

Lambda Script Cheatsheet

Jan 2026 - v0.2

CLI Commands

```
lambda                # Start REPL
// REPL Commands: .quit, .help, .clear
lambda script.ls      # Eval functional script
lambda run script.ls  # Run procedural script
lambda --transpile-only script.ls # Transpile only
lambda --help         # Show help
```

Validation:

```
lambda validate file.json -s schema.ls # With schema
lambda validate file.json              # Default schema
```

Type System

Scalar Types:

```
null bool int float decimal
string symbol binary datetime
```

Container Types:

```
1 to 10           // Range
[123, true]       // Array of values
(0.5, "string:")  // List/tuple
{key: 'symbol'}   // Map
<div class: bold; "text" <br>> // Element
```

Type Operators:

```
int | string      // Union type
int & number      // Intersection
int?             // Optional (int | null)
int*             // Zero or more
int+             // One or more
fn (a: int, b: string) bool // Function type
fn int           // Function type (no params)
{a: int, b: bool} // Map type
<div id:symbol; <br>> // Element type
```

Type Declarations:

```
type User = {name: string, age: int}; // Object type
type Point = (float, float);          // Tuple type
type Result = int | error;            // Union type
```

Literals

Numbers:

```
42          // Integer
3.14        // Float
1.5e-10     // Scientific notation
123.45n     // Decimal (arbitrary precision)
inf nan     // Special values
```

Strings & Symbols:

```
"hello"      // String
"multi-line  // Multi-line string
string"
'symbol'     // Symbol
symbol       // Unquoted symbol
```

Binary & DateTime:

```
b'\xDEADBEEF' // Hex binary
b'\64QUVGRw==' // Base64 binary
t'2025-01-01'  // Date
t'14:30:00'    // Time
t'2025-01-01T14:30:00Z' // DateTime
```

Collections:

```
[1, 2, 3]      // Array
(1, "two", 3.0) // List
{a: 1, b: 2}   // Map
<div id: "main"> // Element
```

Indexing & Slicing:

```
arr[0]          // First element
arr[1 to 3]     // Slice (indices 1, 2, 3)
map.key         // Map field access
map["key"]      // Map field by string
```

Variables & Declarations

Let Expressions:

```
(let x = 5, x + 1, x * 2) // Single binding
(let a = 1, let b = 2, a + b) // Multiple bindings
```

Let Statements:

```
let x = 42; // Variable declaration
let y : int = 100; // With type annotation
let a = 1, b = 2; // Multiple variables
```

Operators

Arithmetic: addition, subtraction, multiplication, division, integer division, modulo, exponentiation

+ - * / div % ^

Comparison: equal, not equal, less than, less equal, greater than, greater equal

== != < <= > >=

Logical: logical and, or, not

and or not

Type & Set: type check, membership, range, union, intersection, exclusion

is in to | & !

Vector Arithmetic: scalar broadcast, element-wise ops

1+[2,3] = [3,4] [1,2]*2 = [2,4] [1,2]+[3,4] = [4,6]

Pipe Expressions

Pipe | with current item ~:

```
[1,2,3] | ~ * 2 // [2,4,6] - map over items
[1,2,3] | sum // 6 - aggregate (no ~)
users | ~.age // [12,20,62] - extract field
['a','b'] | {i:~#, v:~} // ~# = index/key
```

Filter with where:

```
[1,2,3,4,5] where ~ > 3 // [4,5]
users where ~.age >= 18 | ~.name // filter then map
[1,2,3] | ~ ^ 2 where ~ > 5 | sum // 13 (4+9)
```

Pipe to File (procedural only):

```
// Target can be string, symbol, or path
data |> 'output.txt'      // file under CWD
data |> /tmp.'output.txt' // output at full path
data |>> "output.txt"     // append to file
```

```
// Data type determines output format:
// - String: raw text (no formatting)
// - Binary: raw binary data
// - Other types: Lambda/Mark format
```

```
// Output in specific formats:
data | format('json') |> "output.json"
```

Control Flow

If Expressions (require else):

```
if (x > 0) "positive" else "non-positive"
if (score >= 90) "A"
else if (score >= 80) "B" else "C"
```

If Statements (optional else):

```
if (x > 0) { "positive" }
if (condition) { something() } else { otherThing() }
```

For Expressions:

```
for (x in [1, 2, 3]) x * 2      // Array iteration
for (i in 1 to 5) i * i        // Range iteration
for (x in data) if (x > 0) x else 0 // Conditional
```

