonical Rust `nexacore-script` interpreter.
- `SPECIFIED` — defined in NCIP-030, with the implementation landing incrementally.

> **Implementation note.** Callouts marked like this record where the current reference interpreter diverges from the specification — a feature parsed but not yet enforced, a simplification, or a deferral. They are not warnings about your code; they tell you what the runtime does today so the manual never overstates what is built. Chapter 40 collects them all.

Part I — The language

The core language: how a program is written, what its values are, and how it computes. Everything here is independent of the standard library and of capabilities, which Parts II and III cover.

1 · Introduction

ncScript is a deliberately simplified derivation of Rust. It keeps the Rust-familiar surface — `let`, `fn`, `match`, `struct` / `enum`, `Result` / `Option`, `?` — so that a developer who knows Rust is productive within minutes, and it removes the parts of Rust that make it slow to write and expensive to interpret: the borrow checker, lifetimes, monomorphised generics, `unsafe`, and macros.

Five design commitments shape the whole language:

- **Capability safety by construction.** A script's ambient authority is empty. It can perform no filesystem, network, AI, config, process, clock, or randomness effect unless it declares the matching capability in a header and the host grants a token. With no header, a script is a pure computation whose entire influence on the world is the value it returns.
- **Value semantics without a borrow checker.** Scalars copy; aggregates are reference-counted with copy-on-write. There are no raw pointers, no manual `free`, and no shared mutable aliasing — so there is no use-after-free and no data race in script-level code, and none of the borrow checker's friction.
- **Typed errors, no exceptions.** Fallible work returns `Result` or `Option`; the `?` operator threads them. There is no non-local control transfer a script can observe other than `return`, `break`, `continue`, and `?`.

- **Gradual typing.** Annotate where it pays; infer or defer elsewhere. A fully untyped one-liner and a fully annotated module interoperate through the `Any` boundary.
- **Auditability over raw speed.** The reference runtime is a tree-walking interpreter, built `no_std` with a small trust base and deterministic resource accounting. A bytecode VM is deferred, not foreclosed.

The language is called NCIP — the NexaCore Interpreted Program — when referring to the language proper; source files carry the `.ncs` extension (NexaCore Script). ncScript is the lingua franca of NexaCore OS: it is the shell's scripting layer, the executable form of agentic automations, the authoring surface for config-as-code, and a generation target for tooling. Because a script's effects are statically declared and deny-by-default, an automation's complete capability footprint can be shown to a user before it runs.

2 · A first program

Every program is a single `.ncs` file. The smallest useful one prints a line:

hello.ncs — PLAYGROUND

```
fn main() {
    print("Hello, world");
}
```

Execution begins at `main`. `print` writes one line to the script's own output stream; it requires no capability, because writing to your own stdout is not a privileged effect. A file may also omit `fn main` entirely, in which case its top-level statements form an implicit `main`, executed in source order after all items are elaborated.

implicit-main.ncs — REFERENCE

```
// No `fn main` — the top-level statements are the program body.
let name = "NexaCore";
print("hello, " + name);
```

A larger first program combines a function, a loop, and a value-returning `match`:

classify.ncs PLAYGROUND

```
fn classify(n) {
    match n {
        0          => "zero",
        p if p < 0 => "negative",
        _          => "positive",
    }
}

fn main() {
    for n in [-2, 0, 5] {
        print(n.to_string() + " is " + classify(n));
    }
}
```

Running a script. Three contexts execute ncScript: the in-browser tutorial (a documented subset, for learning); the reference `nexacore-script` interpreter embedded in a Rust host (Chapter 34); and, in time, the NexaCore OS shell. This manual's examples are written to be read top to bottom; those tagged `PLAYGROUND` can be pasted into the tutorial and run as-is.

3 · Lexical structure

This chapter defines the tokens of ncScript: how source text is broken into comments, identifiers, keywords, and literals before parsing.

3.1 Comments & layout

A line comment runs from `//` to the end of the line. A block comment runs from `/*` to `*/` and **nests**, so you can comment out a region that already contains block comments. Whitespace separates tokens and is otherwise insignificant. Statements are terminated by `;`; the last expression of a block, written without a trailing `;`, is the block's value.

comments.ncs

```
// a line comment
/* a block comment /* may nest */ to here */
let x = 1;   // trailing comment
```

3.2 Identifiers

An identifier matches `[A-Za-z_][A-Za-z0-9_]*` and is not a keyword. The single underscore `_` is special: it is the wildcard token, not an identifier, so it cannot be used as a variable name; identifiers that merely begin with an underscore (such as `_unused`) are ordinary names.

3.3 Keywords

The reserved words are:

RESERVED KEYWORDS

GROUP	KEYWORDS
bindings & items	`let` `mut` `const` `fn` `struct` `enum` `impl` `use`
control flow	`if` `else` `match` `while` `for` `in` `loop` `break` `continue` `return`
concurrency	`scope` `spawn` `await`
misc	`self` `as` `where` `true` `false`

Implementation note. `self` is meaningful only inside an `impl` method. `as` and `where` are reserved and tokenised but the current parser does not use them (no cast or where-clause syntax is accepted yet).

3.4 Integer literals

Decimal integers are a run of digits with optional `_` separators (`1_000_000`). Hexadecimal integers use a `0x` (or `0X`) prefix (`0xFF`, `0xDEAD_BEEF`). All integers are `Int` (signed 64-bit).

> **Implementation note.** Only decimal and `0x` hexadecimal are recognised; there is no binary (`0b`) or octal (`0o`) literal form. An integer literal that does not fit in `i64` is a lex error (`InvalidNumber`).

3.5 Float literals

A float is a run of digits, a `.`, and at least one more digit, with an optional exponent (`1.5`, `3.14159`, `6.022e23`, `1.0e-9`). Underscores are permitted in the digit runs.

> **Implementation note.** The `.` must be followed by a digit, so `5.` is not a float and `5.foo` lexes as the three tokens `Int(5)` `.` `foo` (an integer with a field/method access). Write `5.0` for a float.

3.6 String literals

A string is delimited by double quotes and is UTF-8. The following escape sequences are recognised:

STRING ESCAPES

ESCAPE	MEANING
`\n`	newline (U+000A)
`\t`	tab (U+0009)
`\r`	carriage return (U+000D)
`\\`	backslash
`\"`	double quote
`\0`	NUL (U+0000)

> **Implementation note.** Any other escape is a lex error (`InvalidEscape`). In particular, `\u{…}` Unicode escapes are not supported in string literals (you can paste the literal character instead, since source is UTF-8). The `json` module does accept `\uXXXX` when parsing JSON text — that is the JSON grammar, not ncScript string syntax. An unterminated string is a lex error.

3.7 Booleans, unit, and the absence of char literals

`true` and `false` are the boolean literals; `()` is the unit value. There is no character literal: a single quote begins a loop label (Chapter 10), so `'a'` is a syntax error in value position and `'name` is a label. Use a one-character string (`"a"`) where you would reach for a char.

4 · Values

At run time every expression evaluates to a value. There are eight kinds:

RUNTIME VALUE KINDS

KIND	DESCRIPTION	STORAGE
`Unit`	the empty value `()`	copy
`Bool`	`true` / `false`	copy
`Int`	signed 64-bit integer	copy
`Float`	IEEE-754 binary64	copy
`String`	immutable UTF-8 text	reference-counted
`List`	growable ordered sequence	reference-counted
`Struct`	named record of fields	reference-counted
`Enum`	a named variant with a positional payload	reference-counted

Scalars (the first four) have copy semantics. Aggregates (the last four) are reference-counted: copying a binding copies a shared reference, and the value

is freed deterministically when the last reference drops. Observable behaviour is value semantics with copy-on-write (Chapter 15).

4.1 How values print

`print` and `.to_string()` render a value with its display form:

DISPLAY FORMAT

VALUE	RENDERS AS
`Unit`	`()`
`Bool`	`true` / `false`
`Int(42)`	`42`
`Float(2.5)`	`2.5`
a string	the raw text, with no surrounding quotes
`[1, 2, 3]`	`[1, 2, 3]` (elements comma-space separated)
a struct `P`	`P { x: 3, y: 4 }` (fields in alphabetical order)
`Some(3)`	`Some(3)`
`None`	`None` (no parentheses for a payload-less variant)

> **Implementation note.** An enum displays using only its variant name, not its enum type — `Result::Ok(5)` prints as `Ok(5)`. Struct fields print in `BTreeMap` (alphabetical) order, not declaration order.

4.2 Equality

`==` and `!=` are defined on every pair of values and never raise an error; comparing values of different kinds yields `false`. Within a kind:

- **Unit** equals unit.
- **Bool, Int** compare by value.
- **Float** compares bit-for-bit, so `NaN != NaN` and `0.0 == 0.0` as usual; floats are rarely a good key for equality.
- **String** compares by contents.

- **List** compares by length and element-wise equality (recursively).
- **Enum** compares by enum name, variant, and payload (element-wise).

> **Implementation note.** Structs have no equality rule in the current interpreter: any `==` involving a struct is `false`, even comparing a struct to itself. Compare structs field by field instead.

