RCBScript — Grammar, Syntactic Structure

Version: 1.0

Status: Draft for review

Language Family: Object-Oriented (OO), with procedural elements and lightweight functional support

Design Goals: Clarity • Safety • Familiarity for C++ users • Low ceremony • Readable tests

---

1) Language Summary (from the baseline spec)

- **OO-first** with `class`, `public`/`private`, and `func` methods.
- **Indentation-based blocks** (offside rule). No braces and no semicolons.
- **Memory model:** explicit `make` and `discard`, no raw `*`/`&`. `handle<T>` is the managed reference type.
- **Control flow:** `if`/`else`, `loop from` / `loop while` / `loop until`, `match` for pattern matching.
- **Data types:** `number`, `text`, `bool` (`yes`/`no`), `char`, `handle<T>`; `record` for immutable product types.
- **No arrays yet** (lists planned). No `switch`/`break`/`continue`.
- **Files:** single-file modules, extension `.rcb`. Executes on RCB VM. Built-in test blocks.

---

2) Lexical Structure

2.1 Character Set

- Unicode source files in UTF-8.

2.2 Line Terminators

- `\n` is the canonical line terminator. Windows `\r\n` allowed; normalized by the lexer.

2.3 Whitespace & Indentation

- Spaces and tabs allowed; **tabs are expanded to 4 spaces** for indentation measurement.
- The **offside rule** produces virtual tokens: `INDENT` and `DEDENT`.
- **Rule:** After any line ending that is syntactically inside a block-introducing construct (e.g., `class`, `func`, `if`, `else`, `loop`, `match`, `record` body when multiline), the next line's leading indentation establishes block depth. Increased indentation emits one `INDENT`; decreased indentation emits one or more `DEDENT`s to match the nearest prior depth.
- **Statement termination:** A physical newline **ends a statement**, unless the lexer is in a **line-continuation mode**, entered when the line ends after any of: binary operator, comma `,`, opening delimiter `( [ <`, or an explicit `\\` line continuation token.

2.4 Comments

- Line: `# ...` to end of line.
- Block: `### ... ###` (non-nesting; may span lines).

2.5 Identifiers

```
Identifier := Letter ( Letter | Digit | '_' )*
Letter := UnicodeLetter
Digit := '0'..'9'
```

- Identifiers are case-sensitive. Keywords are reserved.

2.6 Literals

- **Number**: decimal integers or floats (`123`, `3.14`, `1_000`, `2.5e-3`).
- **Text**: double-quoted Unicode strings with escapes `\n`, `\t`, `\\`, `\"`.
- **Char**: single quotes with one Unicode scalar or escape (e.g., `'a'`, `'\n'`).
- **Bool**: `yes` | `no`.

2.7 Operators & Delimiters

- Arithmetic: `+ - * / %`
- Comparison: `== != < <= > >=`
- Logical: `and or not`
- Assignment: `=` (simple). Compound assignment is **not** in v1.0.
- Member access: `.`
- Grouping / calls / generics / tuples: `( ) < > ,`
- Pattern arrows: `=>`

---

3) Tokens & Keywords

Keywords (reserved):

```
class public private func return make discard loop from while until if else
match handle yes no record test expect is
```

Soft keywords (contextual; not reserved):

```
where as
```

---

4) Grammar (EBNF)

Notes:

- Indentation is expressed through implicit `INDENT`/`DEDENT` tokens.
- Newline is `NL`.
- Optional items use `[ ... ]`, repetition uses `{ ... }`

4.1 Module

```
Module        := { TopLevelDecl NL* }
TopLevelDecl  := ClassDecl | FuncDecl | RecordDecl | TestDecl
```

4.2 Declarations

```
ClassDecl     := 'class' Identifier NL INDENT { ClassMember } DEDENT
ClassMember   := Visibility? ( FieldDecl | MethodDecl ) NL*
Visibility    := 'public' | 'private'

FieldDecl     := Identifier ':' Type
```

```
MethodDecl    := 'func' Identifier '(' ParamList? ')' ReturnAnn? NL
                 INDENT Block DEDENT

FuncDecl      := 'func' Identifier '(' ParamList? ')' ReturnAnn? NL
                 INDENT Block DEDENT

ParamList     := Param { ',' Param }
Param         := Identifier ':' Type [ '=' Expr ]

ReturnAnn     := '->' Type

RecordDecl    := 'record' Identifier '(' RecordFieldList? ')'
RecordFieldList := RecordField { ',' RecordField }
RecordField   := Identifier ':' Type

TestDecl      := 'test' TextLiteral NL INDENT { TestStmt NL } DEDENT
TestStmt      := 'expect' Expr 'is' Expr | Stmt
```

