JSL - JSON Serializable Language

None

None

None

Table of contents

1. JSL - JSON Serializable Language

1.1 A Network-Native Functional Programming Language

JSL is a Lisp-like functional programming language designed from the ground up for network transmission and distributed computing. Unlike traditional languages that treat serialization as an afterthought, JSL makes wire-format compatibility a first-class design principle.

In an era of distributed systems and microservices, JSL addresses common challenges in code mobility, runtime dependencies, and cross-platform interoperability by treating JSON as the canonical representation for both data and code.

1.2 Key Features

- 🔄 **Network-Native**: Every JSL program is valid JSON that can be transmitted over networks
- 🔒 **Secure by Design**: Host environment controls all capabilities and side effects
- 📦 **Closure Serializability**: Functions with captured environments can be serialized and reconstructed
- 🎯 **Homoiconic**: Code and data share the same JSON representation
- ⚡ **Deterministic**: Core language evaluation is predictable and reproducible
- 🔧 **Extensible**: Built-in prelude provides practical functionality
- 🏗️ **First-Class Objects**: JSON objects are native data structures with dynamic construction support

1.3 Core Design Principles

JSL is built upon fundamental principles that guide every aspect of its design:

- **JSON as Code and Data**: All JSL programs and data structures are representable as standard JSON. This ensures universal parsing, generation, and compatibility with a vast ecosystem of tools and platforms.
- **Network-Native**: The language is designed for seamless transmission over networks. Its serialization format is inherently web-friendly and requires no complex marshalling/unmarshalling beyond standard JSON processing.
- **Serializable Closures**: JSL provides a mechanism for serializing closures, including their lexical environments (user-defined bindings), allowing functions to be truly mobile.
- **Effect Reification**: Side-effects are not executed directly within the core language evaluation but are described as data structures, allowing host environments to control, audit, or modify them.
- **Deterministic Evaluation**: The core JSL evaluation (excluding host interactions) is deterministic, facilitating testing, debugging, and predictable behavior.
- **Security through Capability Restriction**: The host environment governs the capabilities available to JSL programs, particularly for side-effecting operations.

1.4 Quick Example

```
// Define and call a factorial function using a local binding
["let", [
  ["factorial",
    ["lambda", ["n"],
      ["if", ["<=", "n", 1],
        1,
        ["*", "n", ["factorial", ["-", "n", 1]]]
      ]
    ]
  ]
],
  ["factorial", 5]
]
// → 120
```

This JSL program:

1. **Is valid JSON** - can be stored, transmitted, and parsed by any JSON-compliant system
2. **Defines a function** - creates a recursive factorial function
3. **Captures closures** - the function can be serialized with its environment
4. **Produces a result** - evaluates to 120

1.5 Theoretical Foundations

JSL draws inspiration from several key concepts in computer science and programming language theory:

- **Homoiconicity**: Like Lisp, JSL code and data share the same structural representation. However, JSL uses JSON arrays and objects instead of S-expressions, leveraging JSON's widespread adoption and strict schema.
- **Lexical Scoping and Closures**: JSL employs lexical scoping. Functions (`lambda` forms) can capture variables from their surrounding lexical environments, forming closures. The serialization mechanism is designed to preserve these captured environments.
- **Functional Programming**: JSL encourages a functional programming style, emphasizing immutability, first-class functions, and expressions over statements.
- **Separation of Pure Computation and Effects**: The core JSL interpreter deals with pure computation. Interactions with the external world (I/O, system calls) are managed via the JSL Host Interaction Protocol (JHIP), where effects are requested as data.

1.6 Why JSL?

1.6.1 The Problem with Traditional Code Mobility

Modern distributed systems need to move code between services, store executable logic in databases, and update running systems dynamically. Traditional approaches face fundamental limitations:

- **Serialization brittleness** - Complex frameworks that break across versions
- **Runtime dependencies** - Code tied to specific environments and libraries
- **Security vulnerabilities** - Deserializing arbitrary code creates attack vectors
- **Platform lock-in** - Language-specific formats prevent interoperability

1.6.2 The JSL Solution

JSL solves these problems by making JSON the native representation for both code and data:

- **Universal compatibility** - Works with any system that supports JSON
- **Intrinsic safety** - Transmitted code contains no executable primitives
- **Runtime independence** - Compatible prelude provides computational foundation
- **Cross-platform** - Language-agnostic JSON representation

1.7 Use Cases

JSL's design makes it suitable for a variety of applications:

- **Distributed Computing**: Send computations to where data resides, reducing network overhead and improving performance
- **Edge Computing**: Deploy and update logic on edge devices dynamically without full redeployment
- **Serverless Functions / FaaS**: Represent functions as JSON, simplifying deployment and management
- **Database Functions**: Store and execute business logic directly in databases in a portable format
- **Microservice Communication**: Share functional components across service boundaries with guaranteed compatibility
- **Code as Configuration**: Express complex configurations as executable programs that can be validated and tested
- **Workflow Automation**: Define complex workflows as JSL programs that can be stored, versioned, and executed anywhere
- **Plugin Systems**: Allow users to extend applications with sandboxed, serializable plugins
- **Live Programming**: Update running systems by transmitting new code without service interruption

1.8 Getting Started

1. **Getting Started** - Set up JSL in your environment and learn the basics.
2. **Language Guide** - Learn the syntax and semantics
3. **Tutorials** - Step-by-step examples

1.9 Architecture Overview

JSL consists of three layers:

1. **Prelude Layer** - Non-serializable built-in functions that form the computational foundation
2. **User Layer** - Serializable functions and data defined by user programs
3. **Wire Layer** - JSON representation for transmission and storage

This separation ensures transmitted code is always safe while remaining fully functional when reconstructed with a compatible prelude.

1.10 Learn More

- **Design Philosophy** - Theoretical foundations and principles
- **AST Specification** - Formal language syntax definition
- **JHIP Protocol** - Host interaction for side effects
- **API Reference** - Complete function documentation

2. Getting Started with JSL

This guide provides a concise introduction to JSL to get you up and running in minutes. JSL is a lightweight, functional programming language that uses JSON for its syntax, making it ideal for data manipulation, configuration, and network-native applications.

2.1 Installation

JSL requires Python 3.8 or later.

2.1.1 From Source

Clone the repository and install JSL using pip:

```
git clone https://github.com/queelius/jsl.git
cd jsl
pip install -e .
```

2.1.2 Verify Installation

You can verify the installation by running a simple program or by starting the interactive REPL:

```
# Run a simple JSL program from the command line
echo '["print", "@Hello, JSL!"]' | jsl

# Start the interactive REPL
jsl --repl
```

2.2 Your First JSL Program

JSL programs are simply JSON data structures. Create a file named `hello.jsl` with the following content:

```
["print", "@Hello, World!"]
```

Execute it from your terminal:

```
jsl hello.jsl
```

You should see the output: `Hello, World!`

Note on File Extensions We recommend using the `.jsl` extension for your JSL program files. This helps distinguish them from regular JSON data files and allows for better editor integration. However, the `jsl` interpreter will happily run files with a `.json` extension, preserving the principle that all JSL code is valid JSON.

2.3 Core Concepts

2.3.1 Literals and Variables

Standard JSON literals like numbers, booleans, and `null` evaluate to themselves. Strings are used for both literal text and variable references. A string with an `@` prefix is a literal, while a string without it is treated as a variable.

```
42          // A number
"@hello"    // A string literal
"my_variable" // A reference to a variable
```

2.3.2 Basic Operations

JSL uses prefix notation (like Lisp) for function calls. The first element of an array is the function to be called, and the rest are its arguments.

```
["+", 1, 2, 3]
```

This expression evaluates to `6`.

2.3.3 Defining Variables and Functions

You can define variables with `def` and functions with `lambda`. The `do` special form lets you execute a sequence of expressions.

```
[
  "do",
  ["def", "x", 10],
  ["def", "square", ["lambda", ["n"], ["*", "n", "n"]]],
  ["square", "x"]
]
```

This evaluates to `100`.

2.3.4 Conditional Logic

The `if` special form provides conditional evaluation:

```
["if", [">", 5, 3], "@Greater", "@Less"]
```

This evaluates to `"Greater"`.

2.4 A Quick Example: Fibonacci

Here is a more complete example that defines a recursive function to compute Fibonacci numbers and then applies it to a list of numbers:

```
[
  "do",
  ["def", "fib",
   ["lambda", ["n"],
    ["if", ["<=", "n", 1],
     "n",
     ["+", ["fib", ["-", "n", 1]], ["fib", ["-", "n", 2]]]]]],
  ["map", "fib", ["list", 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
]
```

2.5 Running JSL Code

You can run JSL programs in several ways:

- **From a file:** `jsl your_program.jsl`
- **From standard input:** `echo '["+", 1, 2]' | jsl`
- **Using the REPL:** `jsl --repl`
- **As a web service:** `jsl --service`

2.6 Next Steps

Now that you have a basic understanding of JSL, you can explore the following sections for more in-depth information:

- **Language Guide**: For a comprehensive overview of JSL's syntax and semantics.
- **Tutorials**: For guided, step-by-step lessons.
- **Examples**: For a collection of practical, real-world examples.

3. Language Guide

3.1 JSL Language Overview

3.1.1 What is JSL?

JSL (JSON Serializable Language) is a powerful, functional programming language where every program is valid JSON. This unique design makes it perfect for network transmission, distributed computing, and safe, sandboxed execution.

JSL is built on three core principles: 1. **Homoiconic**: Code and data share the same structure (JSON). This means you can build and manipulate code as easily as you handle data. 2. **Functional**: With features like immutability, first-class functions, and a rich library of higher-order functions, JSL encourages a clean, declarative style. 3. **Serializable**: Every JSL value, including functions with their environments, can be perfectly serialized to a string, sent across a network, and safely executed on a remote machine.

3.1.2 A Taste of JSL

Because JSL is JSON, operations use a simple prefix notation inside an array. This expression adds three numbers:

```
["+", 1, 2, 3]
```

The evaluator understands that the first element, `"+"`, is a function to be applied to the rest of the elements, resulting in `6`.

3.1.3 The Two Modes of JSL

JSL has a single, consistent evaluation model where:

Core Evaluation Rules

1. **Strings without `@`**: Variable lookups (e.g., `"x"` looks up the value of x)
2. **Strings with `@`**: Literal strings (e.g., `"@hello"` is the string "hello")
3. **Arrays**: Function calls in prefix notation (e.g., `["+", 1, 2]`)
4. **Objects**: Data structures with evaluated keys and values
5. **Other values**: Self-evaluating (numbers, booleans, null)

JSON Object Construction

JSL treats JSON objects as first-class data structures. Unlike arrays (which are interpreted as S-expressions), objects are always data structures, never function calls. This makes them perfect for constructing pure JSON output:

```
["do",
  ["def", "user", "@Alice"],
  ["def", "role", "@admin"],
  {"@name": "user", "@role": "role"}
]
// Result: {"name": "Alice", "role": "admin"}
```

For a complete guide, see **JSON Objects as First-Class Citizens**.

3.1.4 Key Language Features

- **Special Forms**: A small set of core keywords like `if`, `def`, and `lambda` provide the foundation for control flow and variable bindings. See the **Special Forms Guide**.
- **Rich Prelude**: A comprehensive standard library of functions for math, logic, and data manipulation is available everywhere. See the **Prelude Reference**.

- **Lexical Scoping**: JSL uses lexical scoping, meaning functions (closures) capture the environment where they are defined, not where they are called. This provides a robust and predictable module system.
- **Host Interaction**: JSL interacts with the host system through a single, explicit special form, `["host", ...]`, making all side effects transparent and auditable.

3.1.5 A Complete Example

This example defines and calls a recursive factorial function, showcasing variable and function definition (`def`, `lambda`), conditional logic (`if`), and a sequence of operations (`do`).

```
["do",
  ["def", "factorial",
    ["lambda", ["n"],
      ["if", ["<=", "n", 1],
        1,
        ["*", "n", ["factorial", ["-", "n", 1]]]
      ]
    ]
  ],
  ["factorial", 5]
]
```

The result of evaluating this expression is `120`.

3.1.6 Where to Go Next

- **JSL Syntax and Semantics**: The definitive guide to writing and understanding core JSL.
- **JSON Objects**: Learn how to generate dynamic JSON objects.
- **Special Forms**: A detailed reference for all core language constructs.
- **Prelude Functions**: A complete catalog of all built-in functions

3.2 JSL Language Specification v1.0

3.2.1 1. Introduction

JSL (JSON Serializable Language) is a functional programming language where all programs and data are representable as valid JSON. The language has two representations:

1. **Source Language (S-expressions)**: Human-readable JSON arrays
2. **Target Language (JPN - JSL Postfix Notation)**: Stack-based format for execution

3.2.2 2. Formal Grammar

2.1 Source Language (S-expressions)

```
<expr> ::= <literal>
         | <variable>
         | <list-expr>

<literal> ::= <number> | <boolean> | <null> | <string-literal>

<number> ::= JSON number
<boolean> ::= true | false
<null> ::= null
<string-literal> ::= "@" <string>  ; String starting with @ is literal

<variable> ::= <string>  ; String without @ prefix

<list-expr> ::= "[" <operator> <expr>* "]"

<operator> ::= <string>

<string> ::= JSON string
```

2.2 Target Language (JPN - JSL Postfix Notation)

```
<jpn-program> ::= <instruction>*

<instruction> ::= <literal>
                | <variable>
                | <arity> <operator>

<arity> ::= non-negative integer
<operator> ::= <string>
```

Key Design Decision: JPN always encodes arity before the operator as two consecutive elements `[arity, operator]`. This provides: - JSON compatibility (no tuples, only arrays) - Consistent parsing (always look ahead for operator after seeing integer) - Efficient execution (know exactly how many values to pop) - Support for identity elements (0-arity operations)

Note on Further Compilation: The JPN representation could be compiled to raw bytecode (e.g., `[PUSH_INT, 2, ADD]` or `[0x12, 0x02, 0x20]`), but JSL deliberately maintains JSON compatibility for network transparency, debuggability, and universal portability. See Performance Philosophy for rationale.

3.2.3 3. Compilation Rules

The compilation from S-expressions to JPN follows these rules:

3.1 Literals

```
compile(n) = [n]           where n is number/boolean/null
compile("@s") = ["@s"]     where s is a string
```

3.2 Variables

```
compile("x") = ["x"]       where x doesn't start with @
```

3.3 List Expressions

```
compile([op, e₁, ..., eₙ]) = compile(e₁) ++ ... ++ compile(eₙ) ++ [n, op]

where:
  n is the arity (number of arguments)
  ++ denotes list concatenation
```

3.4 Special Cases

```
compile([]) = [0, "__empty_list__"]
```

3.5 Examples

```
compile(['+', 2, 3])       = [2, 3, 2, '+']
compile(['+'])             = [0, '+']
compile(['*', 2, 3, 4])    = [2, 3, 4, 3, '*']
compile(['list', 1, 2, 3]) = [1, 2, 3, 3, 'list']
```

3.2.4 4. Abstract Machine

The JSL abstract machine is a stack-based architecture with the following components:

4.1 Machine State

```
State = {
  stack: Value[],        // Value stack
  pc: Int,               // Program counter
  code: Instruction[],   // Postfix instructions
  env: Map<String, Value>, // Environment for variables
  resources: ResourceBudget  // Optional resource limits
}
```

4.2 Operational Semantics

The machine executes according to these transition rules:

LITERAL PUSH

```
(S, pc, code[pc] = v, E) → (S·v, pc+1, code, E)
  where v is a literal value
```

VARIABLE LOOKUP

```
(S, pc, code[pc] = x, E) → (S·E(x), pc+1, code, E)
  where x is a variable name and E(x) is defined
```

ARITY-OPERATOR PAIR

```
(S·vₙ·...·v₁, pc, code[pc] = n, code[pc+1] = op, E) → (S·op(v₁,...,vₙ), pc+2, code, E)
  where n is the arity and op is the operator
```

TERMINATION

```
([v], pc, code) → v
  where pc ≥ |code|
```

4.3 Notation

- `S·v` denotes pushing value v onto stack S
- `E(x)` denotes looking up variable x in environment E
- `|code|` denotes the length of the code array

3.2.5 5. Built-in Operators

5.1 Arithmetic Operators

Operator	Arity	Semantics	Example
`+`	0	Sum identity: 0	`[+]` → 0
`+`	1	Identity	`[+, 5]` → 5
`+`	2	Addition	`[+, 2, 3]` → 5
`+`	n	Sum	`[+, 1, 2, 3]` → 6
`-`	1	Negation	`[-, 5]` → -5
`-`	2	Subtraction	`[-, 10, 3]` → 7
`*`	0	Product identity: 1	`[*]` → 1
`*`	2	Multiplication	`[*, 3, 4]` → 12
`*`	n	Product	`[*, 2, 3, 4]` → 24
`/`	2	Division	`[/, 10, 2]` → 5
`%`	2	Modulo	`[%, 10, 3]` → 1

5.2 Comparison Operators

Operator	Arity	Semantics
`=`	2	Equality
`!=`	2	Inequality
`<`	2	Less than
`>`	2	Greater than
`<=`	2	Less or equal
`>=`	2	Greater or equal

5.3 Logical Operators

Operator	Arity	Semantics
`not`	1	Logical negation
`and`	2	Logical AND
`or`	2	Logical OR

5.4 List Operators

Operator	Arity	Semantics	Example
`list`	n	Create list	`[list, 1, 2, 3]` → [1,2,3]
`cons`	2	Prepend element	`[cons, 1, [2, 3]]` → [1,2,3]
`first`	1	Get first element	`[first, [1, 2]]` → 1
`rest`	1	Get tail	`[rest, [1, 2, 3]]` → [2,3]
`append`	2	Append element	`[append, [1, 2], 3]` → [1,2,3]
`length`	1	List length	`[length, [1, 2]]` → 2

3.2.6 6. Special Forms

Special forms have unique evaluation rules and are not compiled to postfix in the current implementation:

6.1 Conditional

```
[if, condition, then-expr, else-expr]
```

Evaluates `condition`, then evaluates either `then-expr` or `else-expr`.

6.2 Let Binding

```
[let, [var, value], body]
```

Evaluates `value`, binds it to `var`, then evaluates `body` in extended environment.

6.3 Lambda

```
[lambda, [param₁, ..., paramₙ], body]
```

Creates a closure capturing current environment.

6.4 Definition

```
[def, var, value]
```

Evaluates `value` and binds it to `var` in current environment.

6.5 Sequencing

```
[do, expr₁, ..., exprₙ]
```

Evaluates expressions in sequence, returns last value.

6.6 Quote

```
[quote, expr] or [@, expr]
```

Returns `expr` without evaluation.

3.2.7 7. Type System

JSL is dynamically typed with the following value types:

```
Value = Number
      | Boolean
      | Null
      | String
      | List<Value>
      | Closure
      | Dict<String, Value>
```

3.2.8 8. Memory Model

8.1 Environment Chain

Environments form a linked chain for lexical scoping:

```
Env = {
  bindings: Map<String, Value>,
  parent: Env | null
}
```

8.2 Closure Representation

```
Closure = {
  params: String[],
  body: Expr,
  env: Env
}
```

3.2.9 9. Resource Management

The abstract machine can enforce resource limits:

```
ResourceBudget = {
  gas: Int,           // Computational steps
  memory: Int,        // Memory usage
  time_limit: Time,   // Wall-clock time
  collection_limit: Int  // Max collection size
}
```

Each instruction consumes gas: - Literals: 1 gas - Variable lookup: 2 gas - Binary operation: 3 gas - Function call: 10 gas

3.2.10 10. Serialization

All JSL values must be JSON-serializable:

10.1 Primitive Serialization

```
serialize(n: Number) = n
serialize(b: Boolean) = b
serialize(null) = null
serialize(s: String) = s
```

10.2 Compound Serialization

```
serialize([v₁, ..., vₙ]) = [serialize(v₁), ..., serialize(vₙ)]
serialize({k₁: v₁, ..., kₙ: vₙ}) = {k₁: serialize(v₁), ..., kₙ: serialize(vₙ)}
```

10.3 Closure Serialization

```
serialize(Closure{params, body, env}) = {
  "type": "closure",
  "params": params,
  "body": serialize(body),
```

```
  "env": serialize_env(env)
}
```

3.2.11 11. Decompilation

The decompilation from postfix to S-expressions follows these rules:

11.1 Stack-Based Decompilation Algorithm

```
decompile(postfix):
  stack = []
  for instruction in postfix:
    if instruction is literal or variable:
      stack.push(instruction)
    elif instruction is (op, arity):
      args = []
      for i in 1..arity:
        args.prepend(stack.pop())
      stack.push([op] + args)
    elif instruction is binary_op:
      right = stack.pop()
      left = stack.pop()
      stack.push([op, left, right])
  return stack[0]
```

3.2.12 12. Formal Properties

12.1 Compilation Correctness

For any valid S-expression `e` and environment `E`:

```
eval(e, E) = exec(compile(e), E)
```

12.2 Roundtrip Property

For any valid S-expression `e`:

```
decompile(compile(e)) ≡ e
```

Where ≡ denotes structural equivalence.

12.3 Serialization Safety

For any JSL value `v`:

```
deserialize(serialize(v)) = v
```

12.4 Resumption Safety

For any partial execution state `S`:

```
exec_partial(code, S) = exec_complete(code)
```

When given sufficient resources.

3.2.13 13. Examples

13.1 Arithmetic Expression

Source:

```
["*", ["+", 2, 3], ["-", 10, 6]]
```

Postfix:

```
[2, 3, "+", 10, 6, "-", "*"]
```

Execution trace:

```
[]          | 2 3 + 10 6 - *  ; Initial
[2]         | 3 + 10 6 - *    ; Push 2
[2,3]       | + 10 6 - *      ; Push 3
[5]         | 10 6 - *        ; Apply +
[5,10]      | 6 - *           ; Push 10
[5,10,6]    | - *             ; Push 6
[5,4]       | *               ; Apply -
[20]        |                 ; Apply *
```

13.2 Variable Expression

Source:

```
["+", "x", ["*", "y", 2]]
```

Postfix (with environment `{x: 10, y: 3}`):

```
["x", "y", 2, "*", "+"]
```

Execution:

```
[]          | x y 2 * +  ; Initial
[10]        | y 2 * +    ; Lookup x
[10,3]      | 2 * +      ; Lookup y
[10,3,2]    | * +        ; Push 2
[10,6]      | +          ; Apply *
[16]        |            ; Apply +
```

3.2.14 14. Implementation Notes

1. **Tail Call Optimization**: Not currently implemented
2. **Lazy Evaluation**: Not supported (strict evaluation)
3. **Type Checking**: Dynamic only
4. **Garbage Collection**: Relies on host language (Python)
5. **Concurrency**: Not supported

3.2.15 15. Future Extensions

1. **Pattern Matching**: Destructuring in let and lambda
2. **Module System**: Namespace management
3. **Type Annotations**: Optional static typing
4. **Continuations**: First-class continuations for control flow
5. **Parallel Execution**: Parallel evaluation of independent expressions

---

This specification defines JSL v1.0. The language is designed for network-native computation with complete serializability.

3.3 Syntax and Semantics

3.3.1 Overview

JSL uses JSON as its native syntax, making it both human-readable and machine-parseable. Every JSL program is a valid JSON value, and this document describes how those values are interpreted and evaluated.

3.3.2 Core Concepts

1. **Homoiconicity**: Code and data share the same fundamental representation (JSON).
2. **Evaluation Environments**: Expressions are evaluated within a specific environment that holds variable bindings and links to a parent.
3. **Lisp-like Evaluation**: The evaluation logic follows a pattern similar to Lisp, with prefix notation for function calls.