5 · Types

ncScript's types describe values. Typing is gradual (Chapter 16): annotations are optional and are written after a `:`.

5.1 Primitive types

PRIMITIVE TYPES

TYPE	VALUES	NOTES
`Int`	signed 64-bit integers	arithmetic is checked; overflow is a runtime error
`Float`	IEEE-754 binary64	native arithmetic; no automatic `Int` ↔ `Float` coercion
`Bool`	`true`, `false`	required by `if` / `while` conditions and `&&` / `||`
`String`	UTF-8 text	immutable; concatenate with `+`
`Char`	a Unicode scalar `SPECIFIED`	no char literal yet; use a one-character `String`
`Unit`	`()`	the value of statements and side-effecting expressions

5.2 Compound types

Lists `[T]` are growable, ordered, and homogeneous by convention. They are written with brackets, indexed with `[i]`, and support `.len()`, `.push(x)`, and the `collections` module (Chapter 21).

Tuples `(A, B)` group a fixed number of values. **Maps** `{K: V}` associate keys with values.

> **Implementation note.** In the current interpreter a tuple is represented as a list (a tuple pattern matches a list value), and map literals are not yet implemented — a `{…}` in expression position is parsed as a block, and evaluating a map literal raises a runtime error. Use a list of pairs or a struct in the meantime.

5.3 Option and Result

Two enums are built in and central to the language:

```
prelude

enum Option<T>    { Some(T), None }
enum Result<T, E> { Ok(T), Err(E) }
```

`Option` models a value that may be absent; `Result` models a computation that may fail (Chapter 14). Their constructors `Some`, `None`, `Ok`, `Err` are available without qualification.

5.4 The Any type

A value whose static type is not determined has type `Any`; operations on it are checked at run time. `Any` is the seam that lets untyped and typed code interoperate (Chapter 16). `SPECIFIED`

> **Implementation note.** Type annotations are parsed but the current interpreter performs no static type checking — every operation is checked dynamically. Annotations therefore document intent and prepare for the checker; they do not yet reject a program at load time.

6 · Bindings & mutability

`let` introduces a binding. By default a binding is immutable; `let mut` marks one you intend to reassign. Re-using a name with another `let` shadows the previous binding — it creates a new one rather than mutating the old, and the new binding may even have a different type.

bindings.ncs PLAYGROUND

```
fn main() {
    let x = 10;           // immutable
    let x = x + 1;       // shadows: a new binding, also named x
    let mut total = 0;    // mutable
    total = total + x;   // reassignment
    total                // => 11
}
```

A binding is visible from its declaration to the end of the enclosing block. Inner blocks introduce nested scopes; an inner binding shadows an outer one for the rest of the inner block.

Implementation note. The `mut` marker is parsed but not yet enforced: the current interpreter permits reassigning any in-scope binding. The specification makes assignment to a non-`mut` binding the load-time error `E_IMMUTABLE_ASSIGN`; write `mut` where you intend mutation so your scripts stay correct once the check lands.

6.1 Constants

`const` declares a value fixed at the top level, visible to every function in the unit. Constants are written in `SCREAMING_SNAKE_CASE` by convention.

const.ncs PLAYGROUND

```
const MAX_RETRIES = 3;

fn attempts() { MAX_RETRIES + 1 }

fn main() { attempts() }   // => 4
```

7 · Assignment

Assignment updates an existing binding. It is a statement, not an expression, so it produces no value and cannot appear inside a larger expression. The simple form is `=`; the compound forms apply an arithmetic operator in place:

ASSIGNMENT OPERATORS

OPERATOR	EQUIVALENT
`x = e`	set `x` to `e`
`x += e`	`x = x + e`
`x -= e`	`x = x - e`
`x *= e`	`x = x * e`
`x /= e`	`x = x / e`
`x %= e`	`x = x % e`

The compound forms inherit integer arithmetic's checked semantics: `x += very_large` can raise an overflow error, and `x /= 0` raises a divide-by-zero error (Chapter 8).

> **Implementation note.** The current interpreter only assigns to a simple variable. The specification also allows assigning to a place — a field (`p.x = 1`) or an index (`xs[0] = 9`) of a `mut` binding — but those targets are not yet accepted by the runtime (they raise "only simple variable assignment is supported"). To update an element today, rebuild the list, or use `.push` for growth.

8 · Expressions & operators

Most of an ncScript program is expressions. This chapter gives the operators and their exact semantics; Chapter 30 repeats the table compactly for reference.

8.1 Precedence

From lowest binding strength to highest:

OPERATOR PRECEDENCE (LOW → HIGH)

LEVEL	OPERATORS	ASSOCIATIVITY
logical or	`||`	left, short-circuit
logical and	`&&`	left, short-circuit
comparison	`==` `!=` `<` `<=` `>` `>=`	**non-associative**
additive	`+` `-`	left
multiplicative	`*` `/` `%`	left
unary	`-` `!`	prefix
postfix	`f()` `x[i]` `.field` `.method()` `?` `.await`	left

Comparison is non-associative: `a < b < c` is a syntax error. Write `a < b && b < c`. Assignment is not in this table because it is a statement (Chapter 7).

8.2 Arithmetic

Integer arithmetic is **checked**: `+ - * / %` raise a runtime arithmetic error on overflow, and `/` or `%` by zero raises the same error rather than crashing. Float arithmetic is native IEEE-754 and does not raise — division by zero yields an infinity or `NaN`. There is no implicit conversion between `Int` and `Float`; mixing them in one operator is a type error.

arithmetic.ncs PLAYGROUND

```
fn main() {
    let a = 2 + 3 * 4;      // 14 — * binds tighter than +
    let b = (2 + 3) * 4;    // 20
    let c = 7 % 3;          // 1
    a + b + c                 // => 35
}
```

> **Implementation note.** Binary subtraction is checked, but unary negation `-x` on an `Int` wraps (so `-MIN` does not raise). This asymmetry is a property of the current interpreter.

8.3 Comparison & equality

`< <= > >=` compare two `Int`s or two `Float`s and return a `Bool`. `==` and `!=` work on any pair of values (Chapter 4): they never raise, and unequal kinds compare unequal.

8.4 Boolean operators

`&&` and `||` take booleans and short-circuit: the right operand is evaluated only when needed. `!` negates a boolean. A non-boolean operand is a runtime error.

short-circuit.ncs PLAYGROUND

```
fn main() {
    // the right side (which would divide by zero) is never evaluated
    if false && (1 / 0 == 0) { 1 } else { 2 }   // => 2
}
```

8.5 String concatenation

`+` concatenates two strings into a fresh string (charged against the memory budget, Chapter 27). It is the only operator defined on strings; to repeat a string use `string::repeat`, and to build text from non-strings convert them with `.to_string()` first.

concat.ncs PLAYGROUND

```
fn main() {
    let n = 3;
    "count: " + n.to_string()   // => "count: 3"
}
```

From Rust. ncScript has no `println!` and no `{}` format placeholders. The idiom for `println!("Value: {}", x)` is `print("Value: " + x.to_string())`; each `print` call already emits one line.

9 · Functions

A function is declared with `fn`, a name, a parameter list, an optional return type, and a block body. Parameter and return types may be omitted. The body's last expression — written without a trailing `;` — is the return value; `return e` returns early. A function with no value-producing tail returns `Unit`.

functions.ncs PLAYGROUND

```
fn add(a: Int, b: Int) -> Int { a + b }     // annotated
fn double(x) { x * 2 }                    // inferred
fn log_in(who) { print("welcome, " + who) } // returns Unit

fn main() { add(double(3), 1) }            // => 7
```

9.1 Recursion

Functions may call themselves. Recursion depth is bounded by the call-depth budget (Chapter 27), so unbounded recursion aborts cleanly with a typed error rather than overflowing the host stack.

recursion.ncs PLAYGROUND

```
fn fib(n) {
    if n < 2 { n } else { fib(n - 1) + fib(n - 2) }
}

fn main() { fib(10) }   // => 55
```

9.2 Arity

Calling a function with the wrong number of arguments is an error (the message names the function, the expected count, and the actual count). Functions are not variadic and have no default arguments.

9.3 Generics & methods

Functions and types may be generic (`fn first<T>(xs: [T]) -> Option<T>`); generics are type-erased at run time. Methods and associated functions are declared in an `impl` block (Chapter 12). `SPECIFIED`

> **Implementation note.** The current interpreter registers only top-level `fn` items and ignores generic parameters; `impl` methods are parsed but not yet callable. Write free functions where you would reach for a method today.

10 · Control flow

ncScript distinguishes control-flow expressions (which produce a value) from control-flow statements (which produce `Unit`).

10.1 `if` / `else`

`if` is an expression. The condition must be a `Bool`; the branches are blocks; an `else` is optional, and `else if` chains. When used for its value, both branches should produce the same type.

if.ncs `PLAYGROUND`

```
let sign = if n < 0 { -1 } else if n > 0 { 1 } else { 0 };
```

An `if` with no `else` produces `Unit` when the condition is false, so use it for effects rather than for a value in that form.

