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 procedual 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:

```
range, 1 to 10          // Range (inclusive both ends)
array, [123, true]      // Array of values
list, (0.5, "string:)   // List
map, {key: 'symbol'}    // Map
element, <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                        // Same as fn () int
{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:

```
b'\xDEADBEEF'     // Hex binary
b'\64QUVGRw=='    // Base64 binary
```

DateTime:

```
t'2025-01-01T14:30:00Z'  // DateTime
t'2025-04-26' is date       // Sub-types: date
t'10:30:00' is time         // Sub-types: time

// Member properties
dt.date  dt.year  dt.month  dt.day
dt.time  dt.hour  dt.minute dt.second  dt.millisecond
dt.weekday  dt.yearday  dt.week  dt.quarter
dt.unix  dt.timezone  dt.utc_offset  dt.utc  dt.local

// Formatting
dt.format("YYYY-MM-DD")  dt.format('iso')

// Constructors
datetime()  today()  justnow()   // current date/time
datetime(2025, 4, 26, 10, 30)
date(2025, 4, 26)  time(10, 30, 45)
```

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.0             // Alt. syntax for const index
arr[1 to 3]       // Slice (indices 1, 2, 3)
map.key           // Map field access
map["key"]        // Map field by string
```

Namespaces Support:

```
namespace svg: 'http://www.w3.org/2000/svg'
namespace xlink: 'http://www.w3.org/1999/xlink'

<svg.rect svg.width: 100>   // Namespaced tag & attr
elem.svg.width              // Namespaced member access
svg.rect                    // Qualified symbol
symbol("href", 'xlink_url') // Create namespaced symbol
```

Variables & Declarations

Let Expressions:

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

Let Statements (immutable):

```
let x = 42;               // Immutable binding
let y : int = 100;        // With type annotation
let a = 1, b = 2;         // Multiple bindings
x = 10       // ERROR E211: cannot reassign let binding
```

Var Statements (mutable, pn only):

```
var x = 0;          // Mutable variable
var y: int = 42;    // With type annotation
x = x + 1           // OK: reassignment
x = "hello"         // OK: type widening (int → string)
y = "oops"      // ERROR E201: annotated type enforced
```

Operators

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

```
+  -  *  /  div  %  **
```

Spread: *

```
let a = [1, 2, 3]
(*a, *[10, 20])    // (1, 2, 3, 10, 20)
```

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  |  &  !
```

Query: type-based descendant search

```
?   .?
```

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):

Query Expressions

Query `?` — recursive descendant search (like XPath `//`):

```
html?<img>               // all <img> at any depth
html?<div class: string>  // <div> with class attr
data?int                 // all int values in tree
data?(int | string)      // all int or string values
data?{name: string}      // maps with string 'name'
data?{status: "ok"}      // maps where statu == "ok"
42?int                   // (42) - self-match
```

Direct query `.?` — self + attributes + direct children only (like XPath `/`):

```
body.?<p>      // only <p> directly under body
map.?string    // string values at top level of map
el.?int        // int values in attrs + direct content
```

Results in document order (depth-first, pre-order). Searches attributes (including complex values), children, and map/array items recursively.

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() }
```

Match Expressions:

```
// Type patterns or Literal
match value {
    case 200: "OK"               // case literal
    case string: "text"          // case type
    case int | float: "numnber"  // case type union
    case Circle {                // mixed stam arm
        let area = 3.14 * ~.r ** 2
        "area: " ++ area
    }
    default: "other"
}
```

For Expressions: (produce spreadable arrays)

```
for (x in [1, 2, 3]) x * 2     // [2, 4, 6]
for (i in 1 to 5) i * i        // [1, 4, 9, 16, 25]
for (x in data) if (x > 0) x else 0  // Conditional
// Nested for-expressions flatten
[for (i in 1 to 2) for (j in 1 to 2) i*j]  // [1,2,2,4]
// Empty for produces spreadable null (skipped)
[for (x in [] ) x]             // []

// for loop over map by keys
for (k at {a: 1, b: 2}) k      // "a", "b"
for (k, v at {a: 1, b: 2}) k ++ v  // ["a1", "b2"]
```

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; }
```

Assignment Targets:

```
x = 10              // Variable reassignment (var only)
arr[i] = val        // Array element reassignment
obj.field = val     // Map field reassignment
elem.attr = val     // Element attribute reassignment
elem[i] = val       // Element child reassignment
```

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)`

Range:

`s to e` creates a range from `s` to `e` (inclusive both ends). `range(s,e,step)` creates a range with custom step (exclusive end).

```
1 to 5                // [1, 2, 3, 4, 5]
range(0, 10, 2)       // [0, 2, 4, 6, 8]
```

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(s,e,step)` `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("Message")    error("load failed", inner_err)
error({code: 304, message: "div by zero"})
```

Error Return Types (`T^E`):

```
fn parse(s: string) int^ { ... }           // Return int or
fn divide(a, b) int ^ DivisionError { ... }    // Specific
fn load(p) Config ^ ParseError|IOError { ... } // Multiple
```

raise error , or propagate error with `^`

```
fn compute(x: int) int^ {
    if (b == 0) raise error("div by zero")  // raise error
    let a = parse(input)^    // error → return immediately
    let b = divide(a, x)^    // error → return immediately
    a + b
}
fun()^                        // propagate error, discard v
```

`let a^err` — destructure value and error:

```
let result^err = divide(10, x)
if (^err) print("Err: " ++ err.message)  // ^err to check e
else result * 2
```

Operator Precedence (High to Low)

1. `() [] . ? .?` - Primary, query
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)
```