3.3.3 Syntax and Evaluation Rules

1. Literals (Self-Evaluating Values)

Most JSON primitives are literals; they evaluate to themselves.

- **Numbers**: `42` → `42`
- **Booleans**: `true` → `true`
- **Null**: `null` → `null`
- **Objects**: `{"key": "value"}` → `{"key": "value"}`

2. Strings: Literals vs. Variables

The interpretation of a string depends on its syntax.

- **Variable Reference**: A standard string is treated as a variable lookup. The evaluator will search the current environment for its value.
- `"x"` → The value bound to the name `x`.
- **String Literal**: A string prefixed with `@` is treated as a literal value.
- `"@hello"` → The string `"hello"`.

3. Arrays: Function Calls and Special Forms

Arrays are the primary mechanism for computation in JSL. An array is evaluated by inspecting its first element.

- **Empty Array**: An empty array `[]` evaluates to itself, representing an empty list.
- **Function Call**: If the first element is not a special form, the array represents a function call.

a. All elements of the array (the operator and all arguments) are evaluated in order.

b. The result of the first element (which must be a function) is applied to the results of the remaining elements.

c. `["+", 1, "x"]` → Evaluates `+`, `1`, and `x`, then applies the addition function to the results.

- **Special Forms**: If the first element is a special form, a unique evaluation rule is applied. These forms provide the core control flow and structural logic of the language. Not all arguments are necessarily evaluated.

Form	Syntax	Description
`def`	`["def", "name", expr]`	Binds the result of `expr` to `name` in the current environment.
`lambda`	`["lambda", [params], body]`	Creates a function (closure). Does not evaluate the body.
`if`	`["if", cond, then, else]`	Evaluates `cond`. If truthy, evaluates `then`, otherwise evaluates `else`.
`do`	`["do", expr1, expr2, ...]`	Evaluates expressions in sequence, returning the result of the last one.
`quote` / `@`	`["quote", expr]` or `["@", expr]`	Returns `expr` as literal data without evaluating it.
`try`	`["try", body, handler]`	Evaluates `body`. If an error occurs, evaluates `handler` with the error.
`host`	`["host", cmd, ...]`	Sends a request to the host environment.

Complete Example

This expression calculates the factorial of 5, demonstrating variable definition (`def`), function creation (`lambda`), conditional logic (`if`), and recursive function calls.

```
["do",
  ["def", "factorial",
    ["lambda", ["n"],
      ["if", ["<=", "n", 1],
        1,
        ["*", "n", ["factorial", ["-", "n", 1]]]
      ]
    ]
  ],
  ["factorial", 5]
]
```

3.3.4 JSL for Data Construction: JSON Objects as First-Class Citizens

While the semantics described above define JSL as a general-purpose computation language, JSL also treats **JSON objects as first-class data structures** with special properties that make them ideal for data construction.

Unlike arrays (which are interpreted as S-expressions), JSON objects are always treated as data structures: - **Objects are never function calls**: `{"name": "Alice"}` is always a data structure - **Keys and values are evaluated**: Both use normal JSL evaluation rules - **Keys must be strings**: Runtime type checking ensures valid JSON output - **No operator ambiguity**: Objects provide a "safe zone" for pure data construction

This design makes JSL particularly well-suited for generating clean JSON output without worrying about the first element being interpreted as an operator.

For a complete guide on object construction, see the **JSON Objects** documentation.

3.4 Environments and Execution Contexts

3.4.1 Overview: The Scope Chain

Environments are a fundamental concept in JSL, forming the backbone of its lexical scoping, security model, and module system. An environment is a data structure that maps variable names to their values. Every JSL expression is evaluated within an environment.

When looking up a variable, the JSL runtime first checks the current environment. If the variable is not found, it proceeds to check the parent environment, and so on, creating a "scope chain." This process continues until it reaches the root environment, which is the **Prelude**. The prelude provides all the built-in functions and is implicitly the ultimate parent of all user-defined environments.

3.4.2 The Algebra of Environments

This algebra provides the foundation for JSL's **Level 2 (Advanced) Capability-Based Security Model**. While the standard security model relies on the host dispatcher to authorize requests (see Security Model), the environment algebra allows a trusted orchestrator to provision sandboxed execution contexts that cannot even *attempt* to call unauthorized host commands.

By using operators like `remove` to withhold the raw `["host", ...]` capability and `layer` to provide safe, pre-defined wrapper functions, a host can enforce security at the language level, before a request ever reaches the dispatcher.

For a complete discussion of the layered security model, see the Security Model documentation.

These operations are typically exposed to trusted code (e.g., via the Fluent Python API or a special host configuration) and are essential for creating custom execution contexts. They take one or more environment hashes as input and produce a **new** environment hash as output, never modifying the original environments.

`layer` (Union / Additive Merge)

The `layer` operation creates a new environment by combining the bindings from one or more existing environments on top of a shared parent.

- **Syntax:** `["layer", parent_env_hash, env_hash_1, env_hash_2, ...]`
- **Use Case:** Module composition. You can load multiple modules (each represented by an environment) and `layer` them together to create a single, unified API scope for your application.
- **Conflict Resolution:** If multiple source environments define a binding with the same name, the one from the last environment in the argument list ("last-write-wins") is used.

`remove` (Subtraction / Capability Reduction)

The `remove` operation creates a new, less-privileged environment by removing specified bindings.

- **Syntax:** `["remove", env_hash, "key_to_remove_1", "key_to_remove_2", ...]`
- **Use Case:** Sandboxing and security. If you have a powerful `file_system` module, you can use `remove` to create a "read-only" version of it for untrusted code by removing the `write` and `delete` bindings.

`intersect` (Intersection of Capabilities)

The `intersect` operation creates a new environment containing only the bindings whose names exist in *all* of the provided environments.

- **Syntax:** `["intersect", env_hash_1, env_hash_2, ...]`
- **Use Case:** Enforcing an API interface. You can `intersect` two versions of a module to create an environment that is guaranteed to only contain the functions common to both, making your code more robust against API changes.

`difference` (Exclusive Capabilities)

The `difference` operation creates a new environment containing only the bindings from a base environment whose names do *not* exist in another.

- **Syntax:** `["difference", base_env_hash, env_to_subtract_hash]`
- **Use Case:** Introspection and tooling. This can be used to identify new or deprecated features between two versions of a module.

3.4.3 Environments and Serialization

While the concept of environments is part of the language specification, their representation for transport and storage is an architectural detail. For more information on how environments are serialized using content-addressable hashing, see Code and Data Serialization.

3.5 JSL Special Forms

3.5.1 Overview

Special forms are the fundamental building blocks of JSL that have special evaluation rules. Unlike regular function calls, special forms control how their arguments are evaluated and provide the core language constructs for variable binding, control flow, and meta-programming.

3.5.2 Core Special Forms

Variable Definition - `def`

Binds a value to a variable name in the current environment.

```
["def", "variable_name", value_expression]
```

Evaluation Rules: 1. Evaluate `value_expression` 2. Bind the result to `variable_name` in current environment 3. Return the bound value

Examples:

```
// Simple value binding
["def", "x", 42]
// Result: 42, x is now bound to 42

// Expression binding
["def", "doubled", ["*", "x", 2]]
// Result: 84, doubled is now bound to 84

// Function binding
["def", "square", ["lambda", ["n"], ["*", "n", "n"]]]
// Result: <function>, square is now bound to the function
```

Function Definition - `lambda`

Creates anonymous functions (closures) that capture their lexical environment.

```
["lambda", ["param1", "param2", ...], body_expression]
```

Evaluation Rules: 1. Do NOT evaluate parameters (they are binding names) 2. Do NOT evaluate body (it's evaluated when function is called) 3. Capture current environment as closure environment 4. Return function object

Examples:

```
// Simple function
["lambda", ["x"], ["*", "x", "x"]]
// Result: <function: x -> x * x>

// Multi-parameter function
["lambda", ["a", "b"], ["+", "a", "b"]]
// Result: <function: (a, b) -> a + b>

// Function with closure
["do",
  ["def", "multiplier", 10],
  ["lambda", ["x"], ["*", "x", "multiplier"]]]
// Result: <function: x -> x * 10> (captures multiplier=10)

// Higher-order function
["lambda", ["f", "x"], ["f", ["f", "x"]]]
// Result: <function: (f, x) -> f(f(x))>
```

Conditional Evaluation - `if`

Provides conditional branching with lazy evaluation of branches.

```
["if", condition_expression, then_expression, else_expression]
```

Evaluation Rules: 1. Evaluate `condition_expression` 2. If truthy, evaluate and return `then_expression` 3. If falsy, evaluate and return `else_expression` 4. Only one branch is evaluated (lazy evaluation)

Examples:

```
// Basic conditional
["if", [">", "x", 0], "positive", "non-positive"]

// Nested conditionals
["if", ["=", "status", "admin"],
  "full_access",
  ["if", ["=", "status", "user"], "limited_access", "no_access"]]

// With side effects (only one branch executes)
["if", "debug_mode",
  ["host", "log", "Debug information"],
  ["host", "log", "Production mode"]]
```

Local Bindings - `let`

Creates temporary, local variable bindings for use within a single expression. This is a cornerstone of functional programming as it avoids mutating the parent environment.

```
["let", [["var1", val1_expr], ["var2", val2_expr]], body_expr]
```

Evaluation Rules: 1. Create a new, temporary environment that extends the current one. 2. Evaluate all `val_expr` s in the original environment. 3. Bind the results to the `var` names in the new temporary environment. 4. Evaluate `body_expr` in the new environment. 5. Return the result of `body_expr` . The temporary environment is then discarded.

Example:

```
// Calculate area of a circle without polluting the environment
["let", [["pi", 3.14159], ["r", 10]],
  ["*", "pi", ["*", "r", "r"]]]
// Result: 314.159 (pi and r are not defined afterwards)
```

Error Handling - `try`

Provides a mechanism to catch and handle errors that occur during evaluation.

```
["try", body_expression, handler_expression]
```

Evaluation Rules: 1. Evaluate `body_expression` . 2. If evaluation succeeds, its result is the result of the `try` expression. The `handler_expression` is not evaluated. 3. If an error occurs during the evaluation of `body_expression` , the `handler_expression` is evaluated. The handler should be a function that accepts the error object as its argument. The result of the handler becomes the result of the entire `try` expression.

Example:

```
// Safe file reading with a fallback default
["try",
  ["host", "file/read", "/app/config.json"],
  ["lambda", ["err"],
    ["do",
      ["host", "log/warn", ["str-concat", "Config not found, using default: ", ["get", "err", "message"]]],
      { "default_setting": true }
    ]
  ]
]
```

Sequential Evaluation - `do`

Evaluates multiple expressions in sequence and returns the result of the final expression.

While JSL encourages a functional style using `let` and function composition, `do` is provided as a pragmatic tool for imperative-style sequencing, especially when dealing with multiple side-effects.

Design Note: Why is there no `for` or `while` loop?

JSL intentionally omits traditional `for` or `while` loop special forms. This is a core design decision to encourage a functional approach to collection processing. Instead of imperative looping, you should use the powerful higher-order functions provided in the prelude: - `map`: To transform each element in a list. - `filter`: To select elements from a list. - `reduce`: To aggregate a list into a single value. - `for_each`: To perform a side-effect for each element in a list.

These functions are safer, more declarative, and more composable than manual loops.

Quotation - `quote` and `@`

Prevents evaluation of expressions, returning them as literal data.

```
["quote", expression]
["@", expression]  // Shorthand syntax
```

There is also a syntactic sugar for quoting simple expressions:

```
["quote", "hello"]  // Result: "hello"
"@hello"// Result: "hello" (same as above)
```

It is only useful for simple values, as it does not allow for complex expressions, such as: `["@", ["+", 1, 2]]`.

Evaluation Rules: 1. Do NOT evaluate the argument 2. Return the argument as literal data 3. Preserves structure without interpretation

Examples:

```
// Quote simple values
["quote", "hello"]     // Result: "hello"
["@", "hello"]         // Result: "hello" (same as above)

// Quote expressions
["quote", ["+", 1, 2]]  // Result: ["+", 1, 2] (not 3)
["@", ["+", 1, 2]]      // Result: ["+", 1, 2]

// Quote for data structure creation
["@", {"name": "Alice", "age": 30}]
// Result: {"name": "Alice", "age": 30}

// Compare quoted vs unquoted
["def", "expr", ["@", ["+", "x", "y"]]]  // Store expression as data
["def", "result", ["+", "x", "y"]]       // Store computed result
```

Host Interaction - `host`

Provides controlled interaction with the host environment through JHIP.

```
["host", command_id_expression, arg1_expression, ...]
```

Evaluation Rules: 1. Evaluate `command_id_expression` to get command identifier 2. Evaluate all argument expressions 3. Send JHIP request to host with command and arguments 4. Return host response

Examples:

```
// File operations
["host", "file/read", "/path/to/file.txt"]
["host", "file/write", "/path/to/output.txt", "content"]

// System commands
["host", "system/exec", "ls", ["-la"]]

// Network requests
["host", "http/get", "https://api.example.com/data"]

// Time operations
["host", "time/now"]
["host", "time/format", "2023-12-01T10:30:00Z", "ISO"]
```

3.5.3 Special Form Properties

Argument Evaluation Control

Unlike regular functions, special forms control when and if their arguments are evaluated:

Special Form	Evaluation Pattern
`def`	Evaluate value, don't evaluate variable name
`lambda`	Don't evaluate parameters or body
`if`	Evaluate condition, then only one branch
`let`	Evaluate bindings, then body in new scope
`try`	Evaluate body, then handler only on error
`do`	Evaluate all arguments in sequence
`quote` / `@`	Don't evaluate argument at all
`host`	Evaluate all arguments

Environment Interaction

Special forms interact with the environment in specific ways:

```
// def modifies environment
["do",
  ["def", "x", 10],        // Adds x=10 to environment
  ["def", "y", ["*", "x", 2]], // Uses x from environment, adds y=20
  ["+", "x", "y"]]         // Uses both x and y: result 30

// lambda captures environment
["do",
  ["def", "base", 100],
  ["def", "adder", ["lambda", ["n"], ["+", "n", "base"]]],
  ["adder", 23]]           // Result: 123 (uses captured base=100)
```

3.5.4 Meta-Programming with Special Forms

Code Generation

```
// Generate conditional code
["def", "make_comparator",
  ["lambda", ["op"],
    ["@", ["lambda", ["a", "b"], ["op", "a", "b"]]]]]

// Usage
["def", "greater_than", ["make_comparator", ">"]]
["greater_than", 5, 3]  // Result: true
```

Dynamic Function Creation

```
// Create functions with varying parameter counts
["def", "make_n_ary_function",
  ["lambda", ["params", "body"],
    ["list", "@lambda", "params", "body"]]]

["def", "triple_add",
  ["make_n_ary_function",
    ["@", ["a", "b", "c"]],
    ["@", ["+", ["+", "a", "b"], "c"]]]]
```

Macro-like Patterns

```
// Define a "when" macro-like construct
["def", "when",
  ["lambda", ["condition", "action"],
    ["if", "condition", "action", null]]]
```

```
// Usage
["when", [">", "temperature", 30],
  ["host", "log", "@It's hot outside!"]]
```

3.5.5 Advanced Special Form Usage

Combining Special Forms

```
// Complex initialization pattern
["do",
  ["def", "config", ["host", "file/read", "config.json"]],
  ["def", "database",
    ["if", ["get", "config", "use_database"],
      ["host", "db/connect", ["get", "config", "db_url"]],
      null]],
  ["def", "processor",
    ["lambda", ["data"],
      ["if", "database",
        ["host", "db/store", "database", "data"],
        ["host", "log", "@No database configured"]]]],
  "processor"]
```

Error Handling Patterns

```
// Safe evaluation with fallback
["def", "safe_eval",
  ["lambda", ["expr", "fallback"],
    ["if", ["try", "expr"],
      "expr",
      "fallback"]]]

// Conditional resource acquisition
["def", "with_resource",
  ["lambda", ["resource_id", "action"],
    ["do",
      ["def", "resource", ["host", "resource/acquire", "resource_id"]],
      ["if", "resource",
        ["do",
          ["def", "result", ["action", "resource"]],
          ["host", "resource/release", "resource"],
          "result"],
        null]]]]
```

3.5.6 Special Forms vs Functions

Key Differences

Aspect	Special Forms	Functions
Argument Evaluation	Controlled by form	All arguments evaluated
Environment Access	Can modify environment	Read-only environment access
Evaluation Order	Form-specific rules	Standard left-to-right
Meta-Programming	Enable code generation	Operate on values only
Syntax Extension	Can create new syntax	Cannot extend syntax

When to Use Each

Use Special Forms for: - Control flow (`if` , `do`) - Variable binding (`def` , `let`) - Function creation (`lambda`) - Meta-programming (`quote`) - Host interaction (`host`) - Error handling (`try`)

Use Functions for: - Data transformation - Mathematical operations - String processing - Collection manipulation - Business logic - JSON object construction (using JSL's first-class object support)

3.5.7 Implementation Notes

Evaluation Context

Special forms are evaluated in the context of JSL's evaluator, which maintains:

1. **Environment Stack**: For variable resolution and binding
2. **Continuation Stack**: For function calls and returns
3. **Host Interface**: For JHIP command execution
4. **Error Handling**: For exception propagation

Performance Considerations

- `def` : O(1) environment binding
- `let` : O(1) local binding
- `try` : O(n) body evaluation, O(m) handler evaluation on error
- `lambda` : O(1) closure creation, O(e) environment capture
- `if` : O(1) branch selection, avoids evaluating unused branch
- `do` : O(n) sequential evaluation
- `quote` : O(1) literal return
- `host` : O(h) depends on host operation complexity

Special forms are the foundation of JSL's expressiveness, providing the essential building blocks for all higher-level language constructs while maintaining the language's homoiconic JSON-based structure.

3.5.8 Special Form Details

`host`

The `host` special form is the gateway for all side effects and interactions with the host system.

- **Syntax**: `["host", command_expr, arg1_expr, ...]`
- **Evaluation**: All arguments are evaluated. The results are packaged into a JHIP request and yielded to the host runtime.

3.5.9 JSON Objects as First-Class Citizens

JSL provides powerful support for JSON object construction through its first-class object syntax. Objects in JSL:

- Are always treated as data structures (never function calls)
- Use normal JSL evaluation rules for both keys and values
- Require keys to evaluate to strings
- Support dynamic construction with variables and expressions

For example:

```
["do",
  ["def", "name", "@Alice"],
  ["def", "age", 30],
  {"@greeting": ["str-concat", "@Hello ", "name"],
   "@info": {"@name": "name", "@age": "age"}}
]
```

For more details, see **JSON Objects as First-Class Citizens**.

3.6 JSON Objects as First-Class Citizens

3.6.1 Overview

JSL treats JSON objects as **first-class data structures** with native support for dynamic construction. Unlike arrays (which are interpreted as S-expressions), JSON objects in JSL are always treated as data structures, making them perfect for constructing pure JSON output without ambiguity.

3.6.2 Why Objects Are Special

In JSL, there's an important distinction between arrays and objects:

- **Arrays**: `["+", 1, 2]` are interpreted as function calls (S-expressions)
- **Objects**: `{"name": "Alice"}` are always treated as data structures

This means objects provide a "safe zone" for pure data construction where you don't have to worry about the first element being interpreted as an operator.

3.6.3 Object Construction Syntax

JSL objects use **normal JSL evaluation rules** for both keys and values:

- **Keys** must evaluate to strings
- **Values** can be any JSL expression
- Use `@` prefix for literal strings in both keys and values

Basic Examples

Literal Object:

```
{"@name": "@Alice", "@age": 25}
```

Result: `{"name": "Alice", "age": 25}`

Dynamic Values:

```
["do",
  ["def", "user_name", "@Bob"],
  ["def", "user_age", 30],
  {"@name": "user_name", "@age": "user_age"}
]
```

Result: `{"name": "Bob", "age": 30}`

Dynamic Keys:

```
["do",
  ["def", "field_name", "@username"],
  {"field_name": "@Alice"}
]
```

Result: `{"username": "Alice"}`

3.6.4 String Construction in Objects

For dynamic string construction, use JSL's string functions:

```
["do",
  ["def", "name", "@Alice"],
  ["def", "age", 25],
  {
    "@greeting": ["str-concat", "@Hello ", "name"],
```

```
    "@info": ["str-concat", "@Age: ", "age"],
    "@status": ["if", [">", "age", 18], "@adult", "@minor"]
  }
]
```

Result:

```
{
  "greeting": "Hello Alice",
  "info": "Age: 25",
  "status": "adult"
}
```

3.6.5 Nested Objects and Complex Structures

Objects can contain any JSL expressions, including nested objects and arrays:

```
["do",
  ["def", "users", ["@", ["Alice", "Bob", "Carol"]]],
  {
    "@project": "@My Project",
    "@team": {
      "@lead": ["first", "users"],
      "@members": "users",
      "@size": ["length", "users"]
    },
    "@tags": ["@", ["web", "javascript", "api"]]
  }
]
```

3.6.6 Advantages of Object-First Design

1. **No Operator Ambiguity**: Objects are always data, never function calls
2. **Pure JSON Output**: Objects naturally serialize to clean JSON
3. **Composable**: Works seamlessly with JSL functions and variables
4. **Consistent Syntax**: Uses the same `@` rules as the rest of JSL
5. **Type Safety**: Keys are validated to be strings at runtime

3.6.7 Working with Object Functions

JSL provides built-in functions for object manipulation:

```
["do",
  ["def", "user", {"@name": "@Alice", "@role": "@admin"}],

  ["get", "user", "@name"],        // → "Alice"
  ["has", "user", "@email"],       // → false
  ["keys", "user"],                // → ["name", "role"]

  ["set", "user", "@email", "@alice@example.com"]  // Add new field
]
```

3.7 Templates

3.8 Prelude Functions

3.8.1 Overview

The JSL prelude provides the computational foundation for all JSL programs. These built-in functions are available in every JSL environment.

For a guide to creating and managing execution contexts, see the Environments documentation.

3.8.2 Special Forms (Core Syntax)

While the prelude contains a library of standard functions, the core language is defined by a small set of **special forms**. These are syntactic constructs that do not follow the standard evaluation rule (i.e., they don't necessarily evaluate all of their arguments).

The core special forms include `if`, `def`, `lambda`, `do`, `let`, and `try`. For a complete reference, please see the dedicated **Special Forms** documentation.

3.8.3 Design Principles

- **Immutable Operations**: Functions return new values rather than modifying inputs
- **N-arity Support**: Mathematical and logical operations accept variable numbers of arguments
- **Type Safety**: Comprehensive type predicates and safe conversions
- **Functional Composition**: Higher-order functions that work seamlessly with JSL closures
- **JSON Compatibility**: All operations respect JSON's type system

3.8.4 Data Constructors

`list`

```
["list", 1, 2, 3]  // → [1, 2, 3]
["list"]           // → []
```

Creates a list from the provided arguments.

3.8.5 List Operations

JSL provides comprehensive list manipulation functions following functional programming principles.

`append`

```
["append", [1, 2, 3], 4]  // → [1, 2, 3, 4]
["append", [], 1]         // → [1]
```

Returns a new list with the item appended to the end.

`prepend`

```
["prepend", 0, [1, 2, 3]]  // → [0, 1, 2, 3]
["prepend", 1, []]         // → [1]
```

Returns a new list with the item prepended to the beginning.