10.2 `while` and `for`

`while cond { … }` repeats while the boolean condition holds. `for x in iterable { … }` binds each element in turn. Both are statements and evaluate to `Unit`.

loops.ncs PLAYGROUND

```
fn main() {
    let mut i = 0;
    let mut s = 0;
    while i < 5 { s = s + i; i = i + 1; }   // s => 10

    for x in [10, 20, 30] { s = s + x; }     // s => 70
    s
}
```

Implementation note. `for` iterates over a list. Iterating any other value is a type error today; ranges and string iteration are not built in (build a list explicitly, or use a `while` with an index).

10.3 `loop`, `break`, `continue`

`loop { … }` repeats forever until a `break`. `break` exits the nearest loop; `continue` skips to its next iteration. `break e` yields a value from a `loop`, which makes `loop` an expression. A `loop` may carry a label so an inner `break` can target an outer loop.

loop-value.ncs REFERENCE

```
let mut i = 0;
let first_square_over_50 = loop {
    i = i + 1;
    if i * i > 50 { break i * i; }
};   // => 64
```

Implementation note. Loop labels are accepted only on `loop` (not on labelled `while` / `for`). The in-browser playground supports `break` / `continue` without labels and bounds every loop with the step budget, so even `loop {}` ends cleanly.

11 · Pattern matching

`match` is an expression that tests a value against a sequence of patterns, top to bottom, and evaluates the body of the first arm that matches. Each arm is `pattern => expr` (or `pattern if guard => expr`); arms are separated by commas. Matching binds the names in the chosen pattern for the duration of its body.

11.1 Pattern kinds

PATTERNS

PATTERN	MATCHES
`_`	anything; binds nothing (the wildcard)
`42`, `"hi"`, `true`, `-1`	a value equal to the literal (negative integer literals allowed)
`name`	anything; binds it to `name`
`None`, `Color::Red`	a unit variant (a path with no payload)
`Ok(p)`, `Some(p)`, `Point(a, b)`	a variant, matching each payload position against a sub-pattern
`P { x: a, y: b }`	a struct, matching each named field; `{ x }` shorthand binds the field name
`(a, b)`	a tuple (a list) of matching arity
`a | b`	either alternative (an or-pattern)

11.2 Guards and or-patterns

A guard is a boolean expression attached with `if`; the arm matches only when the pattern matches and the guard is true. An or-pattern matches if any alternative matches.

match.ncs PLAYGROUND

```
fn classify(n) {
    match n {
        0 | 1      => 100,   // or-pattern
        p if p < 0 => -1,    // guard
        p          => p      // binding, catches the rest
    }
}

fn main() { classify(0) + classify(-5) + classify(9) }   // => 108
```

11.3 Matching Result and Option

match-result.ncs PLAYGROUND

```
fn describe(r) {
    match r {
        Ok(v) if v > 0 => "positive ok",
        Ok(_)          => "non-positive ok",
        Err(_)         => "error"
    }
}

fn main() { describe(Ok(7)) }   // => "positive ok"
```

11.4 Exhaustiveness

The specification requires a `match` to be exhaustive — every variant covered or a wildcard present — checked at load time (`E_NONEXHAUSTIVE_MATCH`).

> **Implementation note.** The current interpreter does not check exhaustiveness statically; instead, a value that matches no arm is the runtime error "no match arm covered the value". Include a final `_` arm to be safe.

12 · Structs

A struct is a named record. It is declared with field names and (optionally) types, constructed with a struct literal, and read with dot access.

struct.ncs PLAYGROUND

```
struct Point { x: Int, y: Int }

fn main() {
    let p = Point { x: 3, y: 4 };
    p.x + p.y    // => 7
}
```

Field shorthand. When a local variable has the same name as a field, `{ x }` is shorthand for `{ x: x }`.

shorthand.ncs REFERENCE

```
fn make(name, addr) { Host { name, addr } }   // shorthand for { name: name, addr: addr }
```

12.1 Methods

Structs (and enums) carry methods and associated functions in an `impl` block. A method takes `self` as its first parameter; an associated function does not and is called on the type.

impl.ncs REFERENCE

```
impl Point {
    fn origin() -> Point { Point { x: 0, y: 0 } }     // associated fn
    fn norm2(self) -> Int { self.x * self.x + self.y * self.y }
}
```

Implementation note. Struct definitions, literals, field access, and the field shorthand run in the reference interpreter; the in-browser playground supports literals and field access. `impl` methods are parsed but not yet callable, and two structs never compare equal under `==` (Chapter 4). Compare field by field, and use free functions instead of methods for now.

13 · Enums

An enum is a named sum type: a value is exactly one of its variants. A variant may be a unit (no payload), a tuple (positional payload), or a struct (named fields).

enum.ncs REFERENCE

```
enum Shape {
    Circle(Float),
    Rect { w: Float, h: Float },
    Empty
}

fn kind(s) {
    match s {
        Shape::Circle(_) => "circle",
        Shape::Rect { .. } => "rect",
        Shape::Empty => "empty"
    }
}
```

Construct a variant by naming it (`Shape::Empty` , `Shape::Circle(1.0)`) and match it with a variant pattern. The built-in `Result` and `Option` are enums; their constructors `Ok` , `Err` , `Some` , `None` need no qualification.

Implementation note. The interpreter has no nominal enum registry: a call such as `Color::Red(1)` simply produces an enum value tagged with that name and variant — there is no check that `Color` was declared or that `Red` is one of its variants. Matching and equality compare the enum name, the variant, and the payload structurally. Custom enums therefore run in the reference interpreter; the in-browser playground provides the built-in `Result` / `Option` constructors and matching.

14 · Error handling

ncScript has no exceptions and no stack unwinding a script can observe. Anything that can fail returns a value: `Result<T, E>` for an operation that can error, `Option<T>` for one that can be absent.

14.1 The ? operator

`expr?` inspects a `Result` or `Option`: on `Ok(v)` or `Some(v)` it evaluates to `v`; on `Err(e)` or `None` it returns that `Err` / `None` from the enclosing function immediately. It is the concise way to thread failures up a call chain without nesting `match`.

try.ncs PLAYGROUND

```
fn get(ok) { if ok { Ok(7) } else { Err(42) } }

fn use_it(ok) {
    let v = get(ok)?;   // returns Err(42) early when ok is false
    Ok(v + 1)
}

fn main() {
    match use_it(true) { Ok(x) => x, Err(e) => e }   // => 8
}
```

The specification requires the enclosing function to return `Result` (resp. `Option`) — using `?` elsewhere is `E_QUESTION_CONTEXT` — and, when the error types differ, a `From` conversion to exist (else `E_ERROR_CONVERT`).

14.2 Typed errors

An error is any value; idiomatically it is a user `enum` implementing the built-in `Error` trait, whose single method `fn message(self) -> String` renders it for a human. The standard library defines `IoError`, `NetError`, `AiError`, `ConfigError`, and `ParseError`. A capability denial surfaces as the typed, recoverable error `CapabilityError::Denied(cap)` in `Err` — never as a crash, so a script that lacks a capability can handle the failure.

typed-error.ncs SPECIFIED

```
enum ScanError { Io(IoError), Empty }
impl Error for ScanError {
    fn message(self) -> String { "scan failed" }
}
```

14.3 panic

`panic(msg)` terminates the script for an unrecoverable invariant violation. It is not catchable and is a clean, bounded termination — the host's state is never corrupted. Reserve it for "this should be impossible", not for control flow; ordinary failures belong in `Result`. SPECIFIED

> **Implementation note.** `Result` / `Option`, the `?` operator, and matching on them run today. The `Error` trait, the standard error enums, `From` conversions for `?`, and `panic` are specified and land with the trait/impl machinery; the in-browser playground threads `Result` / `Option` through `?` and surfaces `CapabilityDenied` for out-of-scope effects.

15 · Memory model

ncScript has no borrow checker and no lifetimes. Its memory model gives predictable value semantics by a different route:

- **Scalars copy.** `Int`, `Float`, `Bool`, and `Unit` are copied on assignment and argument passing.
- **Aggregates are reference-counted.** Strings, lists, structs, and enum payloads carry a shared reference; assignment and argument passing copy the reference and increment a count; the value is freed deterministically when the last reference drops.
- **Observable behaviour is value semantics with copy-on-write.** A mutation of a `mut` binding that is shared behaves as if the aggregate were copied first, so it never affects another binding that observed the prior value. The copy may be elided when the reference count is one.

There are no raw pointers, no manual `free`, no `unsafe`, and no shared mutable aliasing across tasks — so script-level code has no use-after-free and no data races by construction.

The runtime accounts memory live: every aggregate carries a guard that charges its bytes against the script's memory budget on creation and credits them back when it drops (Chapter 27). This is what makes the memory limit deterministic.

> **Implementation note.** Reference counting alone cannot reclaim reference cycles. The specification mandates a bounded cycle collector to backstop them; the current interpreter implements reference counting with live accounting, and the cycle collector is a documented follow-up. Acyclic data — the overwhelming common case in scripts — is reclaimed immediately and deterministically.

