# Lambda Script Cheatsheet

## CLI Commands

```
lambda                    # Start REPL
// REPL Commands: :quit, :help, :clear
lambda script.ls          # Run 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
[int]           // Array of integers
(int, string)   // List/tuple
{key: string}   // Map
<tag attr: int> // Element
```

### Type Operators:

```
int | string    // Union type
int & number    // Intersection
int?            // Optional (int | null)
int*            // Zero or more
int+            // One or more
(a: int, b: string) => bool   // Function 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
```

### 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
```

## Variables & Declarations

### Let Expressions:

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

### Let Statements:

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

### Public Declarations:

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

## Operators

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

```
+  -  *  /  _/  %  ^
```

**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  |  &  !
```

## 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
for (i in 1 to 5) i * i         // Range
for (x in data) if (x > 0) x else 0  // Conditional
```

### For Statements:

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

## 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;
```

### Function Calls:

```
add(5, 3)        // Function call
```

## System Functions

### Type Conversion:

```
int("42")        // String to int
float("3.14")    // String to float
string(42)       // Value to string
symbol("text")   // String to symbol
```

### Type Inspection:

```
type(value)      // Get type of value
len(collection)  // Get length
```

### Math Functions:

```
abs(x)           // Absolute value
min(a, b, c)     // Minimum value
max(a, b, c)     // Maximum value
sum([1, 2, 3])   // Sum of array
avg([1, 2, 3])   // Average of array
round(x)  floor(x)  ceil(x)  // Rounding
```

### Date/Time Functions:

```
datetime()       // Current date/time
today()          // Current date
justnow()        // Current time
date(dt)         // Extract date part
time(dt)         // Extract time part
```

### Collection Functions:

```
slice(arr, start, end)    // Array slice
set(arr)                  // Remove duplicates
all([true, false])        // All true?
any([false, true])        // Any true?
```

### I/O Functions:

```
input(file, format)       // Parse file
print(value)              // Print to console
format(data, type)        // Format output
error(message)            // Create error
```

## 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
```

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

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

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

```
format(data, 'yaml')                //
```

## Modules & Imports
### Import Syntax:

```
import module_name;              // B
import .relative_module;         // R
import alias: module_name;       // I
import mod1, mod2, alias: mod3;  // M
```

### 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) }
```

## Advanced Features
### Type Declarations:

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

### 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)
```

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

## 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')
```