`concat`

```
["concat", [1, 2], [3, 4], [5]]  // → [1, 2, 3, 4, 5]
["concat", [1], [2]]             // → [1, 2]
["concat"]                       // → []
```

Concatenates multiple lists into a single list.

`first`

```
["first", [1, 2, 3]]  // → 1
["first", []]         // → null
```

Returns the first element of a list, or null if empty.

`rest`

```
["rest", [1, 2, 3]]  // → [2, 3]
["rest", [1]]        // → []
["rest", []]         // → []
```

Returns all elements except the first, or empty list if insufficient elements.

`nth`

```
["nth", [10, 20, 30], 1]  // → 20
["nth", [10, 20], 5]      // → null
```

Returns the element at the specified index (0-based), or null if out of bounds.

`length`

```
["length", [1, 2, 3]]  // → 3
["length", []]         // → 0
["length", "hello"]    // → 5
```

Returns the length of a list or string.

`empty?`

```
["empty?", []]      // → true
["empty?", [1]]     // → false
["empty?", ""]      // → true
["empty?", "hi"]    // → false
```

Returns true if the collection is empty.

`slice`

```
["slice", [1, 2, 3, 4, 5], 1, 4]  // → [2, 3, 4]
["slice", [1, 2, 3], 1]           // → [2, 3]
["slice", "hello", 1, 4]          // → "ell"
```

Returns a slice of the list or string from start to end (exclusive).

`reverse`

```
["reverse", [1, 2, 3]]  // → [3, 2, 1]
["reverse", "hello"]    // → "olleh"
```

Returns a reversed copy of the list or string.

`contains?`

```
["contains?", [1, 2, 3], 2]      // → true
["contains?", [1, 2, 3], 4]      // → false
["contains?", "hello", "ell"]    // → true
```

Returns true if the collection contains the specified item.

index

```
["index", [10, 20, 30], 20]  // → 1
["index", [10, 20, 30], 40]  // → -1
```

Returns the index of the first occurrence of item, or -1 if not found.

3.8.6 Dictionary Operations

Immutable dictionary operations supporting functional programming patterns.

get

```
["get", {"name": "Alice", "age": 30}, "name"]          // → "Alice"
["get", {"name": "Alice"}, "age", "unknown"]           // → "unknown"
```

Gets a value from a dictionary with optional default.

set

```
["set", {"name": "Alice"}, "age", 30]  // → {"name": "Alice", "age": 30}
["set", {}, "key", "value"]            // → {"key": "value"}
```

Returns a new dictionary with the key-value pair set.

keys

```
["keys", {"name": "Alice", "age": 30}]  // → ["name", "age"]
["keys", {}]                            // → []
```

Returns a list of all keys in the dictionary.

values

```
["values", {"name": "Alice", "age": 30}]  // → ["Alice", 30]
["values", {}]                            // → []
```

Returns a list of all values in the dictionary.

merge

```
["merge", {"a": 1}, {"b": 2}, {"c": 3}]  // → {"a": 1, "b": 2, "c": 3}
["merge", {"a": 1}, {"a": 2}]            // → {"a": 2}
```

Merges multiple dictionaries, with later values overriding earlier ones.

has-key?

```
["has-key?", {"name": "Alice"}, "name"]  // → true
["has-key?", {"name": "Alice"}, "age"]   // → false
```

Returns true if the dictionary contains the specified key.

3.8.7 Arithmetic Operations

Mathematical operations with n-arity support for natural expression.

`+` (Addition)

```
["+", 1, 2, 3]    // → 6
["+", 5]          // → 5
["+"]             // → 0
```

Adds all arguments. With no arguments, returns 0.

`-` (Subtraction)

```
["-", 10, 3, 2]   // → 5 (10 - 3 - 2)
["-", 5]          // → -5 (negation)
["-"]             // → 0
```

Subtracts subsequent arguments from the first. With one argument, returns negation.

`*` (Multiplication)

```
["*", 2, 3, 4]    // → 24
["*", 5]          // → 5
["*"]             // → 1
```

Multiplies all arguments. With no arguments, returns 1.

`/` (Division)

```
["/", 12, 3, 2]   // → 2.0 (12 / 3 / 2)
["/", 5]          // → 0.2 (1 / 5)
```

Divides the first argument by all subsequent arguments. With one argument, returns reciprocal.

`mod` (Modulo)

```
["mod", 10, 3]    // → 1
["mod", 7, 0]     // → 0 (safe: returns 0 for division by zero)
```

Returns the remainder of division.

`pow` (Exponentiation)

```
["pow", 2, 3]     // → 8
["pow", 9, 0.5]   // → 3.0
```

Raises the first argument to the power of the second.

3.8.8 Comparison Operations

Chained comparisons supporting mathematical notation.

`=` (Equality)

```
["=", 1, 1, 1]       // → true
["=", 1, 2]          // → false
["=", "a", "a", "a"] // → true
```

Returns true if all arguments are equal.

`<` (Less Than)

```
["<", 1, 2, 3]    // → true (1 < 2 < 3)
["<", 1, 3, 2]    // → false
```

Returns true if arguments form an ascending sequence.

`>` (Greater Than)

```
[">", 3, 2, 1]    // → true (3 > 2 > 1)
[">", 3, 1, 2]    // → false
```

Returns true if arguments form a descending sequence.

`<=` (Less Than or Equal)

```
["<=", 1, 2, 2, 3]  // → true
["<=", 1, 3, 2]     // → false
```

Returns true if arguments form a non-decreasing sequence.

`>=` (Greater Than or Equal)

```
[">=", 3, 2, 2, 1]  // → true
[">=", 3, 1, 2]     // → false
```

Returns true if arguments form a non-increasing sequence.

3.8.9 Logical Operations

Logical operations with n-arity support and short-circuiting.

`and`

```
["and", true, true, true]   // → true
["and", true, false, true]  // → false
["and"]                     // → true
```

Returns true if all arguments are truthy.

`or`

```
["or", false, false, true]  // → true
["or", false, false]        // → false
["or"]                      // → false
```

Returns true if any argument is truthy.

`not`

```
["not", true]    // → false
["not", false]   // → true
["not", 0]       // → true
["not", ""]      // → true
```

Returns the logical negation of the argument.

3.8.10 Type Predicates

Essential for wire format validation and dynamic type checking.

`null?`

```
["null?", null]    // → true
["null?", 0]       // → false
["null?", false]   // → false
```

Returns true if the value is null.

`bool?`

```
["bool?", true]    // → true
["bool?", false]   // → true
["bool?", 0]       // → false
```

Returns true if the value is a boolean.

`number?`

```
["number?", 42]      // → true
["number?", 3.14]    // → true
["number?", "42"]    // → false
```

Returns true if the value is a number (integer or float).

`string?`

```
["string?", "hello"]  // → true
["string?", 42]       // → false
```

Returns true if the value is a string.

`list?`

```
["list?", [1, 2, 3]]  // → true
["list?", "hello"]    // → false
```

Returns true if the value is a list.

`dict?`

```
["dict?", {"a": 1}]   // → true
["dict?", [1, 2]]     // → false
```

Returns true if the value is a dictionary.

`callable?`

```
["callable?", ["lambda", ["x"], "x"]]  // → true (after evaluation)
["callable?", 42]                      // → false
```

Returns true if the value is callable (function or closure).

3.8.11 String Operations

String manipulation functions for text processing.

`str-concat`

```
["str-concat", "Hello", " ", "World"]  // → "Hello World"
["str-concat", "Number: ", 42]         // → "Number: 42"
```

Concatenates all arguments after converting them to strings.

`str-split`

```
["str-split", "a,b,c", ","]     // → ["a", "b", "c"]
["str-split", "hello world"]    // → ["hello", "world"] (default: space)
```

Splits a string by the specified separator.

str-join

```
["str-join", ["a", "b", "c"], ","]    // → "a,b,c"
["str-join", [1, 2, 3], "-"]          // → "1-2-3"
["str-join", ["a", "b"]]              // → "ab" (default: empty string)
```

Joins a list of values into a string with the specified separator.

str-length

```
["str-length", "hello"]    // → 5
["str-length", ""]         // → 0
```

Returns the length of a string.

str-upper

```
["str-upper", "hello"]     // → "HELLO"
```

Converts a string to uppercase.

str-lower

```
["str-lower", "HELLO"]     // → "hello"
```

Converts a string to lowercase.

3.8.12 Higher-Order Functions

The cornerstone of functional programming, enabling composition and abstraction.

map

```
["map", ["lambda", ["x"], ["*", "x", 2]], [1, 2, 3]]  // → [2, 4, 6]
["map", "+", [[1, 2], [3, 4]]]                        // → [3, 7]
```

Applies a function to each element of a list, returning a new list of results.

filter

```
["filter", ["lambda", ["x"], [">", "x", 5]], [1, 6, 3, 8, 2]]  // → [6, 8]
["filter", "even?", [1, 2, 3, 4, 5, 6]]                        // → [2, 4, 6]
```

Returns a new list containing only elements for which the predicate returns true.

reduce

```
["reduce", "+", [1, 2, 3, 4]]          // → 10
["reduce", "*", [1, 2, 3, 4], 1]       // → 24 (with initial value)
["reduce", "max", [3, 1, 4, 1, 5]]     // → 5
```

Reduces a list to a single value by repeatedly applying a binary function.

apply

```
["apply", "+", [1, 2, 3]]                  // → 6
["apply", ["lambda", ["x", "y"], ["*", "x", "y"]], [3, 4]]  // → 12
```

Applies a function to a list of arguments.

3.8.13 Mathematical Functions

Extended mathematical operations for scientific computing.

`min` / `max`

```
["min", 3, 1, 4, 1, 5]  // → 1
["max", 3, 1, 4, 1, 5]  // → 5
```

Returns the minimum or maximum of the arguments.

`abs`

```
["abs", -5]    // → 5
["abs", 3.14]  // → 3.14
```

Returns the absolute value.

`round`

```
["round", 3.7]     // → 4
["round", 3.14159, 2]  // → 3.14
```

Rounds to the nearest integer or specified decimal places.

Trigonometric Functions

```
["sin", 1.5708]    // → ~1.0 (π/2)
["cos", 0]         // → 1.0
["tan", 0.7854]    // → ~1.0 (π/4)
```

Standard trigonometric functions (arguments in radians).

`sqrt`

```
["sqrt", 16]   // → 4.0
["sqrt", 2]    // → ~1.414
```

Returns the square root.

`log` / `exp`

```
["log", 2.718]   // → ~1.0 (natural log)
["exp", 1]       // → ~2.718 (e^1)
```

Natural logarithm and exponential functions.

3.8.14 Type Conversion

Safe type conversion functions with reasonable defaults.

`to-string`

```
["to-string", 42]     // → "42"
["to-string", true]   // → "True"
["to-string", [1,2]]  // → "[1, 2]"
```

Converts any value to its string representation.

`to-number`

```
["to-number", "42"]      // → 42.0
["to-number", "3.14"]    // → 3.14
["to-number", "hello"]   // → 0 (safe default)
```

Attempts to convert a value to a number, returning 0 for invalid inputs.

`type-of`

```
["type-of", 42]        // → "int"
["type-of", "hello"]   // → "str"
["type-of", [1, 2]]    // → "list"
```

Returns the type name of a value.

3.8.15 I/O Operations

Basic I/O functions (can be customized in sandboxed environments).

`print`

```
["print", "Hello, World!"]  // Outputs: Hello, World!
["print", 42, "is the answer"]  // Outputs: 42 is the answer
```

Prints values to standard output.

`error`

```
["error", "Something went wrong!"]  // Raises RuntimeError
```

Raises a runtime error with the specified message.

3.8.16 Integration with JSL Closures

All higher-order functions in the prelude work seamlessly with JSL closures through the `eval_closure_or_builtin` integration layer. This ensures that:

1. **Lexical scoping is preserved** - Closures maintain access to their captured environments
2. **Built-in access is guaranteed** - All closures can access prelude functions
3. **Performance is optimized** - Environment chains are linked efficiently at call time
4. **Serialization is safe** - Only user bindings are serialized with closures

This design enables powerful functional programming patterns while maintaining JSL's core promise of safe, network-transmissible code.

4. Tutorials

4.1 Your First JSL Program

Welcome to JSL! This tutorial will walk you through creating your first JSL program step by step. By the end, you'll understand JSL's core concepts and be ready to build more complex applications.

4.1.1 What Makes JSL Different

JSL is unlike most programming languages you may have used before. The key insight is that **code and data are the same thing** - both are represented as JSON. This means:

- Your program is valid JSON that can be transmitted over networks
- Functions can be serialized and reconstructed anywhere
- No compilation step - JSON is the native format
- Universal compatibility - any system that handles JSON can run JSL

4.1.2 Hello, World!

Let's start with the classic first program. Create a file called `hello.jsl`:

```
["print", "@Hello, JSL!"]
```

Now, run it from your terminal:

```
jsl hello.jsl
```

You should see the output:

```
Hello, JSL!
```

What happened?

1. `["print", "Hello, JSL!"]` is a **function call**
2. `"print"` is the function name (a built-in function)
3. `"Hello, JSL!"` is the argument
4. JSL evaluates the expression and calls the print function

4.1.3 Understanding Prefix Notation

JSL uses **prefix notation** - the operator comes first:

```
// Traditional: 2 + 3
["+", 2, 3]

// Traditional: 2 + 3 + 4
["+", 2, 3, 4]

// Traditional: 2 * (3 + 4)
["*", 2, ["+", 3, 4]]
```

Try it:

```
["print", "2 + 3 =", ["+", 2, 3]]
```

4.1.4 Variables with `def`

Use `def` to create variables:

```
[
  "do",
  ["def", "name", "Alice"],
  ["def", "age", 30],
  ["print", "Hello,", "name", "! You are", "age", "years old."]
]
```

Breaking it down:

1. `"do"` executes multiple expressions in sequence
2. `["def", "name", "Alice"]` creates a variable called `name`
3. `["def", "age", 30]` creates a variable called `age`
4. The print statement uses the variables by referencing their names

4.1.5 Your First Function

Let's create a function to calculate the area of a circle:

```
[
  "do",
  ["def", "pi", 3.14159],
  ["def", "circle-area",
   ["lambda", ["radius"],
    ["*", "pi", "radius", "radius"]]],
  ["def", "my-radius", 5],
  ["def", "area", ["circle-area", "my-radius"]],
  ["print", "Circle with radius", "my-radius", "has area", "area"]
]
```

Understanding `lambda`:

- `["lambda", ["radius"], ...]` creates a function
- `["radius"]` is the parameter list (the function takes one argument)
- `["*", "pi", "radius", "radius"]` is the function body
- The function calculates π × radius²

4.1.6 Working with Lists

Lists are fundamental in JSL. Let's explore list operations:

```
[
  "do",
  ["def", "numbers", ["list", 1, 2, 3, 4, 5]],
  ["def", "first-number", ["first", "numbers"]],
  ["def", "rest-numbers", ["rest", "numbers"]],
  ["def", "list-length", ["length", "numbers"]],

  ["print", "Original list:", "numbers"],
  ["print", "First element:", "first-number"],
  ["print", "Rest of list:", "rest-numbers"],
  ["print", "List length:", "list-length"]
]
```

4.1.7 Higher-Order Functions

Now for something powerful - functions that work with other functions:

```
[
  "do",
  ["def", "numbers", ["list", 1, 2, 3, 4, 5]],

  // Double each number
  ["def", "double", ["lambda", ["x"], ["*", "x", 2]]],
  ["def", "doubled", ["map", "double", "numbers"]],

  // Filter even numbers
```

```
  ["def", "is-even", ["lambda", ["n"], ["=", ["mod", "n", 2], 0]]],
  ["def", "evens", ["filter", "is-even", "numbers"]],

  // Sum all numbers
  ["def", "total", ["reduce", "+", "numbers", 0]],

  ["print", "Original:", "numbers"],
  ["print", "Doubled:", "doubled"],
  ["print", "Evens only:", "evens"],
  ["print", "Sum:", "total"]
]
```

Key concepts:

- `map` applies a function to each element of a list
- `filter` keeps only elements that match a condition
- `reduce` combines all elements into a single value

4.1.8 Conditional Logic

Use `if` for decisions:

```
[
  "do",
  ["def", "temperature", 75],
  ["def", "weather",
   ["if", [">", "temperature", 80],
    "hot",
    ["if", [">", "temperature", 60],
     "warm",
     "cool"]]],
  ["print", "It's", "weather", "today at", "temperature", "degrees"]
]
```

4.1.9 Data Structures

Work with dictionaries (objects) to structure data:

```
[
  "do",
  ["def", "person", {
    "name": "Bob",
    "age": 25,
    "city": "San Francisco"
  }],

  ["def", "name", ["get", "person", "name"]],
  ["def", "age", ["get", "person", "age"]],

  // Create a new person with updated age
  ["def", "older-person", ["set", "person", "age", ["+", "age", 1]]],

  ["print", "Original person:", "person"],
  ["print", "Person next year:", "older-person"]
]
```

4.1.10 Putting It Together: A Complete Example

Let's build a program that processes a list of people:

```
[
  "do",

  // Sample data
  ["def", "people", [
    {"name": "Alice", "age": 30, "city": "NYC"},
    {"name": "Bob", "age": 25, "city": "SF"},
    {"name": "Charlie", "age": 35, "city": "NYC"},
    {"name": "Diana", "age": 28, "city": "LA"}
  ]],

  // Helper functions
  ["def", "get-name", ["lambda", ["person"], ["get", "person", "name"]]],
  ["def", "get-age", ["lambda", ["person"], ["get", "person", "age"]]],
  ["def", "is-adult", ["lambda", ["person"], [">=", ["get-age", "person"], 18]]],
  ["def", "lives-in-nyc", ["lambda", ["person"], ["=", ["get", "person", "city"], "NYC"]]],

  // Process the data
```

```
  ["def", "adults", ["filter", "is-adult", "people"]],
  ["def", "nyc-adults", ["filter", "lives-in-nyc", "adults"]],
  ["def", "nyc-names", ["map", "get-name", "nyc-adults"]],
  ["def", "average-age",
   ["/", ["reduce", "+", ["map", "get-age", "adults"], 0], ["length", "adults"]]],

  // Output results
  ["print", "All people:", ["map", "get-name", "people"]],
  ["print", "Adults in NYC:", "nyc-names"],
  ["print", "Average age of adults:", "average-age"]
]
```

This program demonstrates:

- Working with structured data (lists and dictionaries)
- Creating helper functions for common operations
- Chaining operations together (filter, then map)
- Computing aggregates (average age)

4.1.11 What You've Learned

Congratulations! You now understand:

1. **Prefix notation** - operators come first
2. **Variables** - using `def` to bind values to names
3. **Functions** - creating them with `lambda`
4. **Lists and dictionaries** - fundamental data structures
5. **Higher-order functions** - `map`, `filter`, `reduce`
6. **Conditional logic** - making decisions with `if`
7. **JSON structure** - how code and data are the same

4.1.12 Next Steps

Ready to learn more? Try these tutorials:

- **Working with Functions** - Advanced function concepts
- **Data Manipulation** - Complex data processing patterns
- **Code Serialization** - Sending code over networks
- **Distributed Computing** - Building distributed applications

4.1.13 Practice Exercises

Try building these programs yourself:

1. **FizzBuzz**: Print numbers 1-100, but "Fizz" for multiples of 3, "Buzz" for multiples of 5, and "FizzBuzz" for multiples of both.
2. **Word Counter**: Given a list of words, count how many times each word appears.
3. **Temperature Converter**: Create functions to convert between Celsius and Fahrenheit.
4. **Shopping Cart**: Calculate the total price of items in a shopping cart, including tax.

Ready to tackle these? You have all the tools you need!

4.2 Learning Functions in JSL

4.2.1 Introduction

Functions are the building blocks of JSL programs. In this tutorial, you'll learn to create, use, and combine functions through hands-on examples.

4.2.2 Your First Function

Let's start with the simplest possible function:

```
["lambda", ["x"], "x"]
```

This is the **identity function** - it returns whatever you give it. Try it:

```
[["lambda", ["x"], "x"], "hello"]
// Result: "hello"
```

4.2.3 Naming Functions

Usually, you'll want to give functions names:

```
["def", "identity", ["lambda", ["x"], "x"]]
```

Now you can use it by name:

```
["identity", "hello"]
// Result: "hello"
```

4.2.4 Functions with Multiple Parameters

```
["def", "add", ["lambda", ["a", "b"], ["+", "a", "b"]]]
["add", 3, 7]
// Result: 10
```

4.2.5 Step-by-Step: Building a Math Library

Step 1: Basic Operations

```
["do",
  ["def", "square", ["lambda", ["x"], ["*", "x", "x"]]],
  ["def", "double", ["lambda", ["x"], ["*", "x", 2]]],
  ["def", "half", ["lambda", ["x"], ["/", "x", 2]]]]
```

Step 2: Test Your Functions

```
["square", 5]    // Result: 25
["double", 5]    // Result: 10
["half", 10]     // Result: 5
```

Step 3: Combining Functions

```
["def", "square_and_double",
  ["lambda", ["x"], ["double", ["square", "x"]]]]

["square_and_double", 3]
// 3 → square → 9 → double → 18
```

4.2.6 Higher-Order Functions

Functions that work with other functions:

Step 1: A Function That Applies Another Function Twice

```
["def", "twice",
  ["lambda", ["f", "x"], ["f", ["f", "x"]]]]
```

Step 2: Use It

```
["twice", "double", 5]
// 5 → double → 10 → double → 20
```

4.2.7 Working with Lists

Step 1: Processing Each Item

```
["def", "numbers", [1, 2, 3, 4, 5]]
["map", "square", "numbers"]
// Result: [1, 4, 9, 16, 25]
```

Step 2: Filtering Lists

```
["def", "is_even", ["lambda", ["x"], ["=", ["mod", "x", 2], 0]]]
["filter", "is_even", "numbers"]
// Result: [2, 4]
```

Step 3: Combining Operations

```
["def", "sum_of_squares_of_evens",
  ["lambda", ["numbers"],
    ["sum", ["map", "square", ["filter", "is_even", "numbers"]]]]]

["sum_of_squares_of_evens", [1, 2, 3, 4, 5]]
// [1,2,3,4,5] → filter evens → [2,4] → square → [4,16] → sum → 20
```

4.2.8 Closures: Functions That Remember

```
["def", "make_adder",
  ["lambda", ["n"],
    ["lambda", ["x"], ["+", "x", "n"]]]]

["def", "add_10", ["make_adder", 10]]
["add_10", 5]
// Result: 15
```

The inner function "remembers" the value of `n` (10) even after `make_adder` finishes.

4.2.9 Practice Exercises

Exercise 1: Temperature Converter

Create functions to convert between Celsius and Fahrenheit:

```
// Your solution here
["def", "celsius_to_fahrenheit", ["lambda", ["c"], ...]]
["def", "fahrenheit_to_celsius", ["lambda", ["f"], ...]]
```

ution

```
["do",
  ["def", "celsius_to_fahrenheit",
    ["lambda", ["c"], ["+", ["*", "c", 9/5], 32]]],
  ["def", "fahrenheit_to_celsius",
    ["lambda", ["f"], ["*", ["-", "f", 32], 5/9]]]]
```

Exercise 2: List Statistics

Create a function that returns statistics about a list of numbers:

```
// Should return: {"min": 1, "max": 5, "avg": 3, "count": 5}
["stats", [1, 2, 3, 4, 5]]
```

ution

```
["def", "stats",
  ["lambda", ["numbers"],
    {
      "min": ["min", "numbers"],
      "max": ["max", "numbers"],
      "avg": ["/", ["sum", "numbers"], ["length", "numbers"]],
      "count": ["length", "numbers"]
    }]]
```

4.2.10 Next Steps

- Learn about working with data
- Explore JSON Objects as first-class citizens
- Try distributed computing

Functions in JSL are powerful and flexible. With closures and higher-order functions, you can build complex programs from simple, composable pieces.