16 · Gradual typing

Typing is gradual: every binding, parameter, and return position may carry an explicit type or omit it. Omitted annotations are inferred by a local, Hindley–Milner-style pass whose boundary is the function signature; an unconstrained position falls back to `Any`. Integer literals default to `Int`, float literals to `Float`.

`Any` is the gradual boundary. Operations where both sides are statically typed are checked statically and are sound — a program that type-checks does not exhibit a static type error there at run time. Operations involving `Any` are checked dynamically, surfacing a mismatch as the typed error `E_TYPE`. Generics are type-erased (a uniform representation), consistent with an interpreter and with the reference-counted aggregate model.

When to annotate. Annotate public function signatures and anything whose type a reader could not infer at a glance; leave throwaway locals to inference. The type checker never requires an annotation; it only uses the ones you give.
`SPECIFIED`

> **Implementation note.** The current interpreter does no static type checking; every operation is checked dynamically and a mismatch is a runtime type error. Annotations are parsed and ignored. The semantics your script observes (typed runtime errors) match the dynamic side of the gradual model; the static checker and load-time `E_TYPE` land with the type system.

17 · Structured concurrency

Concurrency in ncScript is structured: every concurrent task has a parent `scope`, and a scope does not complete until all the tasks it spawned have completed or been cancelled. There are no detached tasks and no global thread pool a script can reach.

- **`scope { … }`** delimits a concurrency region.
- **`spawn expr`**, inside a scope, starts `expr` as a concurrent task and returns a `Task<T>` handle.
- **`task.await`** yields the task's `Result<T, E>`; it is the only way to observe a task's outcome.
- **Cancellation.** If any task in a scope returns `Err` (or panics), the scope may cancel its siblings and propagates the first error out as the scope's result — "abort on first error". A `scope.all()` form instead collects every result without early cancellation.
- **Data sharing.** Tasks may only move values in or share immutable aggregates, so there are no data races. Inter-task communication uses stdlib channels that move values between tasks.
- **No ambient time.** `sleep` and timers are capability-mediated (the `time` capability), not free functions.

scope.ncs — SPECIFIED

```
scope {
    let a = spawn fetch("https://a.example/x");   // needs a net capability
    let b = spawn fetch("https://b.example/y");
    let (ra, rb) = (a.await?, b.await?);          // first Err aborts the
scope
    combine(ra, rb)
}
```

Implementation note. `scope`, `spawn`, and `.await` parse today but run inline and synchronously in the current interpreter — `spawn expr` evaluates `expr` immediately and `scope` behaves like a block. The structured-concurrency semantics above are the specified target; write code against them and it will keep working as the concurrent runtime lands.

Part II — Standard library

The standard library is pure computation: every function here is effect-free and needs no capability. Effectful operations — the filesystem, the network, AI inference, config, time, randomness — are host bindings gated by capabilities, summarised in Chapter 24 and specified in Part III. A call `module::function(args)` reaches one of five modules; an unknown function in a known module is the error "no method `module::function`".

18 · Built-ins

A handful of functions and methods are always available, with no module prefix.

18.1 Functions

GLOBAL BUILT-IN FUNCTIONS PLAYGROUND

SIGNATURE	BEHAVIOUR
`print(args…)`	renders every argument with its display form, joins them with a single space, writes one line to the script's output, and returns `Unit`. Needs no capability.
`len(x)`	the length of a list (element count) or a string (Unicode-scalar count, not bytes). A type error on anything else.

18.2 Methods

BUILT-IN METHODS PLAYGROUND

METHOD	RECEIVER	BEHAVIOUR
`.len()`	list or string	element count / character count
`.push(x)`	list	appends `x` in place (charged to the memory budget); returns `Unit`
`.to_string()`	any value	the value's display form as a string

builtins.ncs PLAYGROUND

```
fn main() {
    let xs = [10, 20];
    xs.push(30);
    print("len = " + xs.len().to_string());   // prints "len = 3"
    len("héllo")                               // => 5 (characters, not
bytes)
}
```

Implementation note. `print` accepts several arguments and space-joins them; the worked examples in this manual mostly pass a single pre-built string for clarity. `.push` is the only built-in that mutates; list element assignment (`xs[0] = …`) is not yet available (Chapter 7).

19 · Module `string`

Pure operations on UTF-8 strings. Indices and lengths count Unicode scalar values, not bytes.

STRING PLAYGROUND

FUNCTION	RETURNS	DESCRIPTION
`string::len(s)`	Int	number of characters
`string::upper(s)`	String	upper-cased copy
`string::lower(s)`	String	lower-cased copy
`string::trim(s)`	String	copy with leading/trailing whitespace removed
`string::contains(s, sub)`	Bool	whether `sub` occurs in `s`
`string::starts_with(s, p)`	Bool	whether `s` begins with `p`
`string::ends_with(s, p)`	Bool	whether `s` ends with `p`
`string::replace(s, from, to)`	String	every occurrence of `from` replaced by `to`
`string::split(s, sep)`	[String]	the pieces of `s` between occurrences of `sep`
`string::repeat(s, n)`	String	`n` copies of `s` (the projected size is reserved against the memory budget first; `n < 0` yields the empty string)
`string::from_int(i)`	String	the decimal rendering of an integer
`string::to_int(s)`	Option<Int>	`Some(n)` if `s` (trimmed) parses as an integer, else `None`

string.ncs PLAYGROUND

```
fn main() {
    let parts = string::split("a,b,c", ",");   // ["a", "b", "c"]
    print(string::upper("done"));               // "DONE"
    print(string::replace("a.b.c", ".", "-"));  // "a-b-c"
    match string::to_int("  42 ") {
        Some(n) => n + parts.len().to_string().len(),   // 42 + 1
        None    => -1
    }   // => 43
}
```

20 · Module `math`

Pure integer arithmetic helpers. Every function except `abs` is integer-only; `abs` also accepts a float. There are no transcendental functions (no `sin`, `log`, …) — the module is `no libm` by design.

MATH PLAYGROUND

FUNCTION	RETURNS	DESCRIPTION / ERRORS
`math::abs(x)`	Int or Float	absolute value; an integer `abs` of the minimum `Int` raises an overflow error
`math::min(a, b)`	Int	the smaller of two integers
`math::max(a, b)`	Int	the larger of two integers
`math::pow(base, exp)`	Int	integer power; `exp < 0` errors; overflow errors
`math::gcd(a, b)`	Int	greatest common divisor (of the absolute values)
`math::isqrt(n)`	Int	integer square root (floor); a negative `n` errors

math.ncs PLAYGROUND

```
fn main() {
    math::pow(2, 10) + math::gcd(48, 36) + math::isqrt(99)
    // 1024 + 12 + 9 => 1045
}
```

21 · Module `collections`

Pure, non-mutating operations on lists. Queries that may not find a result return an `Option` instead of raising, and transforms return a fresh list, leaving the argument untouched.

COLLECTIONS PLAYGROUND

FUNCTION	RETURNS	DESCRIPTION
`collections::len(l)`	Int	element count
`collections::is_empty(l)`	Bool	whether the list has no elements
`collections::get(l, i)`	Option<T>	the element at `i`, or `None` if out of range
`collections::first(l)`	Option<T>	the first element, or `None`
`collections::last(l)`	Option<T>	the last element, or `None`
`collections::contains(l, x)`	Bool	whether `x` is an element (by value equality)
`collections::index_of(l, x)`	Option<Int>	the position of the first `x`, or `None`
`collections::reverse(l)`	[T]	a new list in reverse order
`collections::concat(a, b)`	[T]	a new list, `a` followed by `b`
`collections::slice(l, start, end)`	[T]	a new list of the elements in `[start, end)`; indices are clamped to the list, and an inverted range yields the empty list

collections.ncs PLAYGROUND

```
fn main() {
    let xs = [10, 20, 30, 40];
    let mid = collections::slice(xs, 1, 3);   // [20, 30]
    match collections::index_of(xs, 30) {
        Some(i) => i + mid.len(),    // 2 + 2
        None    => -1
    }   // => 4
}
```

22 · Module `json`

Serialise values to JSON text and parse JSON text into values.

JSON REFERENCE

FUNCTION	RETURNS	DESCRIPTION
`json::stringify(value)`	String	JSON text for the value
`json::parse(text)`	Option<value>	`Some(value)` on success, `None` on malformed input

22.1 The value mapping

HOW VALUES MAP TO JSON (STRINGIFY)

NCSCRIPT	JSON
`Unit`	`null`
`Bool`	`true` / `false`
`Int`	an integer number
`Float`	a number (an integral float keeps a `.0`); `NaN`/infinity cannot be represented and error
`String`	a quoted, escaped string
`List`	an array
`Struct`	an object (the type name is dropped; fields in alphabetical order)
`Enum`	has no JSON representation and errors

Parsing is the inverse, with two specifics: a JSON object becomes an anonymous struct (named `"object"`), and an integral number in `i64` range becomes an `Int` while anything else becomes a `Float`. Parsing rejects trailing non-whitespace after the top-level value (returning `None`) and is depth-limited to 128 levels of nesting so a hostile document cannot overflow the host stack. The parser accepts the standard JSON escapes, including `\uXXXX` with surrogate pairs.

json.ncs — REFERENCE

```
fn main() {
    let text = json::stringify([1, 2, 3]);   // "[1,2,3]"
    match json::parse("[10, 20]") {
        Some(v) => v,        // the list value [10, 20]
        None    => []
    }
}
```

23 · Module `datetime`

Pure calendar arithmetic on Unix timestamps — integers counting seconds since 1970-01-01T00:00:00Z, in the proleptic Gregorian calendar. Every function takes the timestamp as an argument; there is no ambient clock, because reading "now" is a capability-gated effect (the `time` capability), not part of this pure module.

