The CPY Programming Language

Version 1.0 — Language Reference & Technical Documentation

A C-structured, Python-simple programming language

Contents

1 Introduction

CPY (pronounced *"see-pie"*) is a statically-checked, dynamically-typed programming language designed for clarity and learning. It blends the structural rigour of C—braces, semicolons, explicit declarations—with the readability of Python.

1.1 Design Goals

Goal	How CPY achieves it
Readable syntax	Python-inspired keywords (`let`, `print`), minimal boilerplate
Explicit structure	C-style braces `{ }`, semicolons `;`, parenthesised conditions
Static safety	Semantic analysis catches undeclared and duplicate variables *before* execution
Simple mental model	No classes, no modules, no implicit coercions—just values, expressions, and stat

1.2 What CPY Is Not

CPY is an interpreted educational language. It is not intended for production systems, concurrent programming, or large-scale applications.

2 Getting Started

2.1 Prerequisites

- **Java 8** or later (JDK)

2.2 Project Structure

```
Compiler/├──
 src/│
    ├── Main.java│
    ├── lexer/       (TokenType, Token, Lexer)│
    ├── parser/      (Parser)│
    ├── ast/         (AST node classes)│
    ├── semantic/    (SemanticAnalyzer)│
    └── interpreter/ (Interpreter)└──
 test.cpy         (sample program)
```

2.3 Building

```
javac -d out src/Main.java src/lexer/*.java src/parser/*.java \
      src/ast/*.java src/semantic/*.java src/interpreter/*.java
```

2.4 Running a Program

```
java -cp out Main <filename>.cpy
```

If no filename is provided, the compiler defaults to `test.cpy` in the current directory.

2.5 Hello, World!

```
print("Hello, World!");
```

Output:

```
Hello, World!
```

3 Language Fundamentals

3.1 Source Files

CPY source files use the **.cpy** extension. Files are encoded in UTF-8 and processed sequentially, top to bottom.

3.2 Statements

Every statement in CPY is terminated by a **semicolon** `;`. Compound statements (blocks) are wrapped in **braces** `{ }`.

```
let x = 10;          // simple statement
if (x > 5) {         // compound statement
    print(x);
}
```

3.3 Identifiers

Identifiers begin with a letter or underscore and may contain letters, digits, and underscores.

```
Valid:    myVar   _count   player1   MAX_SIZE
Invalid:  2fast   my-var   $amount
```

3.4 Case Sensitivity

CPY is **case-sensitive**. `myVar`, `MyVar`, and `MYVAR` are three distinct identifiers.

4 Data Types

CPY has four primitive data types. All values are dynamically typed at runtime.

Type	**Literal Syntax**	**Java Representation**	**Example**
Number	Integer or decimal digits	`Double`	42, `3.14`
String	Double-quoted text	`String`	`"hello"`
Char	Single-quoted character	`Character`	`'A'`, `'z'`
Boolean	*(implicit — see §6.3)*	`Boolean`	result of ==, <, etc.
Array	Bracket-enclosed list	`List<Object>`	`[1, 2, 3]`

4.1 Numbers

All numeric values are 64-bit floating-point internally. Integer literals are stored as doubles with no fractional part.

```
let a = 42;       // stored as 42.0 internally, displayed as "42"
let b = 3.14;     // stored as 3.14
let c = 0.5;      // leading zero required
```

> **Display rule:** Numbers that end in `.0` are printed without the decimal part (e.g., `10.0` $\rightarrow$ `10`).

4.2 Strings

Strings are enclosed in **double quotes** `"`. They may span only a single source line (multi-line strings are not supported). Strings support concatenation with the + operator.

```
let name = "CPY";
let greeting = "Hello, " + name + "!";  // "Hello, CPY!"
```

4.3 Chars

Character literals are enclosed in **single quotes** `'` and must contain **exactly one character**.

```
let letter = 'A';
let digit  = '7';
```

> **Note:** A char and a single-character string are distinct types. `'A'` is a `Character`; `"A"` is a `String`.

4.4 Booleans

There is no boolean literal keyword. Booleans are produced by comparison and logical operators. The following truthiness rules apply:

Value	**Truthy?**
`0` (number)	□ Falsy
Non-zero numbers	□ Truthy
Non-empty strings	□ Truthy
`null`	□ Falsy
`true` (boolean)	□ Truthy
`false` (boolean)	□ Falsy

5 Variables & Assignment

5.1 Declaration

Variables are declared with the `let` keyword. Every variable **must** be initialized at the point of declaration.

```
Syntax: let <identifier> = <expression> ;
```

```
let x = 10;
let name = "Alice";
let scores = [90, 85, 78];
```

Semantic rule: Re-declaring a variable that already exists in the current scope is a compile-time error.