4.3 Working with Data in JSL

4.3.1 Introduction

JSL uses JSON for all data structures, making it easy to work with familiar formats. This tutorial teaches you to manipulate data step by step.

4.3.2 Basic Data Types

JSL supports all JSON data types:

```
["do",
  ["def", "number", 42],
  ["def", "string", "hello"],
  ["def", "boolean", true],
  ["def", "null_value", null],
  ["def", "array", [1, 2, 3]],
  ["def", "object", {"name": "Alice", "age": 30}]]
```

4.3.3 Working with Objects

Accessing Object Properties

```
["def", "person", {"name": "Alice", "age": 30, "city": "New York"}]

["get", "person", "name"]     // Result: "Alice"
["get", "person", "age"]      // Result: 30
```

Modifying Objects

```
["def", "updated_person",
  ["assoc", "person", "age", 31]]
// Result: {"name": "Alice", "age": 31, "city": "New York"}
```

Adding New Properties

```
["def", "person_with_email",
  ["assoc", "person", "email", "alice@example.com"]]
```

4.3.4 Working with Arrays

Basic Array Operations

```
["def", "numbers", [1, 2, 3, 4, 5]]

["length", "numbers"]             // Result: 5
["first", "numbers"]              // Result: 1
["last", "numbers"]               // Result: 5
["nth", "numbers", 2]             // Result: 3 (0-indexed)
```

Adding Elements

```
["conj", "numbers", 6]            // Result: [1, 2, 3, 4, 5, 6]
["concat", "numbers", [6, 7, 8]]  // Result: [1, 2, 3, 4, 5, 6, 7, 8]
```

4.3.5 Step-by-Step: Building a Contact Manager

Step 1: Create Contact Data

```
["def", "contacts", [
  {"name": "Alice", "email": "alice@example.com", "phone": "555-0101"},
  {"name": "Bob", "email": "bob@example.com", "phone": "555-0102"},
```

```
  {"name": "Charlie", "email": "charlie@example.com", "phone": "555-0103"}
]]
```

Step 2: Find a Contact by Name

```
["def", "find_contact",
  ["lambda", ["name"],
    ["first",
      ["filter",
        ["lambda", ["contact"], ["=", ["get", "contact", "name"], "name"]],
        "contacts"]]]]

["find_contact", "Alice"]
// Result: {"name": "Alice", "email": "alice@example.com", "phone": "555-0101"}
```

Step 3: Get All Email Addresses

```
["def", "get_all_emails",
  ["lambda", [],
    ["map", ["lambda", ["contact"], ["get", "contact", "email"]], "contacts"]]]

["get_all_emails"]
// Result: ["alice@example.com", "bob@example.com", "charlie@example.com"]
```

Step 4: Add a New Contact

```
["def", "add_contact",
  ["lambda", ["name", "email", "phone"],
    ["conj", "contacts", {"name": "name", "email": "email", "phone": "phone"}]]]

["def", "updated_contacts",
  ["add_contact", "Diana", "diana@example.com", "555-0104"]]
```

4.3.6 Data Transformation Patterns

Filtering Data

```
// Find contacts with Gmail addresses
["def", "gmail_contacts",
  ["filter",
    ["lambda", ["contact"],
      ["includes?", ["get", "contact", "email"], "gmail.com"]],
    "contacts"]]
```

Grouping Data

```
// Group contacts by email domain
["def", "group_by_domain",
  ["group_by",
    ["lambda", ["contact"],
      ["last", ["split", ["get", "contact", "email"], "@"]]],
    "contacts"]]
```

Sorting Data

```
// Sort contacts by name
["def", "sorted_contacts",
  ["sort_by", ["lambda", ["contact"], ["get", "contact", "name"]], "contacts"]]
```

4.3.7 Working with Nested Data

Step 1: Complex Data Structure

```
["def", "company", {
  "name": "Tech Corp",
  "departments": [
    {
      "name": "Engineering",
      "employees": [
        {"name": "Alice", "salary": 100000},
        {"name": "Bob", "salary": 95000}
```

```
      ]
    },
    {
      "name": "Sales",
      "employees": [
        {"name": "Charlie", "salary": 80000},
        {"name": "Diana", "salary": 85000}
      ]
    }
  ]
}]
```

Step 2: Extract All Employee Names

```
["def", "all_employee_names",
  ["flatten",
    ["map",
      ["lambda", ["dept"],
        ["map",
          ["lambda", ["emp"], ["get", "emp", "name"]],
          ["get", "dept", "employees"]]],
      ["get", "company", "departments"]]]]

["all_employee_names"]
// Result: ["Alice", "Bob", "Charlie", "Diana"]
```

Step 3: Calculate Total Payroll

```
["def", "total_payroll",
  ["sum",
    ["flatten",
      ["map",
        ["lambda", ["dept"],
          ["map",
            ["lambda", ["emp"], ["get", "emp", "salary"]],
            ["get", "dept", "employees"]]],
        ["get", "company", "departments"]]]]]

["total_payroll"]
// Result: 360000
```

4.3.8 Data Validation

Step 1: Validation Functions

```
["def", "valid_email?",
  ["lambda", ["email"],
    ["and",
      ["includes?", "email", "@"],
      [">", ["length", "email"], 5]]]]

["def", "valid_phone?",
  ["lambda", ["phone"],
    ["=", ["length", "phone"], 12]]]  // Assuming XXX-XXXX format
```

Step 2: Validate Contact

```
["def", "valid_contact?",
  ["lambda", ["contact"],
    ["and",
      ["valid_email?", ["get", "contact", "email"]],
      ["valid_phone?", ["get", "contact", "phone"]],
      [">", ["length", ["get", "contact", "name"]], 0]]]]
```

Step 3: Filter Valid Contacts

```
["def", "valid_contacts",
  ["filter", "valid_contact?", "contacts"]]
```

4.3.9 Dynamic Object Construction with Data

Step 1: Email Object Structure

```
["def", "create_email",
  ["lambda", ["recipient_email", "recipient_name"],
    {
      "@to": "recipient_email",
      "@subject": ["str-concat", "@Welcome ", "recipient_name", "@!"],
      "@body": ["str-concat", "@Hello ", "recipient_name", "@, welcome to our service!"]
    }
  ]
]
```

Step 2: Generate Emails for All Contacts

```
["def", "generate_welcome_emails",
  ["map",
    ["lambda", ["contact"],
      ["create_email",
        ["get", "contact", "@email"],
        ["get", "contact", "@name"]
      ]],
    "contacts"]]
```

4.3.10 Practice Exercises

Exercise 1: Inventory Management

Create functions to manage a product inventory:

```
["def", "inventory", [
  {"id": 1, "name": "Laptop", "price": 999, "stock": 5},
  {"id": 2, "name": "Mouse", "price": 25, "stock": 50},
  {"id": 3, "name": "Keyboard", "price": 75, "stock": 20}
]]

// Find products under $100
// Calculate total inventory value
// Find products with low stock (< 10)
```

Exercise 2: Student Grades

Work with student grade data:

```
["def", "students", [
  {"name": "Alice", "grades": [85, 92, 78, 96]},
  {"name": "Bob", "grades": [76, 84, 91, 88]},
  {"name": "Charlie", "grades": [92, 95, 89, 94]}
]]

// Calculate average grade for each student
// Find the student with the highest average
// List all grades above 90
```

4.3.11 Next Steps

- Learn about JSON Objects as first-class data structures
- Explore functions for data processing
- Try distributed computing with data

Working with data in JSL is straightforward because everything is JSON. The functional approach with `map`, `filter`, and `reduce` makes data transformation both powerful and readable.

5. Host Interaction

5.1 JSL Host Interaction Protocol (JHIP) - Version 1.0

5.1.1 Introduction

The JSL Host Interaction Protocol (JHIP) defines how JSL programs interact with the host environment for side effects. JSL's core philosophy is to reify effects as data - side effects are described as JSON messages rather than executed directly, allowing the host environment to control, audit, and secure all external interactions.

5.1.2 Core Principles

- **Effect Reification**: Side effects are described as data, not executed directly
- **Host Authority**: The host controls what operations are permitted and how they execute
- **JSON-Native**: All messages are valid JSON for universal compatibility
- **Synchronous Model**: From JSL's perspective, host operations are synchronous
- **Capability-Based Security**: Hosts provide only the capabilities they choose to expose

5.1.3 Request Structure

JSL programs request host operations using the `host` special form:

```
["host", "command", "arg1", "arg2", ...]
```

This creates a request message with the following structure:

```
{
  "command": "string",
  "args": ["arg1", "arg2", ...]
}
```

Request Examples

File Operations:

```
// JSL code
["host", "@file/read", "@/tmp/data.txt"]

// Request message
{
  "command": "file/read",
  "args": ["/tmp/data.txt"]
}
```

HTTP Requests:

```
// JSL code
["host", "@http/get", "@https://api.example.com/users", {"@Authorization": "@Bearer token"}]

// Request message
{
  "command": "http/get",
  "args": ["https://api.example.com/users", {"Authorization": "Bearer token"}]
}
```

Logging:

```
// JSL code
["host", "@log/info", "@User logged in", {"@user_id": 123}]

// Request message
{
  "command": "log/info",
```

```
  "args": ["User logged in", {"user_id": 123}]
}
```

5.1.4 Response Structure

The host responds with either a success value or an error object.

Success Response

Any valid JSON value represents success:

```
// File read success
"file content as string"

// HTTP response success
{
  "status": 200,
  "headers": {"content-type": "application/json"},
  "body": {"users": [...]}
}

// Operation with no return value
null
```

Error Response

Errors use a standard structure to distinguish them from successful `null`, `false`, or empty results:

```
{
  "$jsl_error": {
    "type": "ErrorType",
    "message": "Human readable description",
    "details": {}
  }
}
```

Error Fields: - `type`: Error category (e.g., "FileNotFound", "NetworkError", "PermissionDenied") - `message`: Clear description for developers - `details`: Additional structured information (optional)

Error Examples:

```
// File not found
{
  "$jsl_error": {
    "type": "FileNotFound",
    "message": "File does not exist",
    "details": {
      "path": "/tmp/missing.txt",
      "operation": "file/read"
    }
  }
}

// Permission denied
{
  "$jsl_error": {
    "type": "PermissionDenied",
    "message": "Insufficient permissions for operation",
    "details": {
      "operation": "file/write",
      "path": "/etc/passwd",
      "required_permission": "root"
    }
  }
}

// Network error
{
  "$jsl_error": {
    "type": "NetworkError",
    "message": "Connection timeout",
    "details": {
      "url": "https://api.example.com",
      "timeout_ms": 5000
    }
  }
}
```

5.1.5 Standard Commands

While hosts define their own command sets, these common patterns are recommended:

File System

- `@file/read` - Read file content as string
- `@file/write` - Write string to file
- `@file/exists` - Check if file exists
- `@file/list` - List directory contents
- `@file/delete` - Delete file or directory

HTTP

- `@http/get` - GET request
- `@http/post` - POST request
- `@http/put` - PUT request
- `@http/delete` - DELETE request

Logging

- `@log/debug` - Debug level log
- `@log/info` - Info level log
- `@log/warn` - Warning level log
- `@log/error` - Error level log

System

- `@env/get` - Get environment variable
- `@time/now` - Current timestamp
- `@random/uuid` - Generate UUID
- `@process/exec` - Execute system command

5.1.6 Protocol Flow

1. **JSL Evaluation**: JSL encounters `["host", "command", ...args]`
2. **Argument Evaluation**: All arguments are evaluated to JSON values
3. **Message Construction**: Create request message with command and args
4. **Host Processing**: Host validates, executes, and responds
5. **Response Handling**: Success value returned or error thrown in JSL

5.1.7 Security Model

JHIP implements capability-based security:

- **Host Controls Access**: Only commands explicitly enabled by the host are available
- **Argument Validation**: Host validates all arguments before execution
- **Resource Limits**: Host can impose limits on operations (file size, request timeouts, etc.)
- **Audit Trail**: All host interactions can be logged for security analysis

5.1.8 Implementation Notes

Error Handling in JSL

JSL implementations should convert JHIP error responses into JSL errors:

```
// If host returns error, JSL should throw
["try",
  ["host", "file/read", "/missing.txt"],
  ["lambda", ["err"],
    ["get", "err", "message"]]]
```

Async Implementation

While JSL sees synchronous operations, hosts may implement async processing:

- Queue requests for batch processing
- Use connection pooling for HTTP requests
- Implement timeout and retry logic
- Cache results when appropriate

Testing

JHIP enables easy testing by mocking host responses:

```
// Mock successful file read
{"command": "file/read", "args": ["/data.txt"]}
→ "mocked file content"

// Mock error response
{"command": "file/read", "args": ["/missing.txt"]}
→ {"$jsl_error": {"type": "FileNotFound", "message": "File not found"}}
```

5.2 Host Commands

5.2.1 Overview

JHIP defines the protocol structure for host interactions, but **all commands are host-specific** and defined by individual host implementations. Each JSL host chooses which commands to support based on their specific environment and use cases. This extensibility allows JSL to integrate with any system while maintaining the security and audit benefits of the JHIP protocol.

There are no "built-in" or "standard" commands - JHIP is purely a communication protocol. However, community conventions have emerged for common operations.

5.2.2 Community Conventions

While hosts define their own command sets, these naming patterns are commonly used:

File System Operations

```
["host", "@file/read", "@/path/to/file"]
["host", "@file/write", "@/path/to/file", "@content"]
["host", "@file/exists", "@/path/to/file"]
["host", "@file/list", "@/path/to/directory"]
["host", "@file/delete", "@/path/to/file"]
```

HTTP Operations

```
["host", "@http/get", "@https://api.example.com/data"]
["host", "@http/post", "@https://api.example.com/submit", {"@key": "@value"}]
["host", "@http/put", "@https://api.example.com/update", {"@data": "@updated"}]
["host", "@http/delete", "@https://api.example.com/item/123"]
```

Logging Operations

```
["host", "@log/debug", "@Debug message", {"@context": "@additional info"}]
["host", "@log/info", "@Application started successfully"]
["host", "@log/warn", "@Warning message"]
["host", "@log/error", "@Error occurred", {"@error_code": 500}]
```

System Operations

```
["host", "@env/get", "@PATH"]
["host", "@time/now"]
["host", "@time/format", "@2025-01-01T12:00:00Z", "@YYYY-MM-DD"]
["host", "@random/uuid"]
["host", "@process/exec", "@ls", ["@", ["@-la", "@/tmp"]]]
```

Database Operations

```
["host", "@db/query", "@SELECT * FROM users WHERE active = ?", ["@", [true]]]
["host", "@db/transaction", ["@", [
  ["@INSERT INTO logs (message) VALUES (?)", ["@", ["@Log entry 1"]]],
  ["@INSERT INTO logs (message) VALUES (?)", ["@", ["@Log entry 2"]]]
]]]
```

Cryptographic Operations

```
["host", "@crypto/hash", "@sha256", "@data to hash"]
["host", "@crypto/random", 32]
```

5.2.3 Command Design Principles

Naming Conventions

Host commands should follow a hierarchical naming structure:

```
<namespace>/<category>/<operation>
```

Examples: - `myapp/user/create` - `aws/s3/upload` - `database/postgres/query` - `ml/tensorflow/predict`

Command Categories

CLOUD SERVICES

```
["host", "@aws/s3/upload", "@bucket-name", "@key", "data"]
["host", "@gcp/storage/download", "@bucket", "@object"]
["host", "@azure/blob/delete", "@container", "@blob-name"]
```

MACHINE LEARNING

```
["host", "@tensorflow/predict", "@model-id", {"@features": ["@", [1, 2, 3]]}]
["host", "@pytorch/train", "@model-config", "training-data"]
```

BUSINESS LOGIC

```
["host", "@ecommerce/order/create", {"@product": "@123", "@quantity": 2}]
["host", "@crm/contact/update", "@contact-id", {"@email": "@new@example.com"}]
```

5.2.4 Implementation Guidelines

Command Handler Interface

```
class HostCommandHandler:
    def __init__(self, command_id: str):
        self.command_id = command_id

    def validate_args(self, args: list) -> bool:
        """Validate command arguments"""
        # Implement argument validation
        pass

    def execute(self, args: list) -> any:
        """Execute the command and return result"""
        # Implement command logic
        pass

    def get_permissions(self) -> list:
        """Return required permissions for this command"""
        # Return list of required permissions
        pass
```

Registration Pattern

```
# Register host command handlers
jsl_host.register_command("file/read", FileReadHandler())
jsl_host.register_command("http/get", HttpGetHandler())
jsl_host.register_command("myapp/user/create", UserCreateHandler())
```

Error Handling

All commands must return errors in standard JHIP format:

```
{
  "$jsl_error": {
    "type": "CommandError",
    "message": "File not found",
    "details": {
      "command": "file/read",
      "path": "/nonexistent/file"
    }
  }
}
```

5.2.5 Example Command Implementations

File Read Command

```python
class FileReadHandler(HostCommandHandler):
    def validate_args(self, args: list) -> bool:
        return len(args) == 1 and isinstance(args[0], str)

    def execute(self, args: list) -> str:
        path = args[0]
        try:
            with open(path, 'r') as file:
                return file.read()
        except FileNotFoundError:
            return {
                "$jsl_error": {
                    "type": "FileNotFound",
                    "message": f"File does not exist: {path}",
                    "details": {"path": path, "operation": "file/read"}
                }
            }
        except PermissionError:
            return {
                "$jsl_error": {
                    "type": "PermissionDenied",
                    "message": f"Permission denied: {path}",
                    "details": {"path": path, "operation": "file/read"}
                }
            }

    def get_permissions(self) -> list:
        return ["file.read"]
```

HTTP GET Command

```python
import requests

class HttpGetHandler(HostCommandHandler):
    def validate_args(self, args: list) -> bool:
        return len(args) >= 1 and isinstance(args[0], str)

    def execute(self, args: list) -> dict:
        url = args[0]
        headers = args[1] if len(args) > 1 else {}

        try:
            response = requests.get(url, headers=headers, timeout=30)
            return {
                "status": response.status_code,
                "headers": dict(response.headers),
                "body": response.text
            }
        except requests.exceptions.Timeout:
            return {
                "$jsl_error": {
                    "type": "NetworkError",
                    "message": "Request timeout",
                    "details": {"url": url, "timeout_ms": 30000}
                }
            }
        except requests.exceptions.ConnectionError:
            return {
                "$jsl_error": {
                    "type": "NetworkError",
                    "message": "Connection failed",
                    "details": {"url": url}
                }
            }

    def get_permissions(self) -> list:
        return ["network.http"]
```

Custom Business Logic Command

```python
class UserCreateHandler(HostCommandHandler):
    def validate_args(self, args: list) -> bool:
        if len(args) != 1 or not isinstance(args[0], dict):
            return False

        user_data = args[0]
        required_fields = ["email", "name"]
        return all(field in user_data for field in required_fields)

    def execute(self, args: list) -> dict:
        user_data = args[0]
```

```
        # Validate email format
        import re
        if not re.match(r"[^@]+@[^@]+\.[^@]+", user_data["email"]):
            return {
                "$jsl_error": {
                    "type": "ValidationError",
                    "message": "Invalid email format",
                    "details": {"field": "email", "value": user_data["email"]}
                }
            }

        # Check for duplicate email
        if self.email_exists(user_data["email"]):
            return {
                "$jsl_error": {
                    "type": "DuplicateError",
                    "message": "Email already exists",
                    "details": {"field": "email", "value": user_data["email"]}
                }
            }

        # Create user
        user_id = self.create_user_in_database(user_data)

        return {
            "user_id": user_id,
            "email": user_data["email"],
            "created_at": datetime.utcnow().isoformat()
        }

    def get_permissions(self) -> list:
        return ["user.create"]
```

5.2.6 Security Considerations

Permission System

```
class PermissionChecker:
    def check_command_permission(self, command_id: str, user_context: dict) -> bool:
        # Example permission logic
        user_permissions = user_context.get("permissions", [])

        # Admin bypass
        if "admin" in user_context.get("roles", []):
            return True

        # Check specific command permissions
        if command_id.startswith("file/"):
            return "file.access" in user_permissions

        if command_id.startswith("http/"):
            return "network.http" in user_permissions

        # Custom business logic permissions
        if command_id.startswith("myapp/"):
            return f"myapp.{command_id.split('/')[-1]}" in user_permissions

        return False
```

Input Validation

```
def validate_command_args(command_id: str, args: list):
    """Validate arguments for security and correctness"""

    if command_id == "file/read":
        if len(args) != 1 or not isinstance(args[0], str):
            raise ValueError("file/read requires exactly one string argument")

        # Prevent directory traversal
        path = os.path.normpath(args[0])
        if path.startswith("../") or "/../" in path:
            raise ValueError("Directory traversal not allowed")

    elif command_id == "process/exec":
        if len(args) < 1:
            raise ValueError("process/exec requires at least one argument")

        # Whitelist allowed commands
        allowed_commands = ["ls", "cat", "echo", "date"]
        if args[0] not in allowed_commands:
            raise ValueError(f"Command not allowed: {args[0]}")
```

Resource Limits

```
class ResourceLimiter:
    def __init__(self):
        self.limits = {
            "max_execution_time": 30,  # seconds
            "max_memory_usage": 100 * 1024 * 1024,  # 100MB
            "max_file_size": 10 * 1024 * 1024,  # 10MB
            "max_network_requests_per_minute": 60
        }
        self.usage_tracking = {}

    def check_limits(self, command_id: str, user_id: str):
        # Implement rate limiting and resource checking
        current_time = time.time()

        # Check rate limits
        user_requests = self.usage_tracking.get(user_id, [])
        recent_requests = [t for t in user_requests if current_time - t < 60]

        if len(recent_requests) >= self.limits["max_network_requests_per_minute"]:
            raise Exception("Rate limit exceeded")

        # Track this request
        self.usage_tracking[user_id] = recent_requests + [current_time]
```

5.2.7 Testing Host Commands

Unit Testing

```
def test_file_read_command():
    handler = FileReadHandler()

    # Test successful read
    with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
        f.write("test content")
        f.flush()

        result = handler.execute([f.name])
        assert result == "test content"

    # Test file not found
    result = handler.execute(["/nonexistent/file"])
    assert "$jsl_error" in result
    assert result["$jsl_error"]["type"] == "FileNotFound"
```

Integration Testing

```
def test_command_through_jsl():
    # Test command through JSL runtime
    jsl_code = ["host", "@file/read", "@/tmp/test.txt"]

    # Mock the host command
    with patch('jsl_host.execute_command') as mock_execute:
        mock_execute.return_value = "mocked file content"

        result = jsl_runtime.evaluate(jsl_code)
        assert result == "mocked file content"

        mock_execute.assert_called_once_with("file/read", ["/tmp/test.txt"])
```

5.2.8 Documentation Template

Use this template when documenting host commands:

Command: `myapp/user/create`

Description: Creates a new user account in the system.