DATETIME **REFERENCE**

FUNCTION	RETURNS	DESCRIPTION
`datetime::year(ts)`	Int	calendar year
`datetime::month(ts)`	Int	month, 1–12
`datetime::day(ts)`	Int	day of month, 1–31
`datetime::hour(ts)`	Int	hour, 0–23
`datetime::minute(ts)`	Int	minute, 0–59
`datetime::second(ts)`	Int	second, 0–59
`datetime::weekday(ts)`	Int	day of week, 0=Sunday … 6=Saturday
`datetime::format_iso(ts)`	String	`YYYY-MM-DDTHH:MM:SSZ`
`datetime::from_ymd(y, m, d)`	Int	the timestamp at midnight UTC of that date; an out-of-range month or day errors
`datetime::add_seconds(ts, n)`	Int	`ts + n` seconds (overflow errors)
`datetime::add_days(ts, n)`	Int	`ts + n` days (overflow errors)

datetime.ncs REFERENCE

```
fn main() {
    let ts = datetime::from_ymd(2026, 6, 24);
    let tomorrow = datetime::add_days(ts, 1);
    datetime::format_iso(tomorrow)   // => "2026-06-25T00:00:00Z"
}
```

24 · Host effects

Everything in Chapters 18–23 is pure. The effectful surface — reading a file, opening a connection, invoking a model, reading config, telling the time, drawing randomness — is provided by the host, reached through namespaced calls (`fs::read` , `net::connect` , `ai::invoke` , …) and gated by capabilities. The language guarantees that without the matching capability these calls cannot occur; Part III specifies the model in full. The effect families are:

EFFECT FAMILIES AND THEIR CAPABILITIES

NAMESPACE	CAPABILITY	EFFECT
`fs::`	`fs.read(path)` / `fs.write(path)`	read / write files under a path
`net::`	`net.connect(host)` / `net.listen(port)`	outbound connections / accept connections
`ai::`	`ai.invoke`	call AI-runtime syscalls (invoke / embed / classify)
`config::`	`config.read(ns)` / `config.write(ns)`	read / write config under a namespace
`proc::`	`proc.spawn`	spawn host processes / IPC
time / rand	`time` / `rand`	read the clock, sleep, timers / draw from the CSPRNG

A failed effect is a value, not a crash: a denied capability returns `Err(CapabilityError::Denied(..))` , and an I/O failure returns the relevant typed error (`IoError` , `NetError` , …) in `Err` .

> **Implementation note.** The host effect set is provided by whichever host embeds the interpreter (Chapter 34): the host declares which `namespace::function` pairs it implements and which capability each needs. The in-browser tutorial installs a mock host for `fs` / `net` / `ai` / `agent` so the capability gate and its scoping are demonstrable without real I/O. With no host installed, a namespaced call that is neither a stdlib function nor a declared effect is treated as an enum constructor (it produces an inert value) — the language's "no ambient authority" guarantee.

Part III — Capabilities & safety

The properties that make ncScript safe to run untrusted code: a deny-by-default capability model, deterministic resource budgets, and the security and privacy guarantees that follow from them.

25 · The capability model

A script's ambient authority is empty. With no capabilities it is a pure computation: it can calculate, allocate, and return a value, but it can perform no observable side effect. Authority is added only by declaring capabilities (Chapter 26) and having the host grant matching tokens.

25.1 Three outcomes, one rule

Deny-by-default produces three distinct outcomes for an effect:

- **Undeclared.** An effect whose capability is not in the header is rejected at load time (`E_CAP_UNDECLARED`) — the script does not run.
- **Declared but not granted.** The script loads and runs, but the effect call returns `Err(CapabilityError::Denied(cap))` — a typed, recoverable failure, never a crash.
- **Declared and granted.** The effect runs.

Because the footprint is in the header, tooling can show a user exactly what a script may do before granting anything.

25.2 The lattice

THE V1 CAPABILITY LATTICE

CAPABILITY	GRANTS	SCOPE
`fs.read(path)`	read files/dirs under a path	path
`fs.write(path)`	create / modify / delete under a path	path
`net.connect(host)`	open outbound connections	host (and port)
`net.listen(port)`	bind and accept	port
`ai.invoke`	call AI-runtime syscalls	unscoped
`config.read(ns)` / `config.write(ns)`	read / write config keys	namespace
`proc.spawn`	spawn host processes / IPC	unscoped
`time`	read the clock, sleep, set timers	unscoped
`rand`	draw from the host CSPRNG	unscoped

The lattice is extensible by future revisions; v1 fixes these classes.

25.3 Scope matching and attenuation

A capability has a name and an optional scope. A grant covers a use when the names match exactly and either the grant is unscoped, or the use's argument equals the scope, or it lies under the scope at a path boundary. Concretely, a scoped grant for `S` covers an argument `A` when `A == S` or `A` starts with `S` followed by a `/`.

SCOPE MATCHING EXAMPLES FOR FS.READ("/ETC/NEXACORE")

ARGUMENT	COVERED?	WHY
`/etc/nexacore`	yes	equals the scope
`/etc/nexacore/notes/a.txt`	yes	under the scope at a `/` boundary
`/etc/passwd`	no	different path
`/etc/nexacore-secret`	no	not at a `/` boundary — prefix alone is insufficient

This is attenuation: a grant for `/etc/nexacore` does not imply `/etc` or `/`. A host may grant a narrower scope than the script declares, but never a wider one, and never a capability the header omits. A script cannot fabricate, widen, or forward a capability — it holds only the right to attempt an effect, mediated by the runtime.

> **Implementation note.** In the interpreter, the scope checked is the first string argument of the effect call. The declared header is recorded (and readable via the embedding API) but is informational: the interpreter does not auto-grant the header — actual authority comes solely from the grants the host installs. This is the language/runtime split: the header is the contract a tool reads; the host's grants are the enforcement.

26 · The capability header

A `.ncs` file may begin (after an optional shebang) with a capability header: a `#![capabilities(...)]` attribute that is the first non-comment construct. It declares the maximal effect set the script may exercise. With no header, the granted set is empty and the script is pure.

header.ncs PLAYGROUND

```
#![capabilities(
    fs.read("/etc/nexacore"),
    fs.write("/var/nexacore/out"),
    net.connect("api.nexacore.example:443"),
    ai.invoke,
    config.read("ui.theme"),
)]
```

Each entry is a capability literal: a dotted name optionally followed by a parenthesised scope, which is a string (a path, host, or namespace) or an integer (a port). The names and scope forms are exactly those of the lattice (Chapter 25). The header is:

- **Binding.** The effect checker verifies that the union of effects the program uses is a subset of the declared set; an undeclared effect is `E_CAP_UNDECLARED`.
- **Least-privilege-nudging.** Declaring a capability the program never uses is a warning (`W_CAP_UNUSED`), not an error — you may over-declare for forward compatibility, but you are nudged toward the minimum.
- **Machine-readable.** Tooling extracts the declared set by parsing the header alone, without executing the script. This is how an automation's effects are shown to a user before it runs.

In the in-browser tutorial, change the path in the capabilities lesson from an in-scope path to `"/etc/passwd"` and run again to watch the gate reject it with `CapabilityDenied` before any read occurs.

27 · Resource limits

The runtime executes every script under deterministic budgets and aborts cleanly when one is exceeded — a typed error, not a hang or a host crash. Budgets are configured by the host (Chapter 34); each defaults to unlimited when unset.

THE FOUR BUDGETS

BUDGET	FIELD	ON EXCEED	ENFORCED
instruction / steps	`max_steps`	`LimitExceeded(Steps)`	once per evaluated expression, and once per loop iteration — so even `loop {}` aborts
live memory	`max_alloc_bytes`	`LimitExceeded(Memory)`	at every aggregate allocation; credited back on drop
wall-clock	`deadline_micros`	`LimitExceeded(Time)`	against an injected monotonic clock
call depth	`max_call_depth`	`LimitExceeded(CallDepth)`	before each function call descends

The memory budget is live, not cumulative: each aggregate charges its bytes on creation and credits them back deterministically when its last reference drops, so a script that builds and discards data repeatedly is bounded by its peak, not its total. The step budget makes infinite loops safe; the call-depth budget turns unbounded recursion into a clean abort rather than a host stack overflow.

host budget · Rust — REFERENCE

```rust
let limits = Limits {
    max_steps:       Some(1_000_000),  // instruction budget
    max_alloc_bytes: Some(64_000),     // live-memory ceiling
    deadline_micros: Some(5_000_000),  // wall-clock deadline (needs a Clock)
    max_call_depth:  Some(256),         // recursion guard
};
let (value, output) = run_with_limits(src, limits)?;
```

Implementation note. The step and call-depth budgets are enforced by `run_with_limits` and by the in-browser tutorial. The wall-clock deadline additionally requires the host to attach a `Clock` (build an `Interpreter` and call `with_clock`); without one, `deadline_micros` is inert.