5.2 Assignment

After declaration, a variable's value can be changed with the assignment statement.
`Syntax: <identifier> = <expression> ;`

```
x = x + 1;
name = "Bob";
```

Semantic rule: Assigning to a variable that has never been declared is a compile-time error.

6 Operators

6.1 Arithmetic Operators

Operator	Description	Example	Result
`+`	Addition / String concatenation	`3 + 4`	7
`-`	Subtraction	`10 - 3`	7
`*`	Multiplication	`6 * 7`	42
`/`	Division	`20 / 4`	5
`-` (unary)	Negation	`-x`	negated value

String +: If *either* operand is a string, the other operand is converted to its string representation and the two are concatenated.

Division by zero throws a runtime error.

6.2 Comparison Operators

Operator	Description
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

All comparison operators return a **boolean** value. Numeric comparisons require both operands to be numbers.

6.3 Logical Operators

Operator	Description	Example
and	Logical AND	`x > 0 and x < 10`
or	Logical OR	`a == 1 or b == 1`
not	Logical NOT (unary)	`not flag`

Logical operators use **truthiness** (see §4.4) to coerce operands. Both and and `or` evaluate both sides (no short-circuit evaluation).

6.4 Operator Precedence

From **highest** (binds tightest) to **lowest**:

Precedence	Operators	Associativity
1 (highest)	- (unary), not	Right
2	*, /	Left
3	+, -	Left
4	>, >=, <, <=	Left
5	==, !=	Left
6	and	Left
7 (lowest)	`or`	Left

Parentheses ( ) can be used to override precedence:

```
let result = (2 + 3) * 4;   // 20, not 14
```

7 Control Flow

7.1 If / Else

```
Syntax:   if ( <condition> ) { <body> }
          if ( <condition> ) { <body> } else { <body> }
```

```
if (score >= 90) {
    print("Grade: A");
} else {
    print("Grade: B or below");
}
```

For multi-branch logic, nest `if` inside `else`:

```
if (score >= 90) {
    print("A");
} else {
    if (score >= 80) {
        print("B");
    } else {
        print("C");
    }
}
```

7.2 While Loop

`Syntax: while ( <condition> ) { <body> }`

```
let i = 0;
while (i < 5) {
    print(i);
    i = i + 1;
}
```

Tip: Be careful to update the loop variable inside the body to avoid infinite loops.

7.3 For Loop

`Syntax: for ( <init> ; <condition> ; <increment> ) { <body> }`
Each clause is optional:

- **init** — a `let` declaration or assignment (with its own `;`)
- **condition** — any expression (if omitted, defaults to `true` → infinite loop)
- **increment** — an assignment (no trailing `;`)

```
for (let i = 0; i < 10; i = i + 1) {
    print(i);
}
```

▯ CPY does not have ++ or += operators. Use `i = i + 1` for incrementing.

8 Arrays

Arrays are ordered, zero-indexed, heterogeneous collections. They can hold any mix of numbers, strings, chars, booleans, and even other arrays.

8.1 Array Literals

`Syntax: [ <expr> , <expr> , ... ]`

```
let nums   = [10, 20, 30];
let mixed  = [1, "hello", 'X'];
let empty  = [];
let nested = [[1, 2], [3, 4]];
```

8.2 Array Access

`Syntax: <identifier> [ <index> ]`
Indices are **zero-based**. The index expression must evaluate to a number (truncated to an integer).

```
let first = nums[0];    // 10
let last  = nums[2];    // 30
```

▯ **Bounds checking:** Accessing an index outside `[0, length-1]` throws a runtime error.

8.3 Array Mutation

Syntax: `<identifier> [ <index> ] = <expression> ;`

```
nums[1] = 999;
print(nums);    // [10, 999, 30]
```

8.4 Iterating Over an Array

Use a `for` loop with an index counter:

```
let colors = ["red", "green", "blue"];
for (let i = 0; i < 3; i = i + 1) {
    print(colors[i]);
}
```

8.5 Arrays with Expressions

Array elements can be arbitrary expressions:

```
let x = 10;
let y = 20;
let computed = [x + y, x * 2, y - 5];   // [30, 20, 15]
```

8.6 Printing Arrays

When printed, arrays display in bracket notation:

```
print([1, 2, 3]);          // [1, 2, 3]
print(["a", 'b', 3]);      // [a, b, 3]
print([]);                  // []
```

9 Input / Output

9.1 Print Statement

Syntax: `print( <expression> ) ;`

`print` evaluates the expression and writes its string representation to **standard output**, followed by a newline.

```
print(42);                 // 42
print("hello");            // hello
print('A');                // A
print(3 + 4);              // 7
print([1, 2, 3]);          // [1, 2, 3]
print(10 == 10);           // true
```

There is currently no `input` statement. All data must be embedded in the source code.