Parameters: - `user_data` (object): User information - `email` (string, required): User's email address - `name` (string, required): User's full name - `role` (string, optional): User role, defaults to "user"

Returns: - `user_id` (string): Unique identifier for created user - `email` (string): Confirmed email address - `created_at` (string): ISO timestamp of creation

Permissions Required: `user.create`

Example:

```
["host", "@myapp/user/create", {
  "@email": "@john@example.com",
  "@name": "@John Doe",
  "@role": "@editor"
}]
```

Error Types: - `DUPLICATE_EMAIL` : Email address already exists - `INVALID_EMAIL` : Email format is invalid - `PERMISSION_DENIED` : Insufficient permissions

5.2.9 Best Practices

Design Guidelines

1. **Atomic Operations**: Commands should perform single, well-defined operations
2. **Idempotency**: Where possible, commands should be idempotent
3. **Error Transparency**: Provide clear, actionable error messages
4. **Resource Efficiency**: Minimize resource usage and implement proper cleanup
5. **Backward Compatibility**: Maintain compatibility when updating commands

Performance Optimization

1. **Caching**: Cache frequently accessed data
2. **Connection Pooling**: Reuse database and network connections
3. **Async Operations**: Use async patterns for I/O operations
4. **Batch Processing**: Support batch operations where appropriate

Security Best Practices

1. **Input Sanitization**: Always validate and sanitize inputs
2. **Principle of Least Privilege**: Grant minimal required permissions
3. **Audit Logging**: Log all command executions for security analysis
4. **Rate Limiting**: Implement rate limits to prevent abuse
5. **Resource Limits**: Set timeouts and size limits on operations

Host commands are how JSL integrates with the real world while maintaining security, transparency, and auditability. By following these guidelines and community conventions, you can create robust, secure, and maintainable host command implementations that work well with the broader JSL ecosystem.

6. Architecture

6.1 Design Philosophy

6.1.1 The Problem with Traditional Code Mobility

Modern distributed systems require seamless code mobility—the ability to send executable code across network boundaries, store it in databases, and reconstruct it in different runtime environments. Traditional approaches face fundamental challenges:

1. Serialization Complexity

Most languages require complex serialization frameworks (e.g., pickle, protobuf) that are brittle, version-dependent, and often insecure.

2. Runtime Dependencies

Serialized code often depends on specific runtime versions, libraries, or execution contexts that may not be available on the receiving end.

3. Security Vulnerabilities

Deserializing code can execute arbitrary instructions, creating significant attack vectors.

4. Platform Lock-in

Serialization formats are often language-specific, preventing cross-platform code sharing.

6.1.2 The JSL Solution

JSL solves these problems by making JSON the native representation for both data and code. This design enables powerful properties for network-native programming. The core language is purely functional and safe, while interactions with the host system are reified as data and controlled through a programmable, capability-based environment model.

6.1.3 Theoretical Foundations

Homoiconicity

Like classic Lisps, JSL is **homoiconic**, meaning code and data share the same representation. However, instead of S-expressions, JSL uses JSON—a universally supported and standardized format.

Key Benefits:

- **No Parsing Ambiguity**: JSON has a precise, standardized grammar.
- **Universal Tooling**: Every major language and platform can handle JSON.
- **Network Transparency**: Valid JSON travels safely across all network protocols.
- **Human Readability**: Code can be inspected and modified with standard text tools.

Verifiable, Serializable State

Handling closures (functions that capture their lexical environment) is a primary challenge in code mobility. JSL solves this with a **content-addressable storage model** that elegantly handles circular references and makes program state verifiable, efficient, and safely serializable.

- **Content-Addressable Objects:** Every complex object (closure or environment) is identified by a unique hash of its contents. Objects are stored in a hash table and referenced by their content hashes, naturally handling circular references.
- **Serializable Closures:** A closure is serialized as a JSON object containing its parameters, body, and a reference to its captured environment. The environment reference uses the content-addressable format `{"__ref__": "hash"}`.
- **Efficient Sharing:** Identical objects share the same hash, avoiding duplication and creating an efficient storage model for complex object graphs.
- **Format Versioning:** The serialized payload includes `__cas_version__` to enable format evolution while maintaining backward compatibility.

This architecture ensures that a serialized JSL program can handle arbitrarily complex object relationships while remaining a verifiable, self-contained unit of computation.

Wire-Format Transparency

Every JSL value can be serialized to JSON and reconstructed identically in any compliant runtime. This enables:

- **Database Storage**: Store executable code with ACID properties.
- **HTTP Transmission**: Send functions using standard web infrastructure.
- **Cross-Language Interoperability**: Leverage JSON's universal support.
- **Audit Trails**: Create reproducible records of code execution.
- **Version Control**: Use standard JSON diff/merge tools to manage code.

6.1.4 Practical Applications

1. Distributed Computing

Send computations to where data resides, rather than moving data.

```
["lambda", ["data"],
  ["filter",
    ["lambda", ["record"], ["=", ["get", "record", "status"], "active"]],
    "data"]]
```

2. Edge Computing

Deploy and update logic on edge devices dynamically.

```
["lambda", ["sensor_reading"],
  ["if", [">", "sensor_reading", 75],
    ["send-alert", "High temperature detected"],
    ["log", "Normal reading:", "sensor_reading"]]]
```

3. Database Functions

Store and execute business logic directly in databases.

```
["lambda", ["user_id", "new_status"],
  ["host", "db/update", "users",
    {"id": "user_id"},
    {"$set": {"status": "new_status"}}]]
```

6.1.5 SICP-Inspired Design

JSL follows the elegant principles outlined in *"Structure and Interpretation of Computer Programs"*:

- **Simplicity**: Everything is built from a small set of primitives (atoms, lists, functions).
- **Composability**: Complex operations are created by combining simple ones.
- **Abstraction**: Higher-level concepts are built on lower-level foundations.
- **Uniformity**: A consistent evaluation model applies throughout the language.
- **Extensibility**: New capabilities are added through composition, not special cases.

This approach creates a language that is both theoretically elegant and practically useful for distributed computing.

6.2 JSL Abstract Machine Specification

6.2.1 1. Overview

The JSL Abstract Machine is a stack-based virtual machine designed to execute JSL postfix bytecode. It provides a simple, resumable, and network-friendly execution model.

6.2.2 2. Machine Architecture

2.1 Components

The abstract machine consists of:

```
Machine M = ⟨S, E, C, D, R⟩
```

Where: - **S** (Stack): Value stack for operands and results - **E** (Environment): Variable bindings - **C** (Code): Array of instructions (postfix) - **D** (Dump): Saved states for function calls (future) - **R** (Resources): Resource budget and consumption

2.2 Value Domain

```
Value v ::= n           (number)
         | b           (boolean)
         | null        (null)
         | s           (string)
         | [v₁,...,vₙ] (list)
         | {k:v,...}   (dictionary)
         | ⟨λ,p,b,E⟩   (closure)
```

2.3 Instruction Set

```
Instruction i ::= v             (push value)
                | x             (push variable)
                | op            (binary operator)
                | (op, n)       (n-ary operator)
                | MARK          (stack marker)
                | CALL n        (function call)
                | RET           (return)
```

6.2.3 3. Operational Semantics

3.1 Basic Transitions

VALUE PUSH

```
    v is a literal value
──────────────────────────────────────────
⟨S, E, v·C, D, R⟩ → ⟨v·S, E, C, D, R'⟩
```

Where R' = R with gas decremented by 1.

VARIABLE LOOKUP

```
    x ∈ dom(E)    E(x) = v
──────────────────────────────────────────
⟨S, E, x·C, D, R⟩ → ⟨v·S, E, C, D, R'⟩
```

Where R' = R with gas decremented by 2.

BINARY OPERATION

```
    ⊕ is a binary operator
──────────────────────────────────────────
⟨v₂·v₁·S, E, ⊕·C, D, R⟩ → ⟨(v₁⊕v₂)·S, E, C, D, R'⟩
```

Where R' = R with gas decremented by 3.

N-ARY OPERATION

$$\frac{\oplus \text{ is an n-ary operator} \quad |S| \geq n}{\langle v_n \cdot \ldots \cdot v_1 \cdot S, E, (\oplus,n) \cdot C, D, R \rangle \rightarrow \langle \oplus(v_1,\ldots,v_n) \cdot S, E, C, D, R' \rangle}$$

Where R' = R with gas decremented by (3 + n).

3.2 Control Flow (Future Extension)

FUNCTION CALL

$$\frac{v = \langle \lambda, p_1 \ldots p_n, b, E' \rangle \quad |S| \geq n}{\langle v_n \cdot \ldots \cdot v_1 \cdot v \cdot S, E, \text{CALL } n \cdot C, D, R \rangle \rightarrow \langle [], E'[p_1 \mapsto v_1, \ldots, p_n \mapsto v_n], b, (S,E,C) \cdot D, R' \rangle}$$

RETURN

$$\frac{D = (S',E',C') \cdot D'}{\langle v \cdot S, E, \text{RET} \cdot C, D, R \rangle \rightarrow \langle v \cdot S', E', C', D', R \rangle}$$

3.3 Error States

STACK UNDERFLOW

$$\frac{|S| < n \quad (\oplus,n) \text{ requires n arguments}}{\langle S, E, (\oplus,n) \cdot C, D, R \rangle \rightarrow \text{ERROR: Stack underflow}}$$

UNDEFINED VARIABLE

$$\frac{x \notin \text{dom}(E)}{\langle S, E, x \cdot C, D, R \rangle \rightarrow \text{ERROR: Undefined variable } x}$$

RESOURCE EXHAUSTION

$$\frac{R.\text{gas} \leq 0}{\langle S, E, C, D, R \rangle \rightarrow \text{PAUSE: } \langle S, E, C, D, R \rangle}$$

6.2.4 4. Execution Algorithm

4.1 Basic Execution Loop

```python
def execute(code, env, resources):
    S = []  # Stack
    C = code  # Code pointer
    E = env  # Environment
    R = resources  # Resources

    while C:
        if R and R.gas <= 0:
            return PAUSE(S, E, C, R)

        instr = C[0]
        C = C[1:]

        if is_value(instr):
            S = [instr] + S
            R.gas -= 1

        elif is_variable(instr):
            if instr in E:
                S = [E[instr]] + S
                R.gas -= 2
            else:
                return ERROR(f"Undefined: {instr}")

        elif is_binary_op(instr):
            if len(S) < 2:
                return ERROR("Stack underflow")
            v2, v1 = S[0], S[1]
            S = [apply_binary(instr, v1, v2)] + S[2:]
            R.gas -= 3
```

```
        elif is_nary_op(instr):
            op, n = instr
            if len(S) < n:
                return ERROR("Stack underflow")
            args = S[:n]
            S = [apply_nary(op, args)] + S[n:]
            R.gas -= (3 + n)

    if len(S) == 1:
        return S[0]
    else:
        return ERROR(f"Invalid final stack: {S}")
```

4.2 Resumable Execution

```
def execute_resumable(state_or_code, env=None):
    if is_saved_state(state_or_code):
        S, E, C, R = restore_state(state_or_code)
    else:
        S = []
        E = env or {}
        C = state_or_code
        R = default_resources()

    result = execute_step(S, E, C, R)

    if is_pause(result):
        return None, save_state(result)
    else:
        return result, None
```

6.2.5 5. Resource Model

5.1 Gas Costs

Operation	Gas Cost	Rationale
Push literal	1	Minimal cost
Variable lookup	2	Environment search
Binary operation	3	Computation
N-ary operation	3+n	Scales with arguments
Function call	10	Context switch
List creation	1+n	Allocation
Dictionary creation	1+2n	Key-value pairs

5.2 Memory Limits

```
MemoryLimit = {
    max_stack_depth: 10000,
    max_collection_size: 1000000,
    max_string_length: 1000000,
    max_env_depth: 1000
}
```

5.3 Time Limits

```
def check_time_limit(start_time, limit):
    if time.now() - start_time > limit:
        raise TimeExhausted()
```

6.2.6 6. Optimization Opportunities

6.1 Instruction Fusion

Combine common patterns:

```
[2, 3, +] → [PUSH_ADD, 2, 3]
[x, y, *] → [MUL_VARS, x, y]
```

6.2 Constant Folding

Pre-compute constant expressions:

```
[2, 3, +, 4, *] → [20]  ; Computed at compile time
```

6.3 Stack Caching

Keep top stack values in registers:

```
TOS (Top of Stack) → Register 0
TOS-1 → Register 1
```

6.4 Environment Optimization

Cache frequently accessed variables:

```
E_cache = LRU_cache(size=32)
```

6.2.7 7. Implementation Strategies

7.1 Direct Threading

```c
typedef void (*instruction_fn)(Machine*);

instruction_fn dispatch_table[] = {
    [OP_PUSH] = do_push,
    [OP_ADD] = do_add,
    [OP_MUL] = do_mul,
    // ...
};

void execute(Machine* m) {
    while (m->pc < m->code_len) {
        dispatch_table[m->code[m->pc++]](m);
    }
}
```

7.2 Computed Goto (GCC)

```c
void* dispatch_table[] = {
    &&do_push,
    &&do_add,
    &&do_mul,
    // ...
};

#define DISPATCH() goto *dispatch_table[code[pc++]]

execute:
    DISPATCH();

do_push:
    stack[++sp] = code[pc++];
    DISPATCH();

do_add:
    sp--;
    stack[sp] += stack[sp+1];
    DISPATCH();
```

7.3 JIT Compilation

```
def jit_compile(postfix):
    """Compile postfix to native code."""
    native_code = []

    for instr in postfix:
        if isinstance(instr, int):
            native_code.append(f"PUSH_IMM {instr}")
        elif instr == '+':
            native_code.append("POP_ADD")
        # ...

    return assemble(native_code)
```

6.2.8 8. Debugging Support

8.1 Stack Trace

```
Stack trace at pc=42:
  [0] main: [+, x, [*, y, 2]]
  [1] *: [*, y, 2]
  Stack: [10, 3, 2]
  Next: *
```

8.2 Breakpoints

```
breakpoints = {15, 27, 42}  # Instruction addresses

def execute_debug(code, env):
    for pc, instr in enumerate(code):
        if pc in breakpoints:
            debug_prompt(stack, env, code, pc)
        execute_instruction(instr)
```

8.3 State Inspection

```
def inspect_state(machine):
    return {
        'stack': machine.stack,
        'env': machine.env,
        'pc': machine.pc,
        'gas_used': machine.initial_gas - machine.gas,
        'next_instruction': machine.code[machine.pc]
    }
```

6.2.9 9. Comparison with Other VMs

Feature	JSL VM	JVM	Python VM	Lua VM
Model	Stack	Stack	Stack	Register
Bytecode	JSON	Binary	Binary	Binary
Resumable	Yes	No	No	Yes (coroutines)
Serializable	Yes	No	Partial	No
Types	Dynamic	Static	Dynamic	Dynamic
GC	Host	Yes	Yes	Yes

6.2.10 10. Performance Characteristics

10.1 Time Complexity

Operation	Complexity
Push	O(1)
Pop	O(1)
Binary op	O(1)
Variable lookup	O(log n) average
List creation	O(n)
Function call	O(1)

10.2 Space Complexity

Structure	Complexity
Stack	O(n) expressions
Environment	O(m) variables
Code	O(k) instructions
Total	O(n + m + k)

6.2.11 11. Security Considerations

11.1 Resource Isolation

Each execution has isolated resources:

```
resources = ResourceBudget(
    gas=10000,
    memory=1_000_000,
    time_limit=1.0
)
```

11.2 Capability Security

No ambient authority - all effects through capabilities:

```
capabilities = {
    'file_read': FileReadCapability('/allowed/path'),
    'network': NetworkCapability(['allowed.host'])
}
```

11.3 Sandboxing

```
def sandbox_execute(untrusted_code):
    sandbox = Sandbox(
        max_stack=1000,
        max_time=1.0,
        no_host_access=True
    )
    return sandbox.execute(untrusted_code)
```

6.2.12 12. Example Execution Traces

12.1 Simple Arithmetic

Code: `[2, 3, +, 4, *]`

```
Step | Stack | Code        | Action
-----|-------|-------------|--------
0    | []    | 2 3 + 4 *   | Initial
1    | [2]   | 3 + 4 *     | Push 2
2    | [2,3] | + 4 *       | Push 3
3    | [5]   | 4 *         | Add
4    | [5,4] | *           | Push 4
5    | [20]  | ε           | Multiply
```

12.2 With Variables

Code: `[x, y, +]` with env `{x: 10, y: 20}`

```
Step | Stack    | Code  | Action
-----|----------|-------|----------
0    | []       | x y + | Initial
1    | [10]     | y +   | Lookup x
2    | [10,20]  | +     | Lookup y
3    | [30]     | ε     | Add
```

6.2.13 13. Future Directions

1. **Tail Call Optimization**: Reuse stack frames
2. **Lazy Evaluation**: Thunks and promises
3. **Parallel Execution**: Multiple stacks
4. **Native Code Generation**: LLVM backend
5. **Persistent Data Structures**: Immutable collections

This specification defines the JSL Abstract Machine v1.0, providing a formal foundation for JSL execution.

6.3 Stack-Based Evaluation in JSL

6.3.1 Overview

JSL now supports two evaluation strategies: 1. **Recursive evaluation** - Traditional tree-walking interpreter (simple but limited) 2. **Stack-based evaluation** - Compiles to postfix bytecode for efficient execution

6.3.2 Why Stack-Based Evaluation?

Limitations of Recursive Evaluation

The recursive evaluator in `jsl/core.py` has several fundamental limitations:

1. **No true resumption**: When resources are exhausted mid-recursion, we can only capture the top-level expression. The computation restarts from the beginning, making no progress.
2. **Stack overflow risk**: Deep recursion can exceed Python's call stack limit.
3. **Difficult optimization**: Each node visit involves function calls and environment lookups.
4. **Imprecise resource tracking**: Hard to track exact costs mid-recursion.

Advantages of Stack-Based Evaluation

1. **Perfect resumption**: Save the stack and program counter, resume exactly where you left off.
2. **No recursion limits**: Uses a data stack, not the call stack.
3. **Optimization opportunities**: Postfix bytecode can be optimized, cached, and analyzed.
4. **Network-friendly**: Postfix arrays are more compact than S-expressions for transmission.
5. **Clear execution model**: Each instruction is atomic and predictable.

6.3.3 Architecture

Compilation Pipeline

```
S-Expression → Compiler → Postfix → Stack Evaluator → Result
['+', 2, 3]  →          → [2,3,'+'] →               → 5
```

Key Components

- `jsl/compiler.py` : Converts S-expressions to postfix notation
- `jsl/stack_evaluator.py` : Executes postfix using a value stack
- `jsl/eval_modes.py` : Unified interface for both evaluators

N-Arity Handling

For operators with variable arity ($0 \leq n < \infty$), we encode arity explicitly:

```
['+']          → [('+', 0)]      # 0-arity: sum() = 0
['+', 5]       → [5, ('+', 1)]    # 1-arity: 5
['+', 1, 2, 3] → [1, 2, 3, ('+', 3)]  # n-arity: 1+2+3
```

6.3.4 Usage Examples

Basic Evaluation

```
from jsl.compiler import compile_to_postfix
from jsl.stack_evaluator import StackEvaluator

# Compile S-expression to postfix
expr = ['*', ['+', 2, 3], 4]
postfix = compile_to_postfix(expr)  # [2, 3, '+', 4, '*']

# Evaluate
evaluator = StackEvaluator()
result = evaluator.eval(postfix)  # 20
```

Resumable Evaluation

```
# Execute with limited steps (simulating resource limits)
expr = ['*', ['+', 10, 20], ['-', 100, 50]]
postfix = compile_to_postfix(expr)

evaluator = StackEvaluator()
result, state = evaluator.eval_partial(postfix, max_steps=2)
# result = None, state contains stack and pc

# Resume later
result, state = evaluator.eval_partial(postfix, max_steps=10, state=state)
# result = 1500, state = None (complete)
```

Unified Interface

```
from jsl.eval_modes import JSLEvaluator, EvalMode

# Use recursive evaluator (default)
eval1 = JSLEvaluator(mode=EvalMode.RECURSIVE)
result = eval1.eval(['+', 2, 3])  # 5

# Use stack evaluator
eval2 = JSLEvaluator(mode=EvalMode.STACK)
result = eval2.eval(['+', 2, 3])  # 5

# Both produce identical results!
```

6.3.5 Postfix as Primary Representation

In distributed/networked scenarios, postfix can become the primary code representation:

```
# Instead of sharing S-expressions:
send_over_network(['+', [1, 2], [3, 4]])  # Nested structure

# Share postfix directly:
send_over_network([1, 2, '+', 3, 4, '+', '+'])  # Flat array
```

Benefits: - More compact (flat array vs nested) - No parsing ambiguity - Direct execution without compilation - Trivial serialization

6.3.6 Testing

The same test suite runs on both evaluators, proving functional equivalence:

```
# tests/test_both_evaluators.py
@pytest.fixture(params=[RecursiveEvaluator, StackEvaluator])
def evaluator(request):
    return request.param()

def test_arithmetic(evaluator):
    assert evaluator.eval(['+', 2, 3]) == 5
    # This test runs twice - once per evaluator!
```

6.3.7 Future Work

1. **Optimize postfix**: Dead code elimination, constant folding
2. **JIT compilation**: Compile hot paths to native code

3. **Streaming evaluation**: Process postfix as it arrives over network
4. **Parallel execution**: Some postfix sequences can run in parallel
5. **Add special forms**: Implement if, let, lambda in stack evaluator

6.3.8 Conclusion

Stack-based evaluation provides a production-ready alternative to recursive evaluation, with better resumption, optimization opportunities, and network characteristics. The postfix representation can serve as JSL's "bytecode" - the actual computational format, with S-expressions as the human-friendly source language.

6.4 JSL Performance Philosophy

6.4.1 The Spectrum of Optimization

JSL's postfix representation could be further optimized into raw primitives:

```
Current postfix:     [2, 3, "+", 4, "*"]
Could become:        [PUSH_INT, 2, PUSH_INT, 3, ADD, PUSH_INT, 4, MUL]
Or even:            [0x12, 0x02, 0x12, 0x03, 0x20, 0x12, 0x04, 0x21]
```

6.4.2 Why We Don't

We deliberately keep the postfix representation as JSON-compatible values rather than raw bytecode because:

1. Network Transparency

```
// This can be sent over HTTP, stored in databases, inspected by humans
[10, 20, "+", 100, 50, "-", "*"]

// This is opaque binary data
[0x0A, 0x14, 0x20, 0x64, 0x32, 0x22, 0x21]
```

2. Debugging & Introspection

Our postfix is self-documenting:

```
[2, 3, "+"]  # Obviously: 2 + 3
[0x02, 0x03, 0x20]  # What does this mean?
```

3. Universal Serialization

JSON works everywhere - browsers, databases, message queues, REST APIs. Binary formats require custom parsers and version management.

4. The Real Bottleneck

For 99% of distributed computing tasks, the bottleneck isn't instruction dispatch but: - Network latency (100,000x slower than CPU) - Database queries (10,000x slower) - Serialization/deserialization - Coordination overhead

5. Mobility Over Speed

JSL prioritizes computation mobility over raw performance:

```
# This is our superpower - pause here, resume there
state = {
    "stack": [30, 100],
    "code": [50, "-", "*"],
    "pc": 2
}
# Can be sent to any machine, stored, inspected, modified
```

6.4.3 When You Need Real Speed

If you need maximum performance:

1. **Wrong tool**: JSL isn't for high-performance computing
2. **Use native code**: C++, Rust, or assembly for hot paths
3. **JIT compilation**: Compile hot JSL functions to native code
4. **Hybrid approach**: JSL for coordination, native for computation

```
# JSL for orchestration
["parallel-map",
  ["native-function", "fast_matrix_multiply"],
  large_dataset]

# Where fast_matrix_multiply is implemented in C++
```

6.4.4 The 80/20 Rule

80% of your code runs 20% of the time. JSL is perfect for that 80%: - Configuration - Orchestration
- Business logic - Data transformation - API endpoints

The other 20% that needs speed? Call out to native code.