28 · Security & privacy

ncScript exists as a bespoke language, rather than an embedded Lua or Python, for one reason: those languages have ambient authority — `open`, `socket`, `fetch` are reachable by default — which is exactly what NexaCore OS's threat model forbids. The properties below follow structurally from the model in this Part.

- **Bounded blast radius.** The primary adversary is a malicious or buggy script — a hallucinating agent's automation, a third-party extension, a marketplace package. A script with no header is a pure computation; its entire influence is the value it returns. It cannot touch the filesystem, network, AI runtime, config, processes, clock, or RNG.
- **Auditable before execution.** The capability footprint is extractable by parsing the header alone, so a user (or a dashboard) sees the complete set of effects an automation may perform before approving it. There is no hidden ambient authority to surprise them.
- **Defense in depth.** Language-level effect typing (load-time `E_CAP_UNDECLARED`) is backed by a runtime token check at the syscall boundary: even a runtime that mis-typed an effect cannot perform it without a host-granted token, and a declared-but-not-granted capability fails typed and recoverable.
- **Bounded resource use.** Deterministic CPU, memory, time, and call-depth budgets (Chapter 27) abort a runaway or hostile script cleanly, with no host-state corruption — including `panic`, which is a bounded termination.
- **Memory safety.** No raw pointers, no `unsafe`, no manual `free`, no shared mutable aliasing across tasks — so no use-after-free and no data races in script-level code.
- **Capability non-amplification.** Scripts cannot fabricate, widen, or forward capabilities; hosts may only attenuate, never amplify, and never grant an undeclared capability — so there is no privilege escalation through delegation.
- **No telemetry.** The language defines no implicit telemetry or phone-home behaviour; a conforming script performs only the effects its capabilities permit. Scoped, attenuable capabilities (a grant for one directory does not reach another) give data-minimisation and purpose-limitation at the language level.

Part IV — Reference

Lookup material: the complete grammar, the operator and diagnostics tables, the runtime-error and lexer/parser-error

catalogues, the host embedding API, a Rust comparison, the style guide, a cookbook, a glossary, the feature-status matrix, and the consolidated implementation notes.

29 · Grammar (EBNF)

The normative concrete syntax of ncScript v1, reproduced from NCIP-030 §S13. Notation: `=` defines a rule; `|` alternation; `[…]` optional; `{…}` repetition (zero or more); `(…)` grouping; terminals in quotes; `(* … *)` comments. Whitespace and comments may appear between any two tokens.

ncscript.ebnf NORMATIVE

```
(* ---- Compilation unit ---- *)
compilation_unit = [ shebang ] , [ capability_header ] , { item } ,
{ statement } ;
shebang          = "#!" , ?characters up to end of first line? ;

(* ---- Capability header ---- *)
capability_header = "#![" , "capabilities" , "(" ,
                        [ cap_decl , { "," , cap_decl } , [ "," ] ] ,
                    ")" , "]" ;
cap_decl          = cap_name , [ "(" , cap_scope , ")" ] ;
cap_name          = identifier , { "." , identifier } ;
cap_scope         = string_lit | int_lit ;

(* ---- Items ---- *)
item        = fn_item | struct_item | enum_item | const_item | impl_item |
use_item ;
use_item    = "use" , path , ";" ;
path        = identifier , { "::" , identifier } ;
const_item  = "const" , identifier , ":" , type , "=" , expr , ";" ;
fn_item     = "fn" , identifier , [ generics ] , "(" , [ param_list ] , ")" ,
              [ "->" , type ] , block ;
param_list  = param , { "," , param } , [ "," ] ;
param       = ( "self" ) | ( identifier , [ ":" , type ] ) ;
generics    = "<" , identifier , { "," , identifier } , ">" ;
struct_item = "struct" , identifier , [ generics ] ,
              ( "{" , [ field_def , { "," , field_def } , [ "," ] ] , "}" |
";" ) ;
field_def   = identifier , ":" , type ;
enum_item   = "enum" , identifier , [ generics ] ,
              "{" , [ variant , { "," , variant } , [ "," ] ] , "}" ;
variant     = identifier ,
              [ "(" , type , { "," , type } , ")"
              | "{" , field_def , { "," , field_def } , "}" ] ;
impl_item   = "impl" , [ identifier , "for" ] , type , "{" , { fn_item } ,
"}" ;

(* ---- Types ---- *)
type        = type_atom , { "?" } ;
type_atom   = path , [ "<" , type , { "," , type } , ">" ]
            | "[" , type , "]"
            | "{" , type , ":" , type , "}"
            | "(" , [ type , { "," , type } ] , ")" ;

(* ---- Statements ---- *)
```

```
block       = "{" , { statement } , [ expr ] , "}" ;
statement   = let_stmt | expr_stmt | assign_stmt | while_stmt | for_stmt |
item ;
let_stmt    = "let" , [ "mut" ] , pattern , [ ":" , type ] , [ "=" , expr ] ,
";" ;
expr_stmt   = expr , ";" ;
assign_stmt = place , assign_op , expr , ";" ;
assign_op   = "=" | "+=" | "-=" | "*=" | "/=" | "%=" ;
place       = identifier , { "." , identifier | "[" , expr , "]" } ;
while_stmt  = "while" , expr , block ;
for_stmt    = "for" , pattern , "in" , expr , block ;

(* ---- Expressions (lowest to highest precedence) ---- *)
expr        = or_expr ;
or_expr     = and_expr , { "||" , and_expr } ;
and_expr    = cmp_expr , { "&&" , cmp_expr } ;
cmp_expr    = add_expr , [ ( "==" | "!=" | "<" | "<=" | ">" | ">=" ) ,
add_expr ] ;
add_expr    = mul_expr , { ( "+" | "-" ) , mul_expr } ;
mul_expr    = unary_expr , { ( "*" | "/" | "%" ) , unary_expr } ;
unary_expr  = ( "-" | "!" ) , unary_expr | postfix_expr ;
postfix_expr= primary_expr ,
              { "." , identifier            (* field / method receiver *)
              | "." , identifier , call_args (* method call *)
              | call_args                    (* function call *)
              | "[" , expr , "]"             (* index *)
              | "?"                          (* try operator *)
              | ".await"                     (* await a Task *)
              } ;
call_args   = "(" , [ expr , { "," , expr } , [ "," ] ] , ")" ;
primary_expr= literal | path | struct_lit | list_lit | map_lit
            | tuple_or_group | if_expr | match_expr | loop_expr
            | scope_expr | spawn_expr | block ;
struct_lit  = path , "{" , [ field_init , { "," , field_init } , [ "," ] ] ,
"}" ;
field_init  = identifier , [ ":" , expr ] ;        (* shorthand: { name } *)
list_lit    = "[" , [ expr , { "," , expr } , [ "," ] ] , "]" ;
map_lit     = "{" , [ map_entry , { "," , map_entry } , [ "," ] ] , "}" ;
map_entry   = expr , ":" , expr ;
tuple_or_group = "(" , [ expr , { "," , expr } , [ "," ] ] , ")" ;
if_expr     = "if" , expr , block , [ "else" , ( if_expr | block ) ] ;
match_expr  = "match" , expr , "{" , match_arm , { "," , match_arm } ,
[ "," ] , "}" ;
match_arm   = pattern , [ "if" , expr ] , "=>" , ( expr | block ) ;
loop_expr   = [ label , ":" ] , "loop" , block
```

```
            | [ label , ":" ] , "while" , expr , block
            | [ label , ":" ] , "for" , pattern , "in" , expr , block ;
label       = "'" , identifier ;
scope_expr  = "scope" , block ;
spawn_expr  = "spawn" , expr ;
flow_expr   = "return" , [ expr ]
            | "break" , [ label ] , [ expr ]
            | "continue" , [ label ] ;

(* ---- Patterns ---- *)
pattern     = "_" | literal | identifier
            | path , [ "(" , pattern , { "," , pattern } , ")" ]
            | path , "{" , field_pat , { "," , field_pat } , [ "," ] , "}"
            | "(" , pattern , { "," , pattern } , ")"
            | pattern , "|" , pattern ;
field_pat   = identifier , [ ":" , pattern ] ;

(* ---- Lexical ---- *)
literal     = int_lit | float_lit | string_lit | char_lit | bool_lit |
unit_lit ;
int_lit     = digit , { digit | "_" } ;
float_lit   = digit , { digit | "_" } , "." , digit , { digit | "_" } ;
bool_lit    = "true" | "false" ;
unit_lit    = "(" , ")" ;
string_lit  = '"' , { ?char except " or backslash? | escape } , '"' ;
identifier  = ( letter | "_" ) , { letter | digit | "_" } ;   (* not a
keyword *)
```

The current parser implements the bulk of this grammar; Chapter 40 lists the specific deviations (hex-only integers, no `\u{}` in strings, map literals deferred, labels on `loop` only, `as` / `where` unused).