10 Comments

CPY supports **single-line comments** beginning with `//`. Everything from `//` to the end of the line is ignored.

```
// This is a comment
let x = 10;  // inline comment
```

Block comments (`/* */`) are not supported.

11 Grammar Specification

The complete formal grammar in **EBNF** notation:

```
program        = { statement } EOF ;

statement      = varDecl
               | assignment
               | arrayAssignment
               | ifStmt
               | whileStmt
               | forStmt
               | printStmt
               | block ;

varDecl        = "let" IDENTIFIER "=" expression ";" ;
assignment     = IDENTIFIER "=" expression ";" ;
arrayAssignment= IDENTIFIER "[" expression "]" "=" expression ";" ;

ifStmt         = "if" "(" expression ")" block [ "else" block ] ;
whileStmt      = "while" "(" expression ")" block ;
forStmt        = "for" "(" ( varDecl | assignment | ";" )
                          [ expression ] ";"
                          [ IDENTIFIER "=" expression ]
                       ")" block ;
printStmt      = "print" "(" expression ")" ";" ;
block          = "{" { statement } "}" ;

expression     = or ;
or             = and { "or" and } ;
and            = equality { "and" equality } ;
equality       = comparison { ( "==" | "!=" ) comparison } ;
comparison     = term { ( ">" | ">=" | "<" | "<=" ) term } ;
term           = factor { ( "+" | "-" ) factor } ;
factor         = unary { ( "*" | "/" ) unary } ;
unary          = ( "-" | "not" ) unary | primary ;
primary        = NUMBER
               | STRING
               | CHAR
               | IDENTIFIER [ "[" expression "]" ]
               | "(" expression ")"
               | "[" [ expression { "," expression } ] "]" ;
```

12 How It Works Under the Hood

The CPY compiler processes source code through a **four-stage pipeline**. Each stage transforms the program into a progressively higher-level representation before finally executing it.

```
┌──────────────┐      ┌───────────┐      ┌────────────┐      ┌─────────────┐
│ Source Code  │─────▶│   Lexer   │─────▶│   Parser   │─────▶│  Semantic   │
│  (.cpy file) │      │           │      │            │      │  Analyzer   │
└──────────────┘      └───────────┘      └────────────┘      └─────────────┘
                         Tokens            AST Tree         Validated AST
                                                                 │
                                                                 ▼
                                                         ┌───────────────┐
                                                         │  Interpreter  │
                                                         │   (output)    │
                                                         └───────────────┘
```

12.1 Stage 1 — Lexical Analysis (Lexer)

File: `lexer/Lexer.java`

The lexer (also called *scanner* or *tokenizer*) reads the raw source code character by character and groups them into **tokens**—the smallest meaningful units of the language.

What the Lexer does:

1. **Skips whitespace** — spaces, tabs, carriage returns are ignored (newlines increment the line counter).
2. **Recognizes single-character tokens** — `(`, `)`, `{`, `}`, `[`, `]`, `+`, `-`, `*`, `;`, `,`.
3. **Handles one-or-two-character tokens** — `=` vs `==`, `!` vs `!=`, `>` vs `>=`, `<` vs `<=`.
4. **Scans number literals** — reads digits, optionally followed by `.` and more digits.
5. **Scans string literals** — reads characters between `"` delimiters.
6. **Scans char literals** — reads a single character between `'` delimiters.
7. **Scans identifiers and keywords** — reads alphanumeric sequences, then checks against the keyword table to distinguish `let`, `if`, `while`, etc. from user-defined names.
8. **Strips comments** — `//` causes the rest of the line to be discarded.

Example transformation:

```
Source:   let x = 10 + 3;

Tokens:   [LET] [IDENTIFIER:x] [EQUAL] [NUMBER:10] [PLUS] [NUMBER:3] [SEMICOLON] [
```

Each Token object carries three fields:

Field	Purpose
`type`	The TokenType enum value (e.g., LET, NUMBER, PLUS)
`lexeme`	The exact source text that was matched (e.g., `"10"`, `"x"`)
`line`	The line number in the source file (used for error reporting)

12.2 Stage 2 — Parsing (Parser)

File: `parser/Parser.java`

The parser takes the flat list of tokens and builds an **Abstract Syntax Tree (AST)** — a tree data structure that captures the hierarchical, nested structure of the program.