For Expression Clauses: let, where, order by, limit, offset

```
for (x in data where x > 0) x      // filter
for (x in data, let sq = x*x) sq    // let binding
for (x in [3,1,2] order by x) x     // (1,2,3)
for (x in [3,1,2] order by x desc) x // (3,2,1)
for (x in data limit 5 offset 10) x // pagination
for (x in data, let y=x*2
  where y>5 order by y desc limit 3) y
```

For Statements:

```
for item in collection { transform(item) }
```

Procedural Control (in pn):

```
var x=0;    // Mutable variable
while(c) { break; continue; return x; }
```

Functions

Function Declaration:

```
// Function with statement body
fn add(a: int, b: int) int { a + b }
// Function with expression body
fn multiply(x: int, y: int) => x * y
// Anonymous function
let square = (x) => x * x;
// Procedural function
pn f(n) { var x=0; while(x<n) {x=x+1}; x }
```

Advanced Features:

```
fn f(x?:int)    // optional param
fn f(x=10)      // default param value
fn f(...)       // variadic args
f(b:2, a:1)     // named param call
fn outer(n) { fn inner(x)=>x+n; inner } // closure
```

System Functions

Type:

```
int(v) int64(v) float(v) decimal(v) string(v)
symbol(v) binary(v) number(v) type(v) len(v)
```

Math:

```
abs(x) sign(x) min(a,b) max(a,b) round(x) floor(x)
ceil(x) sqrt(x) log(x) log10(x) exp(x) sin(x)
cos(x) tan(x)
```

Stats:

```
sum(v) avg(v) mean(v) median(v) variance(v)
deviation(v) quantile(v,p) prod(v)
```

Date/Time:

```
datetime() today() now() justnow() date(dt)
time(dt)
```

Collection:

```
slice(v,i,j) set(v) all(v) any(v) reverse(v)
sort(v) unique(v) concat(a,b) take(v,n) drop(v,n)
zip(a,b) fill(n,x) range(a,b,s) map(f,v)
filter(f,v) reduce(f,v,init)
```

Vector:

```
dot(a,b) norm(v) cumsum(v) cumprod(v) argmin(v)
argmax(v) diff(v)
```

I/O:

```
input(file,fmt) format(data,fmt) print(v)
output(data,file) fetch(url,opts) cmd(c,args)
error(msg) varg()
```

Input/Output Formats

Supported Input Types: json, xml, yaml, markdown, csv, html, latex, toml, rtf, css, ini, math, pdf

```
input("path/file.md", 'markdown') // Input Markdown
```

Input with Flavors: e.g. math flavors: latex, typst, ascii

```
input("math.txt", {'type':'math', 'flavor':'ascii'})
```

Output Formatting: json, yaml, xml, html, markdown

```
format(data, 'yaml') // Format as YAML
```

Modules, Imports & Exports

Import Syntax:

```
import module_name;           // Basic import
import .relative_module;      // Relative import
import alias: module_name;    // Import with alias
import mod1, mod2, alias: mod3; // Multiple imports
```

Export Declarations:

```
pub PI = 3.14159;           // Export variable
pub fn square(x) => x * x;   // Export function
```

Module Usage Example:

```
// In math_utils.ls:
pub PI = 3.14159;
pub fn square(x) => x * x;

// In main.ls:
import math: .math_utils;
let area = math.PI * math.square(radius);
```

Error Handling

Creating Errors:

```
error("Something went wrong") // Create error value
```

Error Checking:

```
let result = risky_operation();
if (result is error) { print("Error:", result) }
else { print("Success:", result) }
```

Operator Precedence (High to Low)

1. () [] . - Primary expressions
2. - + not - Unary operators
3. ^ - Exponentiation
4. * / div % - Multiplicative
5. + - - Additive
6. < <= > >= - Relational
7. == != - Equality
8. and - Logical AND
9. or - Logical OR
10. to - Range
11. is in - Type operations

12. | where - Pipe and Filter

Quick Examples

Data Processing:

```
let data = input("sales.json", 'json')
let total = sum(
  (for (sale in data.sales) sale.amount))
let report = {total: total,
  count: len(data.sales)}
format(report, 'json')
```

Function Definition:

```
fn factorial(n: int) int {
  if (n <= 1) 1 else n * factorial(n - 1)
}
```

Element Creation:

```
let article = <article title:"My Article"
  <h1 "Introduction">
    <p "Content goes here.">
  >
format(article, 'html')
```

Comprehensions - Complex data processing:

```
(let data = [1, 2, 3, 4, 5],
let filtered = (for (x in data)
  if (x % 2 == 0) x else 0),
let doubled = (for (x in filtered) x * 2), doubled)
```