4.3 Statements

```
Block         := { Stmt NL }

Stmt          := SimpleStmt | IfStmt | LoopFrom | LoopWhile | LoopUntil
               | MatchStmt | DiscardStmt | ReturnStmt
```

```
SimpleStmt    := Assign | ExprStmt

Assign        := LValue '=' Expr
LValue        := Identifier | Primary '.' Identifier

ExprStmt      := Expr

ReturnStmt    := 'return' Expr?
DiscardStmt   := 'discard' Expr

IfStmt        := 'if' Expr NL INDENT Block DEDENT
                 { 'else' NL INDENT Block DEDENT }?

LoopFrom      := 'loop' 'from' Identifier '=' Expr 'to' Expr [ 'step' Expr
] NL
                  INDENT Block DEDENT

LoopWhile     := 'loop' 'while' Expr NL INDENT Block DEDENT
LoopUntil     := 'loop' 'until' Expr NL INDENT Block DEDENT

MatchStmt     := 'match' Expr NL INDENT { PatternArm } DEDENT
PatternArm    := Pattern '=>' NL INDENT Block DEDENT
```

4.4 Expressions

```
Expr          := OrExpr
OrExpr        := AndExpr { 'or' AndExpr }
```

```
AndExpr       := NotExpr { 'and' NotExpr }
NotExpr       := [ 'not' ] CmpExpr

CmpExpr       := AddExpr [ ( '==' | '!=' | '<' | '<=' | '>' | '>=' )
AddExpr ]
AddExpr       := MulExpr { ( '+' | '-' ) MulExpr }
MulExpr       := UnaryExpr { ( '*' | '/' | '%' ) UnaryExpr }

UnaryExpr     := Primary | ( '+' | '-' ) UnaryExpr

Primary       := Literal
               | Identifier
               | Primary '.' Identifier
               | Call
               | Group
               | MakeExpr

Call          := Identifier '(' ArgList? ')'    # calls require parentheses
when args exist
ArgList       := Expr { ',' Expr }

Group         := '(' Expr ')'

MakeExpr      := 'make' TypeOrCtor
TypeOrCtor    := Type | Identifier             # `make Person` or `make
handle<number>`

Literal       := NumberLiteral | TextLiteral | CharLiteral | BoolLiteral
BoolLiteral   := 'yes' | 'no'
```

4.5 Types

```
Type          := SimpleType | GenericType
SimpleType    := 'number' | 'text' | 'bool' | 'char' | Identifier
GenericType   := 'handle' '<' Type '>'
```

4.6 Pattern Grammar (for `match`)

```
Pattern       := LiteralPattern | TypePattern | IdPattern | Wildcard
LiteralPattern:= NumberLiteral | TextLiteral | CharLiteral | BoolLiteral
TypePattern   := Identifier [ '(' PatternArgList? ')' ]     # future
extensibility
PatternArgList:= Pattern { ',' Pattern }
IdPattern     := Identifier                                 # binds
variable
Wildcard      := '_'
```

---

5) Syntactic Structure & Examples

5.1 Classes & Methods

```
class Person
  public name:text
  public age:number

  func greet()
    print("Hello, " + name)

func main() -> number
  p = make Person
  p.name = "John"
  p.age = 42
  p.greet()
  return 0
```

5.2 Functions and Returns

```
func sum(a:number, b:number) -> number
  return a + b

func greet()
  print("Hi")
```

5.3 Loops

```
loop from i = 1 to 10
  print(i)

loop while i < 10
  i = i + 1

loop until done
  done = check()
```

5.4 Match

```
match token
  '+'/char =>
    handle_plus()
  '0'/char =>
    emit_number(0)
  _ =>
    error("unexpected")
```

5.5 Memory & Handles

```
```

```
p = make handle<number>
p = 5
print(p)      # implicit deref by value semantics (implementation defined
in VM)

discard p     # manual release
```

5.6 Records (immutable)

```
record Point(x:number, y:number)

func length2(pt:Point) -> number
  return pt.x * pt.x + pt.y * pt.y
```

5.7 Tests

```

test "addition"
  expect sum(2, 3) is 5
```

---

6) Static Semantics (selected)

- **Visibility:** Unqualified member access within the defining class has full access; outside respects `public`/`private`.
- **Type inference:** Not in v1.0; all parameters and fields require type annotations. Locals can omit type if assigned from a literal or `make` — inferred by the compiler.
- **Bool values:** Only `yes`/`no` are truthy; no implicit numeric → bool conversions.
- **Handles:** `handle<T>` behaves as a reference with copyable handle semantics; `discard` decreases the reference's ownership count, releasing when it reaches zero. Dangling-handle use is a runtime error in v1.0.
- **Overloading:** Not in v1.0. Function names must be unique within a scope.

---

7) Error Handling Strategy

- **Lex/parse errors:** precise span with caret diagnostics, expected token sets, and indentation mismatch hints ("possible mixed tabs/spaces").
- **Type errors:** show actual vs expected types with suggested fixes.
- **Runtime errors:** null/dangling handle deref, out-of-bounds (when lists arrive), division by zero, pattern non-exhaustiveness warnings (error if `match` proven non-exhaustive in v1.1+).