Algorithm: CPY uses a **recursive-descent parser**, which means each grammar rule is implemented as a Java method that calls other methods for sub-rules.

How precedence works:

The expression parser is organized as a chain of methods, each handling one level of precedence. Lower-precedence operators are parsed at the top of the chain (parsed first, meaning they bind *loosely*):

```
expression()   calls -> or()
    or()       calls -> and()
    and()      calls -> equality()
    equality() calls -> comparison()
    comparison() calls -> term()
    term()     calls -> factor()
    factor()   calls -> unary()
    unary()    calls -> primary()
```

This ensures that `2 + 3 * 4` is parsed as `2 + (3 * 4)`, not `(2 + 3) * 4`.

Example AST:

For the statement `let x = 2 + 3 * 4;`, the parser produces:

```
VarDecl
├── name: "x"
└── initializer: BinaryExpr
    ├── left: Literal(2)
    ├── operator: PLUS
    └── right: BinaryExpr
        ├── left: Literal(3)
        ├── operator: STAR
        └── right: Literal(4)
```

AST Node Hierarchy:

```
Expr (abstract)
├── Literal         – numbers, strings, chars
├── Variable        – identifier references
├── BinaryExpr      – left op right
├── UnaryExpr       – op operand
├── ArrayExpr       – [elem, elem, ...]
└── ArrayAccess     – arr[index]

Stmt (abstract)
├── VarDecl         – let name = expr;
├── Assignment      – name = expr;
├── ArrayAssignment– name[idx] = expr;
├── PrintStmt       – print(expr);
├── IfStmt          – if/else
├── WhileStmt       – while loop
├── ForStmt         – for loop
└── Block           – { stmts }
```

12.3 Stage 3 — Semantic Analysis

File: `semantic/SemanticAnalyzer.java`

Before execution, the semantic analyzer walks the entire AST to catch errors that are syntactically valid but semantically wrong. This is a **static analysis** pass — it runs without executing the program.

Checks performed:

Check	Example Error
Undeclared variable	Using x without a prior `let x = ...;`
Duplicate declaration	Writing `let x = 1;` twice in the same scope
Undeclared array access	Using `arr[0]` when `arr` was never declared

How it works:

The analyzer maintains a **symbol table** — a `HashMap<String, String>` that maps variable names to their type metadata. As it walks the AST:

- On `VarDecl`: checks if the name already exists (duplicate), then adds it.
- On `Assignment` / `ArrayAssignment`: checks if the name exists (undeclared).
- On `Variable` / `ArrayAccess`: checks if the name exists (undeclared).
- Recursively walks sub-expressions and nested blocks.

If any errors are found, they are **collected** and reported together as a batch, rather than stopping at the first error:

```
Semantic errors:
  • Variable 'x' used before declaration (line 3)
  • Variable 'y' already declared (line 7)
```

12.4 Stage 4 — Interpretation

File: `interpreter/Interpreter.java`

The interpreter is a **tree-walking evaluator**. It traverses the validated AST and executes each node directly — there is no compilation to bytecode or machine code.

Runtime Environment:

The interpreter maintains an **environment** — a `HashMap<String, Object>` that maps variable names to their current runtime values.

Execution model:

Node Type	Interpreter Action
`VarDecl`	Evaluate initializer, store name $\rightarrow$ `value` in environment
`Assignment`	Evaluate expression, update name $\rightarrow$ `value` in environment
`ArrayAssignment`	Look up array, evaluate index, bounds-check, set element
`PrintStmt`	Evaluate expression, convert to string, write to stdout
`IfStmt`	Evaluate condition; if truthy, execute `thenBranch`; else execute `elseBranch`
`WhileStmt`	Loop: evaluate condition, if truthy execute body, repeat
`ForStmt`	Execute init; loop: evaluate condition, execute body, execute increment

`Block`	Execute each statement sequentially
`Literal`	Return the stored value directly
`Variable`	Look up the name in the environment
`BinaryExpr`	Evaluate both sides, apply operator, return result
`UnaryExpr`	Evaluate operand, apply operator (negate or logical NOT)
`ArrayExpr`	Evaluate each element, collect into a `List<Object>`
`ArrayAccess`	Look up array, evaluate index, bounds-check, return element

Type coercion rules:

- **+ with strings:** if either operand is a `String`, the other is converted via `stringify()` and the result is string concatenation.
- **Arithmetic (`-`, `*`, `/`):** both operands must be numbers; otherwise, a runtime error is thrown.
- **Comparisons (>, <, etc.):** both operands must be numbers.
- **Equality (==, `!=`):** any types can be compared; different types are never equal.
- **Array indices:** must be numbers; the fractional part is truncated (e.g., `2.7` $\rightarrow$ index 2).