30 · Operator reference

ALL OPERATORS

OPERATOR	ARITY	OPERANDS → RESULT	NOTES
`+`	binary	Int,Int→Int · Float,Float→Float · String,String→String	checked for Int; concatenation for String
`-` `*`	binary	Int,Int→Int · Float,Float→Float	checked for Int
`/` `%`	binary	Int,Int→Int · Float,Float→Float	Int by-zero errors; Int truncates
`==` `!=`	binary	any,any→Bool	never raises; different kinds → not equal; structs never equal
`<` `<=` `>` `>=`	binary	Int,Int→Bool · Float,Float→Bool	non-associative
`&&` `||`	binary	Bool,Bool→Bool	short-circuit
`-` (unary)	unary	Int→Int · Float→Float	Int negation wraps
`!`	unary	Bool→Bool	
`?`	postfix	Result/Option→inner	early-returns Err/None
`.field` `.method()` `[i]` `()`	postfix	—	field access, method/function call, index

Mixing an `Int` and a `Float` in one binary operator, or applying a comparison to non-numbers, is a type error — there is no implicit numeric coercion.

31 · Diagnostics

The normative load-time diagnostics. `E_*` are errors (the unit does not run); `W_*` are warnings (it runs).

DIAGNOSTICS CATALOGUE (NCIP-030 §S10)

CODE	PHASE	MEANING
`E_PARSE`	parse	input does not match the grammar
`E_IMMUTABLE_ASSIGN`	check	assignment to a non-`mut` binding or place
`E_NONEXHAUSTIVE_MATCH`	check	a `match` does not cover all variants and has no `_`
`E_QUESTION_CONTEXT`	check	`?` in a function whose return type is not `Result` / `Option`
`E_ERROR_CONVERT`	check	`?` error type has no `From` conversion to the function's error type
`E_TYPE`	check / run	static type mismatch (check) or dynamic `Any` mismatch (run)
`E_CAP_UNDECLARED`	check	a used effect's capability is not in the header
`E_MAIN_AND_TOPLEVEL`	check	both `fn main` and top-level statements are present
`E_OVERFLOW`	run	checked integer overflow
`W_CAP_UNUSED`	check	a declared capability is never exercised

Implementation note. These are the specified load-time diagnostics. The current interpreter runs the program directly (no separate static-check phase), so several of these surface instead as the runtime errors of Chapter 32 — for example, a non-exhaustive match becomes a runtime "no match arm covered the value", and an undeclared effect that the host gates becomes a runtime `CapabilityDenied`.

32 · Runtime errors

When a script fails at run time, the interpreter produces one of these typed errors. They are values to a Rust host (Chapter 34) and are shown in the tutorial's output panel as `error[Kind]: message`.

RUNTIME ERROR KINDS

KIND	MESSAGE	TYPICAL CAUSE
Undefined	undefined name `n`	using a variable or function that is not in scope
Type	type error: …	an operator or built-in applied to the wrong kind of value
NotCallable	`n` is not callable	calling something that is not a function
Arity	`f` expected N args, got M	wrong argument count
Arithmetic	arithmetic error: …	integer overflow, or division/modulo by zero
IndexOutOfBounds	index out of bounds	indexing a list past its length with `[i]`
NoField	no field `n`	accessing a field a struct does not have
NoMethod	no method `n`	calling an unknown method or stdlib function
NonExhaustiveMatch	no match arm covered the value	a `match` with no matching arm and no `_`
NotBool	… condition is not a bool	a non-boolean `if` / `while` condition or `&&` / `||` operand
LimitExceeded	resource limit exceeded: …	steps, memory, time, or call-depth budget hit (Chapter 27)
CapabilityDenied	capability denied: `cap`	an effect without a granting, in-scope capability

33 · Lexer & parser errors

Before a program runs, two earlier phases can reject it. A **lex error** reports a problem tokenising the source, with a line and column:

LEX ERRORS

KIND	CAUSE
UnexpectedChar	a character that cannot begin any token (for example a bare `#` not starting `#![`)
UnterminatedString	a string literal with no closing quote
UnterminatedBlockComment	a `/*` with no matching `*/`
InvalidEscape	an unrecognised `\`-escape in a string (Chapter 3)
InvalidNumber	a numeric literal that does not fit, or malformed digits
InvalidLabel	a `'` not followed by an identifier (a malformed loop label)

A **parse error** reports a token sequence that does not match the grammar (Chapter 29), again with a line and column and a short message such as "expected `fn`, `struct`, or `const`". A lex error encountered during parsing is reported as a parse error wrapping it.

34 · Host embedding API

ncScript is embedded as a Rust library (the `nexacore-script` crate, `no_std` with no `unsafe`). A host parses, configures, and runs scripts, and supplies the effectful surface. This chapter is for integrators; script authors can skip it.

REFERENCE

34.1 One-shot helpers

For a pure script with no host effects, two free functions suffice. Both return the script's value and its captured `print` lines.

embed-simple.rs RUST

```rust
use nexacore_script::{run, run_with_limits, Limits};

let (value, output) = run(src)?;                  // no limits, no
capabilities

let (value, output) = run_with_limits(src, Limits {
    max_steps: Some(1_000_000),
    max_call_depth: Some(256),
    ..Limits::default()
})?;
```

34.2 The `Interpreter` builder

For capabilities, host effects, or a wall-clock deadline, build an interpreter. The builder methods consume and return `self`.

INTERPRETER API

METHOD	PURPOSE
`Interpreter::new()`	a fresh interpreter (deny-all, unlimited)
`.with_limits(Limits)`	set the resource budgets
`.with_clock(Rc<dyn Clock>)`	attach a monotonic clock (enables the time deadline)
`.with_capabilities(Grants)`	install the granted capability set
`.with_effect_handler(Box<dyn EffectHandler>)`	install the host's effectful surface
`.load(&Program)`	register the parsed program's functions and record its declared capabilities
`.run_main()`	call `main`; returns the value or a runtime error
`.output()`	the captured `print` lines
`.declared_capabilities()`	the capabilities the script's header declared
`.steps()` / `.live_bytes()`	current usage against the budgets

embed-full.rs RUST

```rust
use nexacore_script::{parse, Interpreter, Grants, Capability};

let program = parse(src)?;
let mut interp = Interpreter::new()
    .with_capabilities(
        Grants::none().with(Capability::scoped("fs.read", "/etc/nexacore")))
    .with_effect_handler(Box::new(my_host));
interp.load(&program);
let value = interp.run_main()?;
let lines = interp.output();
```

34.3 Providing effects

A host implements `EffectHandler` with two methods: `required_capability(namespace, function)` declares which capability a given `ns::func` call needs (or `None` if this host does not provide it), and `perform(namespace, function, args)` runs the effect — called only after the capability gate passes. Effect results are returned as a small `HostValue` (unit, bool, int, or string) and interned into the script's value space.

This split is the security boundary: the language guarantees an effect cannot run without a granting, in-scope capability; the host decides what the effect actually does.

35 · ncScript for Rust users

If you know Rust, you already know most of ncScript's surface. This chapter is the diff.

RUST → NCSCRIPT

RUST	NCSCRIPT
`let`, `let mut`, shadowing	same
`fn`, tail expression is the value, `return`	same
`if` / `match` as expressions, guards, or-patterns	same
`Result`, `Option`, `?`	same
`struct`, `enum`, field shorthand	same
`println!("{}", x)`	`print("…" + x.to_string())` — no macros, no `{}` format
the borrow checker, lifetimes, `&` / `&mut`	removed — reference counting + copy-on-write instead
`clone()` everywhere	unnecessary — aggregates share by reference, copy on write
monomorphised generics, `unsafe`, macros	removed (generics are type-erased; no `unsafe`; no macros)
wrapping/overflowing integer ops	integer arithmetic is checked (overflow and divide-by-zero error)
ambient `std::fs` / `std::net`	capability-gated host effects; nothing ambient
panics for errors	typed errors in `Result`; `panic` is reserved for invariants

The single most important difference: ncScript has **no ambient authority**. A function that would, in Rust, simply call `std::fs::read` must in ncScript reach a host effect gated by a declared `fs.read` capability — and a script with no header cannot do I/O at all.

36 · Style guide

Conventions the formatter enforces and the community follows. None changes meaning; all aid readability.

- **Indentation.** Four spaces, never tabs.
- **Naming.** `snake_case` for functions, bindings, and modules; `UpperCamelCase` for types, enums, and variants; `SCREAMING_SNAKE_CASE` for constants.
- **Trailing commas** in multi-line lists, struct literals, match arms, and capability headers — so adding a line is a one-line diff.
- **One item per line**; one statement per line.
- **Errors.** Return `Result` for anything that can fail; reserve `panic` for "impossible" invariant violations. Prefer `?` over nested `match` when only threading failures.
- **Least privilege.** Declare the narrowest capability scope that works; let the `W_CAP_UNUSED` warning prune declarations you do not exercise.
- **Annotations.** Annotate public function signatures and anything non-obvious; leave throwaway locals to inference.
- **Files.** One compilation unit per `.ncs` file, UTF-8, ending with a newline; the capability header, if any, comes first.