6.4.5 Educational Value

Showing the progression of representations teaches important lessons:

```
Mathematical:    (λ (x) (× x x))
    ↓
S-expression:    (lambda (x) (* x x))
    ↓
JSON:           ["lambda", ["x"], ["*", "x", "x"]]
    ↓
Postfix:        ["x", "x", "*", ("lambda", 1)]
    ↓
Bytecode:       [LOAD_VAR, 0, LOAD_VAR, 0, MUL, MAKE_CLOSURE, 1]
    ↓
Assembly:       mov eax, [ebp-4]; mov ebx, [ebp-4]; imul eax, ebx
    ↓
Machine code:   8B 45 FC 8B 5D FC 0F AF C3
```

Each level trades human-readability for machine-efficiency. JSL stops at the postfix level because:

1. **It's still JSON** - universally readable
2. **It's resumable** - can pause/resume execution
3. **It's portable** - works on any platform
4. **It's inspectable** - can debug and understand
5. **It's fast enough** - for distributed/networked computing

6.4.6 The JSL Philosophy

> "Make it work, make it right, make it mobile. If you still need to make it fast, you're using the wrong tool."

JSL is about moving computation, not optimizing cycles. It's about expressing ideas clearly, not squeezing out nanoseconds. It's about universal computation, not platform-specific performance.

6.4.7 Conclusion

Could we compile JSL to raw bytecode? Yes. Should we? No.

The JSON-based postfix representation is the sweet spot: - Fast enough for real work - Clear enough for debugging - Portable enough for the network - Simple enough for implementation

If you need more speed than JSL provides, you don't need a faster JSL - you need a different language for that component. JSL's job is to coordinate, not to compete with C++.

Remember: **Premature optimization is the root of all evil** (Knuth), and in distributed systems, **the network is the computer** (Sun Microsystems). JSL embraces both principles.

6.5 Runtime Architecture

6.5.1 Overview

The JSL ecosystem is designed as a layered architecture that separates concerns and ensures both security and portability. Understanding these layers is crucial for implementing JSL systems and reasoning about code execution.

6.5.2 Architecture Layers

The JSL runtime can be conceptualized in six distinct layers:

1. Wire Layer (JSON)

The universal representation for JSL programs, data, and serialized closures. This is what gets transmitted over networks or stored in databases.

Characteristics: - Standard JSON format - Universal compatibility - Human-readable - Version-independent - Platform-agnostic

2. JSL Runtime/Interpreter

The core execution engine that processes JSL code:

PARSER

- Converts JSON into internal JSL abstract syntax
- Validates JSON structure against JSL grammar
- Handles syntax errors and malformed input

EVALUATOR

- Executes JSL code based on language semantics
- Handles special forms (`def`, `lambda`, `if`, `do`, `host`, etc.)
- Manages function applications and argument evaluation
- Implements lexical scoping rules

ENVIRONMENT MANAGER

- Manages lexical environments and scope resolution
- Handles variable binding and lookup
- Maintains environment chains for closures
- Supports environment serialization/deserialization

3. Prelude Layer

A foundational environment provided by the host runtime containing built-in functions and constants.

Key Properties: - Contains essential computational primitives (arithmetic, logic, list operations) - Not serialized with user code - Expected to be available in any compliant JSL runtime - Can be customized or extended by host implementations - Serves as the computational foundation for user programs

Examples: - Arithmetic operations: `+`, `-`, `*`, `/` - Comparison operators: `<`, `>`, `<=`, `>=`, `=` - List operations: `map`, `filter`, `reduce` - Type predicates: `null?`, `number?`, `string?`

4. User Code Layer

JSL programs and libraries written by developers. These are fully serializable and portable.

Characteristics: - Complete JSON serializability - Closure capture and reconstruction - Cross-runtime portability - Environment independence (beyond prelude)

5. Host Interaction Layer (JHIP)

When a JSL program evaluates a `["host", ...]` form, it generates a JHIP (JSL Host Interaction Protocol) request. This layer manages the interface between pure JSL computation and external effects.

Key Features: - Effect reification as data structures - Request-response protocol - Host authority over permitted operations - Audit trail capability - Security boundary

6. Host Environment

The runtime system that executes JSL code, manages resources, and enforces security policies.

Responsibilities: - JSL interpreter hosting - JHIP request processing - Resource management - Security policy enforcement - Capability provisioning

6.5.3 Serialization Architecture

A critical aspect of JSL's runtime architecture is its serialization system:

Closure Serialization

JSL `Closure` objects store: - Function parameters - Function body - Captured lexical environment (only user-defined variables)

Serialization Process: 1. Identify free variables in closure body 2. Extract relevant bindings from lexical environment 3. Serialize environment chain (user bindings only) 4. Exclude prelude bindings (reconstructed at runtime)

Environment Serialization

Environments (`Env` objects) are serialized as: - Dictionary of name-to-value mappings - Parent environment reference (if applicable) - Only user-defined bindings included

Reconstruction Process: 1. Recreate environment hierarchy 2. Restore user-defined bindings 3. Link to appropriate prelude environment 4. Validate binding completeness

6.5.4 Security Model

JSL's architecture provides security through multiple layers:

Capability Restriction

- All side effects must go through JHIP
- Host controls available operations
- Fine-grained permission model

Code Safety

- No native code execution
- JSON-based representation prevents code injection
- Deterministic evaluation (in pure subset)

Effect Reification

- Side effects are described as data

- Host can inspect, audit, or modify requests
- Clear separation between computation and effects

Sandboxing

- JSL programs run within interpreter bounds
- No direct system access
- Host-mediated resource access only

6.5.5 Implementation Considerations

Performance

- JSON parsing overhead
- Environment lookup chains
- Closure reconstruction costs
- JHIP communication latency

Memory Management

- Environment retention for closures
- Garbage collection of unused environments
- Serialization memory overhead

Error Handling

- JSON parsing errors
- Runtime evaluation errors
- JHIP communication failures
- Host capability denials

6.5.6 Deployment Patterns

Distributed Computing

```
Client Runtime -> JSON Code -> Remote Runtime -> Results
```

Database Functions

```
Application -> Stored JSL -> Database -> Executed Results
```

Microservice Communication

```
Service A -> JSL Function -> Service B -> Response
```

Edge Computing

```
Central Control -> JSL Logic -> Edge Devices -> Local Execution
```

This layered architecture ensures JSL maintains its core properties of safety, portability, and network-nativity while providing the flexibility needed for diverse deployment scenarios.

6.6 Security Model

6.6.1 Overview

JSL's security model is built on the principle of **effect reification** and **capability restriction**. Unlike traditional languages where security is layered on top, JSL's design makes security an intrinsic property of the language itself.

6.6.2 Core Security Principles

- **No Arbitrary Code Execution:** JSL code is data. This eliminates entire classes of vulnerabilities like buffer overflows and direct system calls.
- **Effect Reification:** All side effects are represented as data (e.g., `["host", "file/read", ...]`). This makes them auditable, controllable, and transparent before execution.
- **Host Authority:** The host environment has complete control over what operations are permitted.

6.6.3 JSL's Layered Security Model

JSL provides two complementary models for managing security and side effects.

Level 1: Dispatcher-Based Security (The Standard Model)

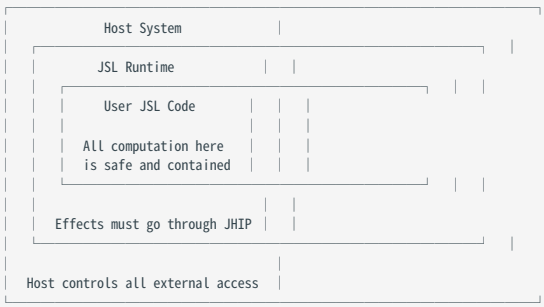
This is the most direct security model. The host system implements a **Host Command Dispatcher** that is the ultimate gatekeeper for all side effects. It is simple to understand and makes all side effects syntactically obvious.

Level 2: Capability-Based Security (The Advanced Model)

This is a more advanced model for high-security applications. It uses the **Environment Algebra** and closures to create sandboxes that can restrict access to the `["host", ...]` special form itself, providing a deeper layer of defense.

6.6.4 The Runtime Boundary

JSL code always executes within the confines of the JSL runtime, which is itself controlled by the host system. All interactions with the outside world must cross this boundary through the JHIP protocol, giving the host the final say.



6.6.5 Security Best Practices

For Host Implementations

1. ENFORCE STRICT RESOURCE LIMITS (THE "GAS" MODEL)

Untrusted code can easily attempt to cause a Denial of Service. The host runtime **must** enforce resource limits.

The most straightforward way to do this is to wrap every JSL evaluation in a **strict timeout**. This is the host's primary defense against infinite loops and other denial-of-service attacks.

For more granular control, a host can implement a **"gas" model**, similar to those used in blockchain systems.

- **How it works:** The host assigns a "gas cost" to every JSL operation (e.g., `+` costs 1 gas, `map` costs 5 gas). A program is started with a finite amount of gas, which is consumed on each operation. If the gas runs out, execution halts.
- **Benefits:** This provides a predictable execution cost that is independent of machine speed.

At a minimum, hosts should enforce simple timeouts and memory caps.

Design Note: Why Gas is a Host-Level Concern

A natural question is why a "gas" model is not a mandatory, built-in part of the JSL language specification. This is a deliberate architectural decision based on JSL's core design philosophy.

- **To Maximize Simplicity and Portability:** The primary goal of the JSL core is to be a simple, elegant, and easily embeddable evaluation engine. Forcing a complex gas accounting system into the language specification would dramatically increase the implementation burden. This would make it much harder to create compliant runtimes in different languages, undermining the goal of portability.
- **To Maintain Flexibility:** Different host environments have vastly different needs. A web server might manage resources via per-request timeouts, while a blockchain requires a strict, deterministic gas model. By defining gas as a host-level *best practice* rather than a language *requirement*, JSL remains flexible enough to be integrated naturally into any of these contexts without imposing a one-size-fits-all solution.

Keeping resource management at the host level preserves the simplicity of the core language while still providing a clear and robust pattern for building secure, production-ready systems.

2. IMPLEMENT THE PRINCIPLE OF LEAST CAPABILITY

When designing host commands, always expose the most specific, narrowly-scoped capability possible. Avoid creating general-purpose "escape hatches."

A good example is a specific, auditable capability for reading a config file:

```
["def", "config", ["host", "file/read", "/app/data/config.json"]]
```

A dangerous, overly broad capability would be:

```
["def", "config", ["host", "shell", "cat /app/data/config.json"]]
```

A specific command like `file/read` can be easily secured and audited by the host dispatcher. A `shell` command is a black box that subverts JSL's security model.

3. MAINTAIN DETAILED AUDIT LOGS

Because all side effects are reified as data, the host can create a perfect audit trail. Every `["host", ...]` request should be logged with a timestamp, the source of the code, the full request, and the outcome. This is invaluable for security analysis and incident response.

For JSL Code Developers

1. NEVER TRUST INPUT

Just as in any other language, you must treat all data coming from an external source as untrusted.

This example shows a function that expects a number for a calculation. It validates the input's type before using it.

```
["def", "calculate",
  ["lambda", ["input"],
    ["if", ["is_num", "input"],
      ["*", "input", 10],
      ["error", "InvalidInput", "Expected a number"]
    ]
  ]
]
```

Always validate the type and structure of data before using it in your logic.

2. HANDLE ERRORS GRACEFULLY

Host commands can fail for many reasons. Robust JSL code should anticipate these failures using the `try` special form.

This example attempts to read a configuration file, but returns a default value if it fails, logging the error as a warning.

```
["try",
  ["host", "file/read", "/app/config.json"],
  ["lambda", ["err"],
    ["do",
      ["host", "log/warn", ["@", "Config not found, using default: ", ["get", "err", "message"]]],
      { "default_setting": true }
    ]
  ]
]
```

This prevents unexpected host errors from crashing your entire program.

6.7 Network Transparency

6.7.1 Overview

Network transparency is one of JSL's defining characteristics - the ability to seamlessly transmit, store, and execute code across network boundaries with the same fidelity as local execution. This capability is fundamental to JSL's design and enables new patterns in distributed computing.

6.7.2 What is Network Transparency?

Network transparency means that code can be:

1. **Serialized** into a universal format (JSON)
2. **Transmitted** over any network transport
3. **Stored** in any JSON-compatible storage system
4. **Reconstructed** in a different runtime environment
5. **Executed** with identical behavior to the original

This creates a programming model where the physical location of code execution becomes an implementation detail rather than a fundamental constraint.

6.7.3 Technical Foundation

JSON as Universal Representation

JSL achieves network transparency by using JSON as the canonical representation for both code and data.

```
// This JSL function can be transmitted anywhere JSON is supported
["lambda", ["x"], ["*", "x", "x"]]
```

Advantages: - **Universal parsing**: Every major platform supports JSON - **Human readable**: Code can be inspected and understood - **Schema validation**: Structure can be verified - **Version stable**: JSON specification is stable and backward compatible

Closure Serialization

JSL's closure serialization ensures that functions retain their behavior across network boundaries.

```
// Original environment: x = 10
["lambda", ["y"], ["+", "x", "y"]]

// Serialized with captured environment:
{
  "type": "closure",
  "params": ["y"],
  "body": ["+", "x", "y"],
  "env": {"x": 10}
}
```

6.7.4 Network Transport Patterns

Request-Response Pattern

```
Client                    Server
  |                         |
  |------ JSL Function ---->|
  |                         |
  |<----- JSON Result ------|
  |                         |
```

Example of sending a computation to a server:

```
{
  "code": ["map", ["lambda", ["x"], ["*", "x", 2]], [1, 2, 3, 4]],
  "data": {}
}
```

And receiving the result:

```
{"result": [2, 4, 6, 8]}
```

Code Migration Pattern

```
Runtime A                Runtime B
  |                        |
  |-- Serialize Closure --->|
  |                        |
  |                        |-- Execute -->
  |                        |
  |<-- Return Result -------|
```

Distributed Pipeline Pattern

```
Data Source -> JSL Stage 1 -> JSL Stage 2 -> JSL Stage 3 -> Result
    |            |            |            |
    |            |            |            |
Network A     Network B     Network C     Network D
```

6.7.5 Storage Transparency

JSL code can be stored in any system that supports JSON:

Database Storage

```
-- Store JSL functions in database
CREATE TABLE jsl_functions (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255),
  code JSONB,
  created_at TIMESTAMP
);

-- Insert JSL function
INSERT INTO jsl_functions (name, code) VALUES (
  'square',
  '["lambda", ["x"], ["*", "x", "x"]]'::JSONB
);
```

File System Storage

```
// functions/math.jsl
{
  "square": ["lambda", ["x"], ["*", "x", "x"]],
  "cube": ["lambda", ["x"], ["*", "x", "x", "x"]],
  "factorial": ["lambda", ["n"],
    ["if", ["<=", "n", 1],
     1,
     ["*", "n", ["factorial", ["-", "n", 1]]]]]
}
```

Distributed Storage

```
// Configuration for distributed JSL library
{
  "repositories": [
    "https://jsllib.example.com/math",
    "https://jsllib.example.com/string",
    "https://jsllib.example.com/data"
  ],
  "cache": {
    "local": "/tmp/jsl-cache",
    "ttl": 3600
```

```
  }
}
```

6.7.6 Implementation Strategies

Eager Loading

```
// Load all dependencies upfront
{
  "main": ["do",
    ["use", "math/statistics"],
    ["mean", [1, 2, 3, 4, 5]]
  ],
  "dependencies": {
    "math/statistics": {
      "mean": ["lambda", ["xs"], ["quotient", ["sum", "xs"], ["length", "xs"]]],
      "sum": ["lambda", ["xs"], ["reduce", "+", 0, "xs"]]
    }
  }
}
```

Lazy Loading

```
// Load dependencies on demand
{
  "main": ["do",
    ["import", "https://jsllib.com/math/statistics.jsl"],
    ["mean", [1, 2, 3, 4, 5]]
  ]
}
```

Caching Strategies

```
// Cache configuration
{
  "cache_policy": {
    "strategy": "content_hash",
    "ttl": 86400,
    "max_size": "100MB",
    "locations": [
      "memory",
      "disk",
      "distributed"
    ]
  }
}
```

6.7.7 Network Protocols

HTTP Transport

```
POST /jsl/execute HTTP/1.1
Content-Type: application/json

{
  "code": ["host", "http/get", "https://api.example.com/data"],
  "timeout": 30000
}
```

WebSocket Transport

```
// Real-time JSL execution
{
  "type": "execute",
  "id": "req-123",
  "code": ["stream-map", ["lambda", ["x"], ["inc", "x"]], "input-stream"]
}
```

Message Queue Transport

```
// Queue: jsl-tasks
{
  "task_id": "task-456",
  "code": ["batch-process", "data-batch-1"],
```

```
  "priority": "high",
  "retry_count": 3
}
```

6.7.8 Performance Considerations

Bandwidth Optimization

1. **Code Compression**
2. JSON compression (gzip, brotli)
3. Code minification (remove whitespace)
4. Delta compression (send only changes)
5. **Caching**
6. Function memoization
7. Code artifact caching
8. Network-level caching
9. **Batching**
10. Multiple operations in single request
11. Pipeline optimization
12. Bulk data transfer

Latency Optimization

1. **Preloading**
2. Predictive code loading
3. Warm caches
4. Connection pooling
5. **Locality**
6. Edge computing deployment
7. Regional code distribution
8. Data locality optimization

6.7.9 Security Considerations

Transport Security

- **Encryption**: TLS for all network transport
- **Authentication**: Verify code sources
- **Integrity**: Hash verification of transmitted code

Code Validation

- **Schema validation**: Verify JSON structure
- **Security scanning**: Detect malicious patterns
- **Resource limits**: Prevent resource exhaustion

Access Control

- **Code signing**: Cryptographic verification
- **Capability restrictions**: Limit available operations

- **Audit logging**: Track all code execution

6.7.10 Use Cases

Distributed Computing

```
// Send computation to data location
{
  "target": "data-center-eu",
  "code": ["analyze-user-behavior", "european-users"],
  "resources": {"cpu": "4-cores", "memory": "8GB"}
}
```

Edge Computing

```
// Deploy logic to edge devices
{
  "targets": ["edge-device-*"],
  "code": ["if", ["sensor-reading", ">", 100],
    ["alert", "temperature-high"],
    null
  ]
}
```

Database Functions

```
-- Execute JSL directly in database
SELECT jsl_execute('["group-by", "status", "orders"]', orders_table);
```

Microservice Communication

```
// Service A requests computation from Service B
{
  "service": "analytics-service",
  "function": ["lambda", ["data"], ["statistical-summary", "data"]],
  "data": {...}
}
```

Network transparency fundamentally changes how we think about distributed computing, making code mobility as natural as data mobility and enabling new architectures that were previously impractical or impossible.

6.8 Code and Data Serialization

For a conceptual overview of JSL environments and the operations you can perform on them, see the Environments language guide.

6.8.1 Overview

Serialization is fundamental to JSL's design philosophy. Since JSL code is JSON, any JSL program is already in a transmittable format. However, to capture the full, executable state of a program—including functions with captured lexical environments (closures)—JSL uses a content-addressable serialization format.

This structure enables true code mobility, persistence, and robust distributed computing patterns while handling circular references elegantly.

6.8.2 Content-Addressable Serialization Format

JSL uses a content-addressable serialization format where each unique object gets a deterministic hash based on its content. When a JSL value is serialized, the payload includes three key parts:

1. `__cas_version__` : Version number of the content-addressable serialization format (currently 1).
2. `objects` : A key-value map where each key is the content-hash of a complex object (closure or environment), and the value is the object itself. This is the "object store."
3. `root` : The root value of the serialization. If it's a complex object, it will reference an object by its hash using `{"__ref__": "hash"}` .

Complex Object Structure

Environment Objects represent scopes and have the structure: - `__type__` : Always `"env"` to identify the object type - `bindings` : A JSON object mapping variable names to their values (which may include references to other objects)

Closure Objects represent functions and have the structure: - `__type__` : Always `"closure"` to identify the object type - `params` : List of parameter names - `body` : The function body as a JSL expression - `env` : Reference to the captured environment (may be `{"__ref__": "hash"}`)

The hash of each object is calculated over its content, creating a stable identifier. Objects that reference each other use the `{"__ref__": "hash"}` format, which allows the deserializer to reconstruct the object graph correctly.

Serialization Walkthrough

Consider this simple closure example:

```
["do",
  ["def", "base", 100],
  ["lambda", ["x"], ["+", "x", "base"]]
]
```

When evaluated, this produces a closure that captures the `base` variable. The serialized payload might look like this:

```
{
  "__cas_version__": 1,
  "root": {"__ref__": "a1b2c3d4e5f6g7h8"},
  "objects": {
    "a1b2c3d4e5f6g7h8": {
      "__type__": "closure",
      "params": ["x"],
      "body": ["+", "x", "base"],
      "env": {"__ref__": "h8g7f6e5d4c3b2a1"}
    },
    "h8g7f6e5d4c3b2a1": {
      "__type__": "env",
      "bindings": {
        "base": 100
      }
    }
  }
}
```

For primitive values (numbers, strings, booleans, simple arrays/objects), the serialization is direct JSON without the CAS wrapper:

```
// Simple values serialize directly
42              → "42"
[1, 2, 3]       → "[1,2,3]"
{"key": "val"} → "{\"key\":\"val\"}"
```

This content-addressable model efficiently handles circular references, avoids data duplication, and correctly represents complex object relationships in a purely serializable way.

6.8.3 Serialization Patterns

The following patterns demonstrate how the canonical serialized state payload can be used.

Network Transmission

The serialized payload can be sent directly as the body of an HTTP request to a remote JSL node for execution.

```
POST /execute HTTP/1.1
Content-Type: application/json

{
  "__cas_version__": 1,
  "root": {"__ref__": "a1b2c3d4e5f6g7h8"},
  "objects": {
    "a1b2c3d4e5f6g7h8": {
      "__type__": "closure",
      "params": ["x"],
      "body": ["+", "x", "base"],
      "env": {"__ref__": "h8g7f6e5d4c3b2a1"}
    },
    "h8g7f6e5d4c3b2a1": {
      "__type__": "env",
      "bindings": {"base": 100}
    }
  }
}
```

Code Storage

The entire payload can be stored in a database (e.g., in a JSONB column) to persist a function, a user's workflow state, or a configuration template.

```
INSERT INTO jsl_functions (name, payload) VALUES (
  'increment_function',
  '{
    "__cas_version__": 1,
    "root": {"__ref__": "a1b2c3d4e5f6g7h8"},
    "objects": {
      "a1b2c3d4e5f6g7h8": {
        "__type__": "closure",
        "params": ["x"],
        "body": ["+", "x", "base"],
        "env": {"__ref__": "h8g7f6e5d4c3b2a1"}
      },
      "h8g7f6e5d4c3b2a1": {
        "__type__": "env",
        "bindings": {"base": 100}
      }
    }
  }'::JSONB
);
```

Object Serialization

JSON objects with embedded code serialize cleanly:

```
{
  "response_object": {
    "@status": "@success",
    "@user": "username",
    "@computed_score": ["*", "base_score", "multiplier"],
    "@timestamp": ["host", "time/now"]
  }
}
```

When evaluated, this produces a pure JSON object with all expressions resolved.