13 Error Handling

CPY reports errors at three stages with clear, human-readable messages:

13.1 Lexer Errors (Character Level)

Error	Message
Unknown character	`Unexpected character '€' at line 5`
Unterminated string	`Unterminated string at line 12`
Empty char literal	`Empty char literal at line 8`
Multi-char literal	`Unterminated or multi-character char literal at line 8`

13.2 Parser Errors (Structure Level)

Error	Example Message
Missing semicolon	`Expected ';' after variable declaration (got 'let' at line 4)`
Missing parenthesis	`Expected ')' after if condition (got '{' at line 6)`
Missing brace	`Expected '{' (got 'print' at line 8)`
Unexpected token	`Unexpected token: }`

13.3 Semantic Errors (Logic Level)

Error	Example Message
Undeclared variable	`Variable 'x' used before declaration (line 3)`
Duplicate variable	`Variable 'x' already declared (line 7)`

13.4 Runtime Errors (Execution Level)

Error	Example Message
Division by zero	`Runtime error at line 5: Division by zero`
Type mismatch	`Runtime error at line 8: Operands must be numbers`
Undefined variable	`Runtime error at line 3: Undefined variable 'x'`
Index out of bounds	`Runtime error at line 12: Array index 5 out of bounds (size 3)`
Not an array	`Runtime error at line 10: 'x' is not an array`

14 Complete Example Program

```
// ============================================
//  CPY Demo — Showcasing all language features
// ============================================

// --- Variables & Arithmetic ---
let x = 10;
let y = 20;
let sum = x + y;
print(sum);                    // 30

// --- Strings ---
let name = "CPY";
print("Hello, " + name + "!");  // Hello, CPY!

// --- Chars ---
let grade = 'A';
print(grade);                   // A

// --- Comparisons & If/Else ---
if (x < y) {
    print("x is smaller");
} else {
    print("y is smaller");
}

// --- While Loop ---
let count = 0;
while (count < 3) {
    print(count);
    count = count + 1;
}
// Output: 0  1  2

// --- For Loop ---
for (let i = 0; i < 3; i = i + 1) {
    print(i * 10);
}
// Output: 0  10  20

// --- Arrays ---
let fruits = ["apple", "banana", "cherry"];
print(fruits);                  // [apple, banana, cherry]
print(fruits[1]);               // banana
```

```
fruits[1] = "blueberry";
print(fruits);                // [apple, blueberry, cherry]

// --- Mixed Array ---
let data = [42, "hello", 'Z', [1, 2]];
print(data);                  // [42, hello, Z, [1, 2]]

// --- Logical Operators ---
let a = 5;
let b = 10;
if (a > 0 and b > 0) {
    print("both positive");
}

if (a > 100 or b > 5) {
    print("at least one condition met");
}

print(not (a == b));          // true
```

15 Appendix A — Token Types

Category	Tokens
Keywords	LET, IF, ELSE, WHILE, FOR, PRINT
Identifiers & Literals	IDENTIFIER, NUMBER, STRING, CHAR
Arithmetic	PLUS (+), MINUS (-), STAR (*), SLASH (/)
Assignment	EQUAL (=)
Comparison	EQUAL_EQUAL (==), BANG_EQUAL (!=), GREATER (>), GREATER_EQUAL (>=), LESS (<), LESS_EQUAL (<=)
Logical	AND, OR, NOT
Grouping	LPAREN ((), RPAREN ()) , LBRACE ({), RBRACE (}), LBRACKET ([), RBRACK
Delimiters	SEMICOLON (;), COMMA (,)
Special	EOF

16 Appendix B — Reserved Keywords

The following words cannot be used as variable names:

Keyword	Purpose
let	Variable declaration
if	Conditional branch
else	Alternative branch
while	While loop
for	For loop
print	Output statement
and	Logical AND operator
or	Logical OR operator
not	Logical NOT operator

CPY Language Documentation v1.0 — February 2026
Built with □ as an educational compiler project