LambdaScript

Language Reference Manual

Version 1.0.0

Designed for High-Performance Algorithmic Processing

Author:
CodeStitcher

February 28, 2026

Contents

Chapter 1

Introduction

1.1 What is LambdaScript?

LambdaScript is a statically-typed, imperative programming language designed to bridge the gap between the ease of scripting and the rigor of systems programming. Unlike dynamically typed languages like JavaScript or Python, LambdaScript enforces type safety at compile-time and run-time, ensuring robust application development.

It features a C-family syntax that is immediately familiar to developers, but introduces modern concepts such as stream-based output, structural data modeling, and functional array manipulation.

1.2 Design Philosophy

The core design pillars of LambdaScript are:

- **Explicit Typing:** Variables must have defined types to prevent implicit coercion errors.
- **Streamlined I/O:** Input and Output operations are handled via distinct operators (`«`) and keywords to separate logic from I/O.
- **First-Class Functions:** Functions are treated as values, enabling higher-order logic such as mapping.
- **Structure-Oriented:** Data is modeled using `structs`, promoting clean data architecture without the overhead of complex class hierarchies.

Getting Started

LambdaScript source files typically use the `.ls` extension. The interpreter executes code sequentially from the top of the file, although function definitions are hoisted.

Chapter 2

Lexical Structure & Types

2.1 Comments

LambdaScript supports standard commenting styles. Comments are ignored by the parser and are used solely for documentation.

```
// This is a single-line comment.

/*
   This is a multi-line comment block.
   It allows for detailed explanations across
   multiple lines of text.
*/
```

Listing 2.1: Comment Syntax

2.2 Primitive Data Types

LambdaScript is a strongly typed language. Every variable must be declared with one of the following primitive types.

Type Keyword	Description	Examples
`int`	Signed 32-bit integers	`42`, `-100`, `0`
`float`	Double-precision floating point	`3.14`, `-0.001`
`string`	Sequence of UTF-8 characters	`"Hello World"`
`boolean`	Logical truth values	`true`, `false`

Table 2.1: Primitive Types in LambdaScript

2.3 Variables

Variables are declared using the syntax: `<type> <identifier> = <value>;`.

```
// Integer declaration
int userCount = 150;

// Float declaration
float pi = 3.14159;

```

```
// String declaration
string greeting = "Welcome to LambdaScript";

// Boolean declaration
boolean isOnline = true;
```

Listing 2.2: Variable Declaration

Type Safety

You cannot assign a value of one type to a variable of another type without explicit conversion. For example, assigning `3.5` to an `int` variable will result in a generic runtime error.

Chapter 3

Input and Output

3.1 The Print Stream

LambdaScript utilizes a C++ style stream operator (≪) for output. This allows for chaining multiple values—strings, numbers, or booleans—into a single output line.

```
int score = 95;
string player = "Alex";

// Basic string output
print ≪ "Game Over";

// Chaining variables and literals
print ≪ "Player: " ≪ player ≪ " | Score: " ≪ score;
```

Listing 3.1: Using Print

3.2 String Interpolation

For easier formatting, variables can be embedded directly into strings using curly braces `{ }`.

```
int age = 25;
string name = "John";

// The variable values replace the {placeholders}
print ≪ "User {name} is {age} years old.";
```

Listing 3.2: String Interpolation

3.3 Input

Input is handled via the `input` keyword, which pauses execution and waits for the user to type a line of text. It returns a `string`.

```
print ≪ "Please enter your name:";
string name = input();

print ≪ "Hello, " ≪ name;
```

Listing 3.3: Taking Input

Alternatively, you can provide a prompt string directly:

```
1 string color = input("What is your favorite color? ");
```

Chapter 4

Control Flow

4.1 Conditional Statements

Conditional logic is handled via `if`, `else if`, and `else` blocks. The condition must evaluate to a `boolean` type.

```
int battery = 15;

if (battery > 50) {
    print ≪ "Battery is healthy.";
} else {
    if (battery > 10) {
        print ≪ "Low battery warning.";
    } else {
        print ≪ "Critical battery level!";
    }
}
```

Listing 4.1: If-Else Structure

4.2 Loops

LambdaScript supports both `while` and `for` loops for iteration.