37 · Cookbook

Short, complete recipes. Those tagged `PLAYGROUND` run as-is in the tutorial.

37.1 Sum a list

sum.ncs PLAYGROUND

```
fn sum(xs) {
    let mut total = 0;
    for x in xs { total = total + x; }
    total
}
fn main() { sum([3, 9, 15]) }   // => 27
```

37.2 Count elements that satisfy a predicate

count-even.ncs PLAYGROUND

```
fn count_even(xs) {
    let mut n = 0;
    for x in xs { if x % 2 == 0 { n = n + 1; } }
    n
}
fn main() { count_even([1, 2, 3, 4, 6]) }   // => 3
```

37.3 Build a transformed list

doubled.ncs PLAYGROUND

```
fn main() {
    let mut out = [];
    for x in [1, 2, 3] { out.push(x * 2); }
    out.len()   // => 3   (out is [2, 4, 6])
}
```

37.4 FizzBuzz with match guards

fizzbuzz.ncs PLAYGROUND

```
fn word(n) {
    match 0 {
        _ if n % 15 == 0 => "fizzbuzz",
        _ if n % 3  == 0 => "fizz",
        _ if n % 5  == 0 => "buzz",
        _ => n.to_string()
    }
}
fn main() {
    for n in [9, 10, 15] { print(word(n)); }   // fizz, buzz, fizzbuzz
}
```

37.5 Safe division returning Result

safe-div.ncs PLAYGROUND

```
fn safe_div(a, b) {
    if b == 0 { Err("divide by zero") } else { Ok(a / b) }
}
fn main() {
    match safe_div(20, 4) { Ok(q) => q, Err(_) => -1 }   // => 5
}
```

37.6 Parse-or-default

parse-default.ncs PLAYGROUND

```
fn to_int_or(s, fallback) {
    match string::to_int(s) { Some(n) => n, None => fallback }
}
fn main() { to_int_or("not a number", 0) + to_int_or("42", 0) }   // => 42
```

37.7 Clamp a value

clamp.ncs PLAYGROUND

```
fn clamp(x, lo, hi) { math::max(lo, math::min(x, hi)) }
fn main() { clamp(120, 0, 100) + clamp(-5, 0, 100) }   // 100 + 0 => 100
```

37.8 Greatest common divisor (recursive)

gcd.ncs PLAYGROUND

```
fn gcd(a, b) { if b == 0 { a } else { gcd(b, a % b) } }
fn main() { gcd(48, 36) }   // => 12
```

37.9 Running maximum

max-of.ncs PLAYGROUND

```
fn max_of(xs) {
    let mut best = xs[0];
    for x in xs { if x > best { best = x; } }
    best
}
fn main() { max_of([3, 9, 2, 7]) }   // => 9
```

37.10 Find with collections

find.ncs PLAYGROUND

```
fn main() {
    let xs = [5, 6, 7, 8];
    match collections::index_of(xs, 7) {
        Some(i) => i,    // => 2
        None    => -1
    }
}
```

37.11 Join strings

join.ncs PLAYGROUND

```
fn join(parts, sep) {
    let mut out = "";
    let mut first = true;
    for p in parts {
        if first { out = p; first = false; } else { out = out + sep + p; }
    }
    out
}
fn main() { join(["a", "b", "c"], "-") }   // => "a-b-c"
```

37.12 A capability-gated read

read.ncs PLAYGROUND

```
#![capabilities(fs.read("/etc/nexacore"))]

fn main() {
    let notes = fs::read("/etc/nexacore/notes/a.txt");   // in scope ->
allowed (mocked)
    print(notes);
}
```

Change the path to one outside `/etc/nexacore` and the gate returns `CapabilityDenied` before any read — deny-by-default in action (Chapter 25).

38 · Glossary

ARC	automatic reference counting: the memory scheme for aggregates, with deterministic, immediate reclamation (Chapter 15).
ATTENUATION	narrowing a capability when granting or delegating it; a grant can only shrink, never widen, in scope (Chapter 25).
CAPABILITY	a named, optionally scoped right to perform a class of effect (Chapter 25).
CAPABILITY HEADER	the `#![capabilities(...)]` attribute declaring a script's maximal effect set (Chapter 26).
COMPILATION UNIT	one `.ncs` file: header, items, then statements (Chapter 2).
COPY-ON-WRITE	the observable rule that mutating a shared aggregate copies it first, so value semantics hold without a borrow checker (Chapter 15).
DENY-BY-DEFAULT	the rule that ambient authority is empty: no effect is possible unless a capability grants it (Chapter 25).
EFFECT	an observable interaction with the host or outside world — filesystem, network, AI, config, process, time, randomness (Chapter 24).
GRADUAL TYPING	optional types: annotate where it pays, infer or defer elsewhere through the `Any` boundary (Chapter 16).
ITEM	

	a top-level declaration: `fn`, `struct`, `enum`, `const`, `impl`, or `use` (Chapter 2).
NCIP	NexaCore Interpreted Program — the name for the language proper; also the NexaCore Improvement Proposal series that specifies it.
.NCS	NexaCore System Script — the source-file extension.
PLACE	an assignable location: a variable, a field, or an index (Chapter 7).
SCOPE (CONCURRENCY)	a `scope { … }` region that owns its spawned tasks (Chapter 17).
SCOPE (LEXICAL)	the region of a program in which a binding is visible (Chapter 6).
SHADOWING	re-using a name with a new `let`, creating a fresh binding (Chapter 6).
STRUCTURED CONCURRENCY	concurrency whose task lifetimes are tied to lexical scopes; no detached tasks (Chapter 17).
TAIL EXPRESSION	a block's final expression, written without a `;`, which is the block's value (Chapter 9).
TREE-WALKING INTERPRETER	the reference runtime: it evaluates the AST directly, for auditability and a small trust base (Chapter 1).
VALUE SEMANTICS	the property that a value behaves as if independently owned: no surprising aliasing (Chapter 15).
WILDCARD	the _ pattern, which matches anything and binds nothing (Chapter 11).

39 · Status matrix

This manual documents ncScript v1. The language version is independent of the NexaCore OS release; the grammar is the wire-stable definition. Additive changes may ship in minor revisions; any change that rejects a previously-valid v1 program is a new major version. The matrix below records where each area runs today.

FEATURE STATUS

AREA	PLAYGROUND	REFERENCE INTERPRETER
let / mut, const, functions, recursion	yes	yes
if / match (guards, or-patterns), while / for / loop	yes	yes
lists, indexing, `.len` / `.push` / `.to_string`	yes	yes
structs (literals, field access, shorthand)	literals + access	yes
Result / Option, `?`	yes	yes
`string`, `math`, `collections`	yes	yes
`json`, `datetime`	—	yes
capability header + deny-by-default gate + scope matching	yes (mocked effects)	yes
step / call-depth budgets	yes	yes (+ memory, time)
custom enums, `impl` methods, traits	—	declarations + values; methods pending
Float, Char, maps, tuples-as-distinct-type	—	Float yes; tuples-as-lists; maps/Char pending
generics, gradual typing, structured concurrency	—	parsed; semantics landing incrementally

Authoritative sources: the specification NCIP-ncScript-030 (CC0) and the reference implementation in the `nexacore-script` crate.

40 · Implementation notes

The divergences between the v1 specification and the current reference interpreter, gathered in one place. None changes the language you should write toward; each tells you what the runtime does today so you are never surprised.

SPEC VS. CURRENT INTERPRETER

AREA	TODAY	CHAPTER
Integer literals	decimal and `0x` hex only; no `0b` / `0o`	3
Float literals	the `.` must be followed by a digit; `5.` is not a float	3
String escapes	`\u{}` not supported in string literals (JSON parsing accepts `\uXXXX`)	3
Char literals	none; `'…` is a loop label — use a one-character string	3
Keywords `as`, `where`	reserved but unused by the parser	3
`mut` & type annotations	parsed but not enforced at run time	6, 16
Assignment targets	simple variables only; no field/index assignment	7
Unary integer negation	wraps (binary subtraction is checked)	8
Struct equality	two structs never compare equal under `==`	4
Maps	map literals deferred; no distinct map value yet	5
Tuples	represented as lists; tuple patterns match lists	5
`impl` methods, generics	parsed; methods not yet callable, generic params ignored	9, 12
Match exhaustiveness	checked at run time (no static check), as `NonExhaustiveMatch`	11
Loop labels	accepted on `loop` only	10
`scope` / `spawn` / `await`	run inline and synchronously	17
Cycle collector	reference counting today; collector is a follow-up	15
Static type checker	not yet; all checks are dynamic (runtime errors)	16
Capability header	recorded but not auto-granted; the host installs grants	25

The in-browser tutorial runs an independent JavaScript re-implementation of the documented subset, for demonstration; the canonical interpreter is the Rust `nexacore-script` crate, and the normative definition is NCIP-ncScript-030.