6.8.4 Conclusion

JSL's serialization model is designed for simplicity, efficiency, and portability. By leveraging JSON as the underlying format, JSL ensures that code can be easily transmitted, stored, and executed across different environments without loss of fidelity or context.

6.9 Distributed Computing with JSL

6.9.1 Overview

JSL's core design—being homoiconic and having a robust, verifiable serialization model—makes it an ideal language for building distributed systems. Because both code and state can be safely transmitted over the network, complex distributed patterns can be expressed with the same clarity as local computations.

An Architectural Showcase: The following examples are an architectural showcase of what is possible. They are not a standard library reference. These patterns assume the host environment provides a rich set of networking primitives (e.g., `remote/execute`, `remote/call`). The purpose is to demonstrate how JSL can be used as the foundation for a powerful distributed computing framework.

6.9.2 Core Patterns

1. Remote Execution

The most fundamental pattern is executing a function on a remote node. JSL's serializable closures make this trivial. The closure packages its code and its environment, which can be sent to a remote host for evaluation.

The host provides `remote/execute` which takes a node, a function, and arguments.

```
["host", "remote/execute", "node-2",
  ["lambda", ["x"], ["*", "x", "x"]],
  5]
```

2. Master-Worker Pattern

A coordinator node can partition a workload and distribute it among a set of worker nodes. This pattern highlights how JSL's functional nature (`map`, `zip`) simplifies parallel processing logic. The worker function itself is passed as an argument, making this a flexible, higher-order function.

```
["def", "distribute_work",
  ["lambda", ["work_items", "workers", "work_fn"],
    ["do",
      ["def", "chunks", ["partition", "work_items", ["length", "workers"]]],
      ["def", "tasks", ["zip", "workers", "chunks"]],
      ["map",
        ["lambda", ["task"],
          ["host", "remote/execute",
            ["first", "task"],
            "work_fn",
            ["second", "task"]]],
        "tasks"]]]]
```

3. Fault Tolerance via Retries

Handling network failures is critical. Because JSL code is data, we can easily write higher-order functions that wrap any remote call with a retry mechanism.

This example defines a recursive inner function, `try_call`, to handle the retry loop. The `try` special form is used to catch failures, and the error handler recursively calls itself with one fewer attempt.

```
["def", "reliable_remote_call",
  ["lambda", ["node", "function", "args", "max_retries"],
    ["do",
      ["def", "try_call",
        ["lambda", ["attempts_left"],
          ["if", ["=", "attempts_left", 0],
            ["error", "Max retries exceeded"],
            ["try",
              ["host", "remote/call", "node", "function", "args"],
              ["lambda", ["error"],
                ["do",
                  ["host", "log/warn", ["@", "Retry attempt ", ["-", ["+", "max_retries", 1], "attempts_left"], " failed. Retrying..."]],
                  ["try_call", ["-", "attempts_left", 1]]
                ]
              ]
            ]
```

```
          ]
        ]
      ],
      ["try_call", "max_retries"]
    ]
  ]
]
```

6.9.3 Advanced Patterns

JSL's composability allows these simple building blocks to be combined into sophisticated distributed algorithms.

MapReduce Implementation

This example shows how a full MapReduce job can be expressed by composing the `distribute_work` function defined earlier.

The process is broken down into three phases: 1. **MAP PHASE:** Distribute the map function across the map nodes. 2. **SHUFFLE PHASE:** Group the intermediate results by key. 3. **REDUCE PHASE:** Distribute the reduce function across the reduce nodes to produce the final result.

```
["def", "mapreduce",
  ["lambda", ["map_fn", "reduce_fn", "data", "map_nodes", "reduce_nodes"],
    ["do",
      ["def", "map_results", ["distribute_work", "data", "map_nodes", "map_fn"]],
      ["def", "grouped", ["group_by", "first", ["flatten", "map_results"]]],
      ["def", "reduce_tasks", ["items", "grouped"]],
      ["distribute_work", "reduce_tasks", "reduce_nodes", "reduce_fn"]
    ]
  ]
]
```

These examples illustrate that JSL provides the ideal substrate for building resilient, scalable systems while maintaining the simplicity and clarity of the language's core design principles.

7. Examples

7.1 Simple JSL Examples

7.1.1 Basic Arithmetic

```
// Addition
["+", 5, 3]
// Result: 8

// Multiplication
["*", 4, 6]
// Result: 24

// Nested operations
["+", ["*", 2, 3], ["*", 4, 5]]
// Result: 26
```

7.1.2 Variables and Functions

```
// Define and use variables with let
["let", [
  ["x", 10],
  ["y", 20]
],
  ["+", "x", "y"]
]
// Result: 30

// Define and use a function
["let", [
  ["double", ["lambda", ["n"], ["*", "n", 2]]]
],
  ["double", 5]
]
// Result: 10
```

7.1.3 Working with Lists

```
// Creating lists with quote
["let", [
  ["numbers", ["@", [1, 2, 3, 4, 5]]],
  ["double", ["lambda", ["x"], ["*", "x", 2]]]
],
  ["map", "double", "numbers"]
]
// Result: [2, 4, 6, 8, 10]

// Creating lists with the list function
["let", [
  ["numbers", ["list", 1, 2, 3, 4, 5]],
  ["is-even", ["lambda", ["x"], ["=", ["mod", "x", 2], 0]]]
],
  ["filter", "is-even", "numbers"]
]
// Result: [2, 4]
```

7.1.4 Conditional Logic

```
// Simple if expression
["if", [">", 10, 5], "@yes", "@no"]
// Result: "yes"

// Function with conditional logic
["let", [
  ["check-age",
    ["lambda", ["age"],
      ["if", [">=", "age", 18], "@adult", "@minor"]]]
],
  ["list",
    ["check-age", 25],  // Result: "adult"
    ["check-age", 15]   // Result: "minor"
  ]
]
```

7.1.5 Working with Objects

```
// Create an object with computed values
["let", [
  ["name", "@Alice"],
  ["age", 30]
],
  {"@user": "name",
   "@adult": [">=", "age", 18]}
]
// Result: {"user": "Alice", "adult": true}

// Access object properties
["let", [
  ["person", {"@name": "@Bob", "@age": 25}]
],
  ["get", "person", "@name"]
]
// Result: "Bob"
```

7.2 Practical JSL Examples

These examples demonstrate core JSL patterns for real-world programming tasks.

7.2.1 Data Processing

Filtering and Transforming Lists

Extract email addresses from active users:

```
["let", [
  ["users", ["@", [
    {"@name": "@Alice", "@email": "@alice@example.com", "@active": true},
    {"@name": "@Bob", "@email": "@bob@example.com", "@active": false},
    {"@name": "@Charlie", "@email": "@charlie@example.com", "@active": true}
  ]]],
  ["active-emails",
    ["map",
      ["lambda", ["user"], ["get", "user", "@email"]],
      ["filter",
        ["lambda", ["user"], ["get", "user", "@active"]],
        "users"]]]
],
  "active-emails"
]
```

Result: `["alice@example.com", "charlie@example.com"]`

7.2.2 File Operations

Reading and Processing a File

Read a configuration file and parse it:

```
["let", [
  ["config-path", "@/app/config.json"],
  ["config", ["try",
    ["json-parse", ["host", "file/read", "config-path"]],
    ["lambda", ["err"],
      {"@debug": false, "@timeout": 30}]]]
],
  ["get", "config", "@debug", false]
]
```

This safely reads a JSON config file with a fallback default if the file doesn't exist.

7.2.3 Dynamic Configuration

Environment-Based Settings

Build configuration that adapts to the current environment:

```
["let", [
  ["env", "@production"],
  ["is-prod", ["=", "env", "@production"]]
],
  {
    "@app": "@my-service",
    "@database": {
      "@host": ["if", "is-prod", "@db.prod.example.com", "@localhost"],
      "@port": 5432,
      "@pool_size": ["if", "is-prod", 20, 5]
    },
    "@logging": {
      "@level": ["if", "is-prod", "@warn", "@debug"],
      "@format": "@json"
    }
  }
]
```

7.2.4 API Request with Error Handling

Safe HTTP Request

Fetch user data with proper error handling:

```
["let", [
  ["fetch-user",
    ["lambda", ["id"],
      ["try",
        ["host", "http/get", ["str-concat", "@/api/users/", "id"]],
        ["lambda", ["err"],
          {"@error": true, "@message": ["get", "err", "@message"]}]]]]
],
  ["fetch-user", "@12345"]
]
```

7.2.5 Key Patterns Demonstrated

1. `let` **for functional bindings** - Clean variable scoping without mutation
2. **First-class objects** - JSON objects as native data structures
3. **Error handling with** `try` - Graceful failure recovery
4. **Higher-order functions** - `map` and `filter` for data transformation
5. **Dynamic values** - Computing object properties with expressions

7.3 Advanced JSL Examples

7.3.1 Distributed Computing

This example demonstrates a map-reduce implementation for word counting.

```
["let", [
  ["map-reduce",
    ["lambda", ["data", "map-fn", "reduce-fn"],
      ["let", [["mapped", ["map", "map-fn", "data"]]],
        ["reduce", "reduce-fn", {}, "mapped"]]]],

  ["word-count",
    ["lambda", ["text"],
      ["let", [
        ["words", ["str-split", ["str-lower", "text"], "@ "]],
        ["count-word",
          ["lambda", ["acc", "word"],
            ["set", "acc", "word",
              ["+", ["get", "acc", "word", 0], 1]]]]
      ],
        ["reduce", "count-word", {}, "words"]]]]
],
  ["word-count", "@The quick brown fox jumps over the lazy dog"]
]
```

7.3.2 Closure Serialization

This example shows how a closure can be serialized to JSON, then deserialized and executed.

```
["let", [
  ["create-adder",
    ["lambda", ["n"],
      ["lambda", ["x"], ["+", "x", "n"]]]],

  ["add-five", ["create-adder", 5]],
  ["serialized", ["serialize", "add-five"]],
  ["restored", ["deserialize", "serialized"]]
],
  ["restored", 10]
]
```

Result: `15` - The closure maintains its captured environment even after serialization.

7.3.3 Dynamic Configuration Objects

Build configuration objects that adapt based on environment settings.

```
["let", [
  ["env", "@production"],
  ["is-prod", ["=", "env", "@production"]],
  ["db-host", ["if", "is-prod", "@db.prod.example.com", "@localhost"]]
],
  {
    "@database": {
      "@host": "db-host",
      "@port": ["if", "is-prod", 5432, 5433],
      "@name": ["str-concat", "@myapp_", "env"],
      "@ssl": "is-prod",
      "@pool_size": ["if", "is-prod", 20, 5]
    },
    "@services": {
      "@auth": {
        "@url": ["str-concat", "@https://auth.", "env", "@.example.com"],
        "@timeout": ["if", "is-prod", 5000, 30000]
      },
      "@cache": {
        "@enabled": "is-prod",
        "@ttl": ["if", "is-prod", 3600, 300]
      }
    }
  }
]
```

7.3.4 Memoization Pattern

Create a memoized version of expensive computations.

```
["let", [
  ["memoize",
    ["lambda", ["fn"],
      ["let", [["cache", {}]],
        ["lambda", ["arg"],
          ["let", [["key", ["str", "arg"]]],
            ["if", ["has", "cache", "key"],
              ["get", "cache", "key"],
              ["let", [["result", ["fn", "arg"]]],
                ["do",
                  ["set", "cache", "key", "result"],
                  "result"]]]]]]]],

  ["fibonacci",
    ["lambda", ["n"],
      ["if", ["<=", "n", 1],
        "n",
        ["+",
          ["fibonacci", ["-", "n", 1]],
          ["fibonacci", ["-", "n", 2]]]]]],

  ["fast-fib", ["memoize", "fibonacci"]]
],
  ["map", "fast-fib", ["@", [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]]
]
```

7.3.5 Pipeline Processing

Build a data processing pipeline with error handling.

```
["let", [
  ["pipeline",
    ["lambda", ["data", "transformations"],
      ["reduce",
        ["lambda", ["acc", "transform"],
          ["try",
            ["transform", "acc"],
            ["lambda", ["err"],
              {"@error": true,
               "@message": ["get", "err", "@message"],
               "@data": "acc"}]]],
        "data",
        "transformations"]]],

  ["validate-user",
    ["lambda", ["user"],
      ["if", ["and",
              ["has", "user", "@email"],
              ["str-contains", ["get", "user", "@email"], "@@"]],
        "user",
        ["error", "@Invalid email"]]]],

  ["enrich-user",
    ["lambda", ["user"],
      ["set", "user", "@id",
        ["str-concat", "@user_", ["str", ["random-int", 1000, 9999]]]]]],

  ["user-data", {"@name": "@Alice", "@email": "@alice@example.com"}]
],
  ["pipeline", "user-data", ["@", ["validate-user", "enrich-user"]]]
]
```

7.3.6 Key Advanced Patterns

1. **Map-Reduce** - Process large datasets in a functional way
2. **Closure Serialization** - Send functions with their context over the network
3. **Dynamic Objects** - Build configuration that adapts to runtime conditions
4. **Memoization** - Cache expensive computations transparently
5. **Pipeline Processing** - Chain transformations with error handling

8. API Reference

8.1 Core Module API Reference

The core module (`jsl.core`) provides the fundamental data structures, evaluator, and environment management for JSL.

8.1.1 Overview

The `jsl.core` module provides the fundamental data structures that represent the state of a JSL program: `Env` for environments and `Closure` for functions. These are the building blocks used by the **Evaluator** and managed by the **JSLRunner**.

8.1.2 Classes

`Env`

The environment class manages variable bindings and scope chains.

8.1.3 `jsl.core.Env(bindings=None, parent=None)`

Represents a JSL environment - a scope containing variable bindings.

Environments form a chain: each environment has an optional parent. When looking up a variable, we search the current environment first, then its parent, and so on until we find it or reach the root.

Source code in `jsl/core.py`

```
69    def __init__(self, bindings: Optional[Dict[str, Any]] = None, parent: Optional['Env'] = None):
70        self.bindings = bindings or {}
71        self.parent = parent
```

`content_hash()`

Generate a content-addressable hash with cycle detection.

Source code in jsl/core.py

```
def content_hash(self) -> str:
    """Generate a content-addressable hash with cycle detection."""
    # Thread-local cycle detection for rock-solid safety
    import threading
    if not hasattr(Env, '_cycle_detection'):
        Env._cycle_detection = threading.local()

    if not hasattr(Env._cycle_detection, 'computing'):
        Env._cycle_detection.computing = set()

    env_id = id(self)
    if env_id in Env._cycle_detection.computing:
        # Cycle detected - return deterministic placeholder
        return f"cycle_{env_id:016x}"

    # Add to cycle detection set
    Env._cycle_detection.computing.add(env_id)

    try:
        canonical = {
            "bindings": self._serialize_bindings(),
            "parent_hash": self.parent.content_hash() if self.parent else None
        }
        # Convert to string - handle special cases
        try:
            content = json.dumps(canonical, sort_keys=True)
        except (TypeError, ValueError):
            # If we can't serialize (due to complex objects), use a fallback
            # This can happen when bindings contain data structures with Closures
            content = str(sorted(canonical.get("bindings", {}).keys())) + str(canonical.get("parent_hash"))
        return hashlib.sha256(content.encode()).hexdigest()[:16]

    finally:
        # Always clean up, even on exceptions
        Env._cycle_detection.computing.discard(env_id)

        # Clean up thread-local storage when empty
        if not Env._cycle_detection.computing:
            delattr(Env._cycle_detection, 'computing')
```

`define(name, value)`

Define a variable in this environment.

Source code in jsl/core.py

```
def define(self, name: str, value: Any) -> None:
    """Define a variable in this environment."""
    self.bindings[name] = value
```

`extend(new_bindings)`

Create a new environment that extends this one with additional bindings.

Source code in jsl/core.py

```
def extend(self, new_bindings: Dict[str, Any]) -> 'Env':
    """Create a new environment that extends this one with additional bindings."""
    return Env(new_bindings, parent=self)
```

`get(name)`

Look up a variable in this environment or its parents.

Source code in `jsl/core.py`

```
73  def get(self, name: str) -> Any:
74      """Look up a variable in this environment or its parents."""
75      if name in self.bindings:
76          return self.bindings[name]
77      elif self.parent:
78          return self.parent.get(name)
79      else:
80          raise SymbolNotFoundError(f"Symbol '{name}' not found")
```

`to_dict()`

Convert environment bindings to a dictionary (for serialization).

Source code in `jsl/core.py`

```
90  def to_dict(self) -> Dict[str, Any]:
91      """Convert environment bindings to a dictionary (for serialization)."""
92      result = {}
93      if self.parent:
94          result.update(self.parent.to_dict())
95      result.update(self.bindings)
96      return result
```

KEY CONCEPTS

- **Scope Chain:** When looking up a variable, if it's not found in the current `Env`, the search continues up to its parent, and so on, until the root `prelude` is reached.
- **Immutability:** Methods like `extend` create a *new* child environment rather than modifying the parent, preserving functional purity.

```
from jsl.core import Env

# Create a new environment
env = Env({"x": 10, "y": 20})

# Create a child environment that inherits from the parent
child_env = env.extend({"z": 30})

# Variable resolution follows the chain
print(child_env.get("x"))  # 10 (from parent)
print(child_env.get("z"))  # 30 (from child)
```

`Closure`

Represents a user-defined function with captured lexical environment.

8.1.4 `jsl.core.Closure(params, body, env)` `dataclass`

Represents a JSL function (closure).

A closure captures three things: 1. The parameter names it expects 2. The body expression to evaluate when called 3. The environment where it was defined (lexical scoping)

`__call__(evaluator, args)`

Apply this closure to the given arguments.

Source code in jsl/core.py

```
def __call__(self, evaluator: 'Evaluator', args: List[JSLValue]) -> JSLValue:
    """Apply this closure to the given arguments."""
    if len(args) != len(self.params):
        raise JSLTypeError(f"Function expects {len(self.params)} arguments, got {len(args)}")

    # Create new environment extending the closure's captured environment
    call_env = self.env.extend(dict(zip(self.params, args)))
    return evaluator.eval(self.body, call_env)
```

Closures capture their defining environment:

```
from jsl.core import Closure, Env

# Create an environment that the closure will capture
env = Env({"multiplier": 3})

# Create a closure that captures the 'multiplier' variable from its environment
closure = Closure(
    params=["x"],
    body=["*", "multiplier", "x"],
    env=env
)

# The closure remembers the 'multiplier' value
```

Evaluator

The main JSL expression evaluator:

8.1.5 jsl.core.Evaluator(host_dispatcher=None, resource_limits=None, host_gas_policy=None)

The core JSL evaluator - recursive evaluation engine.

This is a clean, elegant reference implementation that uses traditional recursive tree-walking to evaluate S-expressions. It serves as the specification for JSL's semantics.

Characteristics: - Simple and easy to understand - Direct mapping from S-expressions to evaluation - Perfect for learning and testing JSL semantics - Limited by Python's recursion depth for deep expressions

For production use with resumption and better performance, use the stack-based evaluator which compiles to JPN (JSL Postfix Notation).

Source code in jsl/core.py

```
def __init__(self, host_dispatcher: Optional[HostDispatcher] = None,
             resource_limits: Optional[ResourceLimits] = None,
             host_gas_policy: Optional['HostGasPolicy'] = None):
    self.host = host_dispatcher or HostDispatcher()
    self.resources = ResourceBudget(resource_limits, host_gas_policy) if resource_limits else None
```

eval(expr, env)

Evaluate a JSL expression in the given environment.

This is a pure recursive evaluator without resumption support. For resumable evaluation, use the stack-based evaluator.

Source code in `jsl/core.py`

```
def eval(self, expr: JSLExpression, env: Env) -> JSLValue:
    """
    Evaluate a JSL expression in the given environment.

    This is a pure recursive evaluator without resumption support.
    For resumable evaluation, use the stack-based evaluator.
    """
    # Resource checking
    if self.resources:
        # Check time periodically
        self.resources.check_time()

        # Consume gas based on expression type
        if isinstance(expr, (int, float, bool)) or expr is None:
            self.resources.consume_gas(GasCost.LITERAL)
        elif isinstance(expr, str):
            if expr.startswith("@"):
                self.resources.consume_gas(GasCost.LITERAL)
            else:
                self.resources.consume_gas(GasCost.VARIABLE)
        elif isinstance(expr, dict):
            self.resources.consume_gas(GasCost.DICT_CREATE +
                                       len(expr) * GasCost.DICT_PER_ITEM)

    # Literals: numbers, booleans, null, objects
    if isinstance(expr, (int, float, bool)) or expr is None:
        return expr

    # Objects: evaluate both keys and values, keys must be strings
    if isinstance(expr, dict):
        return self._eval_dict(expr, env)

    # Strings: variables or string literals
    if isinstance(expr, str):
        return self._eval_string(expr, env)

    # Arrays: function calls or special forms
    if isinstance(expr, list):
        return self._eval_list(expr, env)

    raise JSLTypeError(f"Cannot evaluate expression of type {type(expr)}")
```

`HostDispatcher`

Manages host interactions for side effects:

8.1.6 `jsl.core.HostDispatcher()`

Handles JHIP (JSL Host Interaction Protocol) requests.

This is where all side effects are controlled. The host environment registers handlers for specific commands and decides what operations are permitted.

Source code in `jsl/core.py`

```
def __init__(self):
    self.handlers: Dict[str, Callable] = {}
```

`dispatch(command, args)`

Dispatch a host command with arguments.

Source code in `jsl/core.py`

```
def dispatch(self, command: str, args: List[Any]) -> Any:
    """Dispatch a host command with arguments."""
    if command not in self.handlers:
        raise JSLError(f"Unknown host command: {command}")

    try:
        return self.handlers[command](*args)
    except Exception as e:
        raise JSLError(f"Host command '{command}' failed: {e}")
```

`register(command, handler)`

Register a handler for a specific host command.

Source code in `jsl/core.py`

```
def register(self, command: str, handler: Callable) -> None:
    """Register a handler for a specific host command."""
    self.handlers[command] = handler
```

8.1.7 Resource Management

`ResourceBudget`

8.1.8 `jsl.core.ResourceBudget(limits=None, host_gas_policy=None)`

Comprehensive resource tracking for secure JSL execution.

Tracks gas consumption, memory allocation, execution time, and stack depth to prevent DOS attacks and ensure fair resource allocation.

Initialize resource budget.

Parameters:

Name	Type	Description	Default
`limits`	`Optional[ResourceLimits]`	Resource limits configuration	`None`
`host_gas_policy`	`Optional[HostGasPolicy]`	Gas cost policy for host operations	`None`

`allocate_memory(bytes_count, description='')`

Account for memory allocation.

Parameters:

Name	Type	Description	Default
`bytes_count`	`int`	Number of bytes to allocate	*required*
`description`	`str`	Description of allocation	`''`

Raises:

Type	Description
`MemoryExhausted`	If memory limit would be exceeded

`check_collection_size(size)`

Check if a collection size is within limits.

Parameters:

Name	Type	Description	Default
`size`	`int`	Size of the collection	*required*

Raises:

Type	Description
`MemoryExhausted`	If collection size exceeds limit

`check_result(result)`

Check resource constraints for a computed result.

This centralizes checking for collection sizes, string lengths, etc.

Parameters:

Name	Type	Description	Default
`result`	`Any`	The result value to check	*required*

`check_string_length(length)`

Check if a string length is within limits.

Parameters:

Name	Type	Description	Default
`length`	`int`	Length of the string	*required*

Raises:

Type	Description
`MemoryExhausted`	If string length exceeds limit

`check_time()`

Check if time limit has been exceeded.