4.2.1 While Loop

Executes a block of code as long as the condition remains true.

```
int counter = 5;

while (counter > 0) {
    print ≪ "Countdown: " ≪ counter;
    counter = counter - 1;
}
print ≪ "Liftoff!";
```

Listing 4.2: While Loop

4.2.2 For Loop

The standard C-style loop for iteration with an initializer, condition, and incrementor.

```
// Print even numbers from 0 to 8
for (int i = 0; i < 10; i = i + 2) {
    print ≪ "Number: " ≪ i;
}
```

Listing 4.3: For Loop

4.2.3 Flow Control Keywords

- `break`: Immediately terminates the loop.
- `continue`: Skips the rest of the current iteration and proceeds to the next.

Chapter 5

Arrays (Lists)

Arrays in LambdaScript are dynamic lists that hold elements of a single type. They are denoted by brackets `[]`.

5.1 Declaration

```
// An array of integers
int[] fib = [1, 1, 2, 3, 5, 8];

// An array of strings
string[] fruits = ["Apple", "Banana", "Cherry"];
```

Listing 5.1: Array Declaration

5.2 Accessing Elements

Elements are accessed using 0-based indexing.

```
print << fruits[0]; // Outputs: Apple
fruits[1] = "Blueberry"; // Changes Banana to Blueberry
```

5.3 Built-in Array Methods

LambdaScript arrays come with powerful built-in methods for manipulation.

`push(value)` Adds an element to the end of the array.

`pop()` Removes and returns the last element of the array.

`remove(value)` Finds the first occurrence of the value and removes it.

`insert(index, value)` Inserts a value at the specified index, shifting subsequent elements.

`len(array)` Global function that returns the size of the array.

```
int[] stack = [];

stack.push(10);
stack.push(20);
stack.push(30);
```

```
6
7 int last = stack.pop(); // Returns 30, stack is now [10, 20]
8 print ≪ "Popped: " ≪ last;
```

Listing 5.2: Array Manipulation

Chapter 6

Functions

Functions are the building blocks of LambdaScript. They allow code reuse and modularity.

6.1 Syntax

A function definition consists of the `func` keyword, a name, a parameter list, a return type arrow (`->`), and the body.

```
// Function that takes two ints and returns an int
func add(int a, int b) → int {
    return a + b;
}

// Function usage
int sum = add(5, 7);
print ≪ "Sum is: " ≪ sum;
```

Listing 6.1: Basic Function

6.2 Void Functions

If a function performs an action but does not return a value, use the `void` return type.

```
func logError(string msg) → void {
    print ≪ "[ERROR]: " ≪ msg;
}
```

Listing 6.2: Void Function

6.3 Higher-Order Functions (Map)

LambdaScript supports functional programming concepts. The `map` method allows you to apply a function to every element in an array.

```
func square(int x) → int {
    return x * x;
}

int[] numbers = [1, 2, 3, 4];

// Applies 'square' to every item in 'numbers'
int[] squaredNumbers = numbers.map(square);
```

```
9
10 // squaredNumbers is now [1, 4, 9, 16]
```

Listing 6.3: Mapping Arrays

Chapter 7

Data Structures (Structs)

Structs allow developers to define custom complex data types. This is LambdaScript's approach to object-oriented data modeling.

7.1 Defining a Struct

Structs are defined using the `struct` keyword. They contain named fields with specific types.

```
struct Point {
    int x;
    int y;
    string label;
}
```

Listing 7.1: Struct Definition

7.2 Instantiation and Usage

To create an instance of a struct, call the struct name like a function, passing arguments in the order defined. Fields are accessed using dot notation.

```
// Create a new Point
Point p1 = Point(10, 20, "Start");

print ≪ "Point " ≪ p1.label ≪ " is at " ≪ p1.x ≪ ", " ≪ p1.y;

// Modify a field
p1.x = 50;
```

Listing 7.2: Using Structs

7.3 Structs in Arrays

Structs are fully compatible with arrays, allowing for complex data sets.

```
struct User {
    string username;
    int id;
}

User[] users = [];
users.push(User("Alice", 1));
```

```
8  users.push(User("Bob", 2));
9
10 print ≪ users[0].username; // Outputs: Alice
```

Chapter 8

Standard Library

LambdaScript includes a set of global functions available in any scope.

8.1 Utility Functions

- **`len(object)`**: Returns the length of a string or an array.

```
    string s = "Hello";
    print ≪ len(s); // Outputs 5
```

- **`str_to_int(string)`**: Parses a string and returns an integer. Useful for processing input.

```
    string inputStr = "123";
    int val = str_to_int(inputStr);
    print ≪ val + 5; // Outputs 128
```

- **`random(min, max)`**: Returns a pseudo-random integer between `min` (inclusive) and `max` (exclusive).

```
    int diceRoll = random(1, 7);
    print ≪ "You rolled a " ≪ diceRoll;
```