Raises:

Type	Description
`TimeExhausted`	If time limit has been exceeded

`checkpoint()`

Create a checkpoint of current resource usage.

Returns:

Type	Description
`Dict[str, Any]`	Dictionary with current resource usage

`consume_gas(amount, operation='')`

Consume gas for an operation.

Parameters:

Name	Type	Description	Default
`amount`	`int`	Gas amount to consume	*required*
`operation`	`str`	Description of operation (for error messages)	`''`

Raises:

Type	Description
`GasExhausted`	If gas limit would be exceeded

`consume_host_gas(operation)`

Consume gas for a host operation based on namespace.

Parameters:

Name	Type	Description	Default
`operation`	`str`	Host operation path (e.g., "@file/read")	*required*

`enter_call()`

Enter a function call (increase stack depth).

Raises:

Type	Description
`StackOverflow`	If stack depth limit would be exceeded

`exit_call()`

Exit a function call (decrease stack depth).

`restore(checkpoint)`

Restore resource usage from a checkpoint.

Parameters:

Name	Type	Description	Default
`checkpoint`	`Dict[str, Any]`	Previously saved checkpoint	*required*

`ResourceLimits`

8.1.9

`jsl.core.ResourceLimits(max_gas=None, max_memory=None, max_time_ms=None, max_stack_depth=100, max_collection_size=10000, max_string_length=100000)`

Configuration for resource limits.

GasCost

8.1.10 jsl.core.GasCost

Bases: IntEnum

Gas costs for different operation types.

8.1.11 Error Types

JSLError

8.1.12 jsl.core.JSLError

Bases: Exception

Base exception for all JSL runtime errors.

SymbolNotFoundError

8.1.13 jsl.core.SymbolNotFoundError

Bases: JSLError

Raised when a symbol cannot be found in the current environment.

JSLTypeError

8.1.14 jsl.core.JSLTypeError

Bases: JSLError

Raised when there's a type mismatch in JSL operations.

8.1.15 Global State

prelude

The global prelude environment containing all built-in functions. A global, read-only instance of `Env` that contains all the JSL built-in functions (e.g., `+`, `map`, `get`). It serves as the ultimate parent of all other environments.

```
from jsl.core import prelude

# Access built-in functions
plus_func = prelude.get("+")
map_func = prelude.get("map")
```

8.1.16 Implementation Details

Environment Chains

JSL uses environment chains for variable resolution:

1. **Current Environment**: Look for variable in current scope
2. **Parent Environment**: If not found, check parent scope
3. **Continue Chain**: Repeat until variable found or chain ends
4. **Prelude Access**: All chains eventually reach the global prelude

Closure Serialization

Closures are designed for safe serialization:

- **Parameters**: Always serializable (list of strings)
- **Body**: Always serializable (JSON expression)
- **Environment**: Only user-defined bindings are serialized
- **Prelude**: Built-in functions are reconstructed, not serialized

This ensures transmitted closures are safe and can be reconstructed in any compatible JSL runtime.

8.1.17 Type Definitions

The module defines the following type aliases for clarity:

```
from typing import Union, List, Dict, Any

JSLValue = Union[None, bool, int, float, str, List[Any], Dict[str, Any], Closure]
JSLExpression = Union[JSLValue, List[Any], Dict[str, Any]]
```

8.1.18 Usage Examples

Basic Evaluation

```
from jsl.core import Evaluator, Env
from jsl.prelude import make_prelude

# Create evaluator and environment
evaluator = Evaluator()
env = make_prelude()

# Evaluate an expression
result = evaluator.eval(["+", 1, 2, 3], env)
print(result)  # Output: 6
```

Working with Closures

```
from jsl.core import Evaluator, Env
from jsl.prelude import make_prelude

evaluator = Evaluator()
env = make_prelude()

# Define a function
evaluator.eval(["def", "square", ["lambda", ["x"], ["*", "x", "x"]]], env)

# Call the function
result = evaluator.eval(["square", 5], env)
print(result)  # Output: 25
```

Resource-Limited Execution

```
from jsl.core import Evaluator, ResourceBudget, ResourceLimits

# Create evaluator with resource limits
limits = ResourceLimits(max_steps=1000, max_gas=10000)
budget = ResourceBudget(limits=limits)
evaluator = Evaluator(resource_budget=budget)

# Execute with resource tracking
result = evaluator.eval(expensive_computation, env)
print(f"Gas used: {budget.gas_used}")
print(f"Steps taken: {budget.steps_taken}")
```

8.2 Evaluator API Reference

The core module (`jsl.core`) contains the main evaluation engine for JSL expressions.

8.2.1 Main Evaluator Class

Evaluator

The main evaluator class for JSL expressions.

8.2.2 `jsl.core.Evaluator(host_dispatcher=None, resource_limits=None, host_gas_policy=None)`

The core JSL evaluator - recursive evaluation engine.

This is a clean, elegant reference implementation that uses traditional recursive tree-walking to evaluate S-expressions. It serves as the specification for JSL's semantics.

Characteristics: - Simple and easy to understand - Direct mapping from S-expressions to evaluation - Perfect for learning and testing JSL semantics - Limited by Python's recursion depth for deep expressions

For production use with resumption and better performance, use the stack-based evaluator which compiles to JPN (JSL Postfix Notation).

Source code in `jsl/core.py`

```
218    def __init__(self, host_dispatcher: Optional[HostDispatcher] = None,
219                 resource_limits: Optional[ResourceLimits] = None,
220                 host_gas_policy: Optional['HostGasPolicy'] = None):
221        self.host = host_dispatcher or HostDispatcher()
222        self.resources = ResourceBudget(resource_limits, host_gas_policy) if resource_limits else None
```

`eval(expr, env)`

Evaluate a JSL expression in the given environment.

This is a pure recursive evaluator without resumption support. For resumable evaluation, use the stack-based evaluator.

Source code in `jsl/core.py`

```
    def eval(self, expr: JSLExpression, env: Env) -> JSLValue:
        """
        Evaluate a JSL expression in the given environment.

        This is a pure recursive evaluator without resumption support.
        For resumable evaluation, use the stack-based evaluator.
        """
        # Resource checking
        if self.resources:
            # Check time periodically
            self.resources.check_time()

            # Consume gas based on expression type
            if isinstance(expr, (int, float, bool)) or expr is None:
                self.resources.consume_gas(GasCost.LITERAL)
            elif isinstance(expr, str):
                if expr.startswith("@"):
                    self.resources.consume_gas(GasCost.LITERAL)
                else:
                    self.resources.consume_gas(GasCost.VARIABLE)
            elif isinstance(expr, dict):
                self.resources.consume_gas(GasCost.DICT_CREATE +
                                           len(expr) * GasCost.DICT_PER_ITEM)

        # Literals: numbers, booleans, null, objects
        if isinstance(expr, (int, float, bool)) or expr is None:
            return expr

        # Objects: evaluate both keys and values, keys must be strings
        if isinstance(expr, dict):
            return self._eval_dict(expr, env)

        # Strings: variables or string literals
        if isinstance(expr, str):
            return self._eval_string(expr, env)

        # Arrays: function calls or special forms
        if isinstance(expr, list):
            return self._eval_list(expr, env)

        raise JSLTypeError(f"Cannot evaluate expression of type {type(expr)}")
```

8.2.3 Overview

The evaluator implements JSL's core evaluation semantics:

- **Expressions**: Everything in JSL is an expression that evaluates to a value
- **Environments**: Lexical scoping with nested environment chains
- **Host Commands**: Bidirectional communication with the host system
- **Tail Call Optimization**: Efficient recursion handling

8.2.4 Evaluation Rules

Literals

- **Numbers**: `42`, `3.14` evaluate to themselves
- **Strings**: `"@hello"` evaluates to the literal string `"hello"`
- **Booleans**: `true`, `false` evaluate to themselves
- **null**: `null` evaluates to itself

Variables

Variable references are resolved through the environment chain:

```
["let", {"x": 42}, "x"]
```

Special Forms

- `let`: Creates local bindings

- `def` : Defines variables in the current environment
- `lambda` : Creates function closures
- `if` : Conditional evaluation
- `do` : Sequential execution
- `quote` : Prevents evaluation
- `host` : Executes host commands

Function Calls

Regular function calls use list syntax:

```
["func", "arg1", "arg2"]
```

Where `func` evaluates to a callable (function or closure).

Objects

Objects are evaluated by evaluating all key-value pairs:

```
{"key": "value", "computed": ["add", 1, 2]}
```

Keys must evaluate to strings, values can be any JSL expression.

8.2.5 Error Handling

The evaluator provides detailed error information including:

- Expression context
- Environment state
- Call stack trace
- Host command failures

8.2.6 Security

The evaluator includes security measures:

- **Sandboxing**: Host commands are controlled by the dispatcher
- **Resource Limits**: Evaluation depth and memory usage controls
- **Safe Evaluation**: No access to Python internals by default

8.2.7 Usage Examples

Basic Evaluation

```
from jsl.core import Evaluator, Env

evaluator = Evaluator()
env = Env()

# Evaluate a simple expression
result = evaluator.eval(["+", 1, 2], env)
print(result)  # 3
```

With Variables

```
# Define a variable
evaluator.eval(["def", "x", 42], env)

# Use the variable
```

```
result = evaluator.eval(["*", "x", 2], env)
print(result)  # 84
```

Function Definition and Call

```
# Define a function
evaluator.eval(["def", "square", ["lambda", ["x"], ["*", "x", "x"]]], env)

# Call the function
result = evaluator.eval(["square", 5], env)
print(result)  # 25
```

Host Commands

```
from jsl.core import HostDispatcher

# Create a dispatcher with custom commands
dispatcher = HostDispatcher()
dispatcher.register("print", lambda args: print(*args))

evaluator = Evaluator(host_dispatcher=dispatcher)

# Execute a host command
evaluator.eval(["host", "print", "@Hello, World!"], env)
```

8.2.8 Performance Considerations

Tail Call Optimization

The evaluator optimizes tail calls to prevent stack overflow in recursive functions.

Memory Management

- Environments use reference counting
- Closures are garbage collected when no longer referenced
- Host commands can implement resource limits

Caching

- Function closures cache their compiled form
- Environment lookups are optimized for common patterns
- Object evaluation caches key-value pairs when possible

8.3 Serialization API

For conceptual background on JSL's serialization design, see Architecture: Serialization.

8.3.1 Overview

The JSL serialization API provides functions for converting JSL code and data structures to and from JSON representations using content-addressable storage. This approach elegantly handles circular references and ensures efficient serialization of complex object graphs including closures and environments.

8.3.2 Core Functions

JSL Serialization - JSON serialization for JSL values and closures

This module handles the serialization and deserialization of JSL values, including closures with their captured environments. Uses content-addressable storage to handle circular references elegantly.

8.3.3 `serialize(obj)`

Serialize a JSL value to JSON string.

Parameters:

Name	Type	Description	Default
`obj`	`Any`	The JSL value to serialize	*required*

Returns:

Type	Description
`str`	JSON string representation

Source code in `jsl/serialization.py`

```
360  def serialize(obj: Any) -> str:
361      """
362      Serialize a JSL value to JSON string.
363
364      Args:
365          obj: The JSL value to serialize
366
367      Returns:
368          JSON string representation
369      """
370      serializer = ContentAddressableSerializer()
371      return serializer.serialize(obj)
```

8.3.4 `deserialize(json_str, prelude_env=None)`

Deserialize a JSON string to JSL value.

Parameters:

Name	Type	Description	Default
`json_str`	`str`	JSON string to deserialize	*required*
`prelude_env`	`Env`	Optional prelude environment for closure reconstruction	`None`

Returns:

Type	Description
`Any`	Reconstructed JSL value

<details>
<summary>Source code in <code>jsl/serialization.py</code></summary>

```
def deserialize(json_str: str, prelude_env: Env = None) -> Any:
    """
    Deserialize a JSON string to JSL value.

    Args:
        json_str: JSON string to deserialize
        prelude_env: Optional prelude environment for closure reconstruction

    Returns:
        Reconstructed JSL value
    """
    deserializer = ContentAddressableDeserializer(prelude_env)
    return deserializer.deserialize(json_str)
```

</details>

8.3.5 `to_json(obj)`

Convert JSL value to JSON-compatible dictionary.

Parameters:

Name	Type	Description	Default
`obj`	`Any`	JSL value to convert	*required*

Returns:

Type	Description
`Dict`	JSON-compatible dictionary

<details>
<summary>Source code in <code>jsl/serialization.py</code></summary>

```
def to_json(obj: Any) -> Dict:
    """
    Convert JSL value to JSON-compatible dictionary.

    Args:
        obj: JSL value to convert

    Returns:
        JSON-compatible dictionary
    """
    serialized = serialize(obj)
    return json.loads(serialized)
```

</details>

8.3.6 `from_json(json_data, prelude_env=None)`

Reconstruct JSL value from JSON data.

Parameters:

Name	Type	Description	Default
`json_data`	`Any`	JSON string or dictionary	*required*
`prelude_env`	`Env`	Optional prelude environment	`None`

Returns:

Type	Description
`Any`	Reconstructed JSL value

Source code in `jsl/serialization.py`

```
403    def from_json(json_data: Any, prelude_env: Env = None) -> Any:
404        """
405        Reconstruct JSL value from JSON data.
406
407        Args:
408            json_data: JSON string or dictionary
409            prelude_env: Optional prelude environment
410
411        Returns:
412            Reconstructed JSL value
413        """
414        if isinstance(json_data, str):
415            return deserialize(json_data, prelude_env)
416        else:
417            # Convert dict back to JSON string and deserialize
418            json_str = json.dumps(json_data)
419            return deserialize(json_str, prelude_env)
```

8.3.7 `serialize_program(program, prelude_hash=None)`

Serialize a complete JSL program with metadata.

Parameters:

Name	Type	Description	Default
`program`	`Any`	The JSL program to serialize	*required*
`prelude_hash`	`str`	Optional hash of the prelude version	`None`

Returns:

Type	Description
`Dict`	Dictionary with program and metadata

Source code in `jsl/serialization.py`

```
422    def serialize_program(program: Any, prelude_hash: str = None) -> Dict:
423        """
424        Serialize a complete JSL program with metadata.
425
426        Args:
427            program: The JSL program to serialize
428            prelude_hash: Optional hash of the prelude version
429
430        Returns:
431            Dictionary with program and metadata
432        """
433        return {
434            "version": "0.1.0",
435            "prelude_hash": prelude_hash,
436            "program": to_json(program),
437            "timestamp": None  # Could add timestamp if needed
438        }
```

8.3.8 `deserialize_program(program_data, prelude_env=None)`

Deserialize a complete JSL program.

Parameters:

Name	Type	Description	Default
`program_data`	`Dict`	Serialized program data	*required*
`prelude_env`	`Env`	Prelude environment for reconstruction	`None`

Returns:

Type	Description
`Any`	Reconstructed JSL program

Source code in `jsl/serialization.py`

```
def deserialize_program(program_data: Dict, prelude_env: Env = None) -> Any:
    """
    Deserialize a complete JSL program.

    Args:
        program_data: Serialized program data
        prelude_env: Prelude environment for reconstruction

    Returns:
        Reconstructed JSL program
    """
    # Could add version checking here
    program = program_data.get("program")
    return from_json(program, prelude_env)
```

8.3.9 Usage Examples

Basic Serialization

```
from jsl.serialization import serialize, deserialize

# Serialize simple JSL values
expr = ["+", 1, 2]
json_str = serialize(expr)
# Result: '["+", 1, 2]'

# Deserialize back to JSL
restored = deserialize(json_str)
# Result: ["+", 1, 2]

# Complex values use content-addressable format
from jsl import eval_expression, make_prelude
closure = eval_expression('["lambda", ["x"], ["+", "x", 1]]', make_prelude())
serialized = serialize(closure)
# Result contains "__cas_version__", "root", and "objects" fields
```

Closure Serialization

```
from jsl import eval_expression, make_prelude, serialize, deserialize

# Create and serialize a closure with captured environment
program = '''
["do",
  ["def", "base", 100],
  ["lambda", ["x"], ["+", "x", "base"]]
]
'''
closure = eval_expression(program, make_prelude())
serialized = serialize(closure)

# Deserialize with prelude environment
restored = deserialize(serialized, make_prelude())
```

Network Transmission

```
import json
from jsl.serialization import serialize
```

```
# Prepare JSL code for network transmission
code = ["lambda", ["x"], ["*", "x", "x"]]
payload = {
    "type": "execute",
    "code": serialize(code),
    "timestamp": "2023-12-01T10:00:00Z"
}

# Send as JSON
json_payload = json.dumps(payload)
```

Program Serialization

```
from jsl.serialization import serialize_program, deserialize_program

# Serialize complete program with metadata
program = ["+", 1, 2, 3]
program_data = serialize_program(program, prelude_hash="v1.0")

# Result includes version, prelude_hash, and program data
# {
#   "version": "0.1.0",
#   "prelude_hash": "v1.0",
#   "program": 6,
#   "timestamp": null
# }

# Deserialize program
restored = deserialize_program(program_data)
```

8.3.10 Type Mappings

JSL Type	JSON Type	Notes
Number	Number	Direct mapping for primitives
String	String	Direct mapping for primitives
Boolean	Boolean	Direct mapping for primitives
Null	Null	Direct mapping for primitives
Array	Array	Direct for simple arrays, CAS for arrays containing closures
Object	Object	Direct for simple objects, CAS for objects containing closures
Closure	Object	CAS format with `__type__: "closure"`
Environment	Object	CAS format with `__type__: "env"`

8.3.11 Serialization Formats

JSL uses two serialization formats depending on the complexity of the data:

1. **Direct JSON**: For primitive values and simple structures without closures
2. **Content-Addressable Storage (CAS)**: For complex objects containing closures or environments

```
# Direct JSON format
serialize(42)              # "42"
serialize([1, 2, 3])      # "[1,2,3]"
serialize({"key": "val"}) # "{\"key\":\"val\"}"

# CAS format (contains closures/environments)
serialize(some_closure)   # {"__cas_version__": 1, "root": {...}, "objects": {...}}
```

8.3.12 Error Handling

The serialization API handles various error conditions:

- **Circular References**: Handled elegantly using content-addressable storage
- **Invalid JSON**: Proper error messages for malformed input
- **Type Errors**: Clear indication of unsupported types
- **Encoding Issues**: UTF-8 handling for international text
- **Missing Objects**: Validation of object references during deserialization

8.3.13 Performance Notes

- **Time Complexity**: O(n) where n is the size of the data structure
- **Space Complexity**: Efficient sharing of identical objects through content addressing
- **Circular Reference Handling**: No stack overflow or infinite loops
- **Deterministic Hashing**: Same content always produces same hash

8.4 Runner API

8.4.1 Overview

The JSL Runner API provides the core execution engine for JSL programs, handling evaluation, environment management, and host interaction coordination.

8.4.2 Core Classes

JSL Runner - High-level execution interface

This module provides the JSLRunner class and related utilities for executing JSL programs with advanced features like environment management, host interaction, and performance monitoring.

8.4.3 `JSLRunner(config=None, security=None, resource_limits=None, host_gas_policy=None, use_recursive_evaluator=False)`

High-level JSL execution engine with advanced features.

Initialize JSL runner.

Parameters:

Name	Type	Description	Default
`config`	`Optional[Dict[str, Any]]`	Configuration options (recursion depth, debugging, etc.)	`None`
`security`	`Optional[Dict[str, Any]]`	Security settings (allowed commands, sandbox mode, etc.)	`None`
`resource_limits`	`Optional[ResourceLimits]`	Resource limits for execution	`None`
`host_gas_policy`	`Optional[HostGasPolicy]`	Gas cost policy for host operations	`None`
`use_recursive_evaluator`	`bool`	If True, use recursive evaluator instead of stack (default: False)	`False`

Source code in `jsl/runner.py`

```
def __init__(self, config: Optional[Dict[str, Any]] = None,
             security: Optional[Dict[str, Any]] = None,
             resource_limits: Optional[ResourceLimits] = None,
             host_gas_policy: Optional[HostGasPolicy] = None,
             use_recursive_evaluator: bool = False):
    """
    Initialize JSL runner.

    Args:
        config: Configuration options (recursion depth, debugging, etc.)
        security: Security settings (allowed commands, sandbox mode, etc.)
        resource_limits: Resource limits for execution
        host_gas_policy: Gas cost policy for host operations
        use_recursive_evaluator: If True, use recursive evaluator instead of stack (default: False)
    """
    self.config = config or {}
    self.security = security or {}
    self.use_recursive_evaluator = use_recursive_evaluator

    # Set up host dispatcher
    self.host_dispatcher = HostDispatcher()

    # Set up base environment
    self.base_environment = make_prelude()

    # Set up resource limits
    if resource_limits is None and self.config:
        # Build from config
        resource_limits = ResourceLimits(
            max_gas=self.config.get('max_gas'),
            max_memory=self.config.get('max_memory'),
            max_time_ms=self.config.get('max_time_ms'),
            max_stack_depth=self.config.get('max_stack_depth'),
            max_collection_size=self.config.get('max_collection_size'),
            max_string_length=self.config.get('max_string_length')
        )

    # Set up evaluators
    if use_recursive_evaluator:
        # Recursive evaluator as reference implementation
        self.recursive_evaluator = Evaluator(
            self.host_dispatcher,
            resource_limits=resource_limits,
            host_gas_policy=host_gas_policy
        )
        self.stack_evaluator = None
    else:
        # Stack evaluator is the default for production use
        # Create resource budget if we have limits
        budget = None
        if resource_limits:
            from .resources import ResourceBudget
            budget = ResourceBudget(resource_limits, host_gas_policy)
        self.stack_evaluator = StackEvaluator(
            env=self.base_environment.to_dict(),
            resource_budget=budget,
            host_dispatcher=self.host_dispatcher
        )
        self.recursive_evaluator = None

    # Keep backward compatibility - evaluator points to the active one
    self.evaluator = self.recursive_evaluator if use_recursive_evaluator else self.stack_evaluator

    # Store for reference
    self.resource_limits = resource_limits
    self.host_gas_policy = host_gas_policy

    # Performance tracking
    self._profiling_enabled = False
    self._performance_stats = {}

    # Apply configuration
    self._apply_config()
```

`add_host_handler(command, handler)`

Add a host command handler.

Parameters:

Name	Type	Description	Default
`command`	`str`	Command name (e.g., "file", "time")	*required*
`handler`	`Any`	Handler object or function	*required*

Source code in `jsl/runner.py`

```
def add_host_handler(self, command: str, handler: Any) -> None:
    """
    Add a host command handler.

    Args:
        command: Command name (e.g., "file", "time")
        handler: Handler object or function
    """
    # Check security restrictions
    allowed_commands = self.security.get('allowed_host_commands')
    if allowed_commands and command not in allowed_commands:
        raise JSLRuntimeError(f"Host command '{command}' not allowed by security policy")

    self.host_dispatcher.register(command, handler)
```

`disable_profiling()`

Disable performance profiling.

Source code in `jsl/runner.py`

```
def disable_profiling(self) -> None:
    """Disable performance profiling."""
    self._profiling_enabled = False
```

`enable_profiling()`

Enable performance profiling.

Source code in `jsl/runner.py`

```
def enable_profiling(self) -> None:
    """Enable performance profiling."""
    self._profiling_enabled = True
    self._performance_stats = {}
```

`execute(expression)`

Execute a JSL expression.

Supports multiple input formats: - S-expression Lisp style: "(+ 1 2 3)" - S-expression JSON style: "["+", 1, 2, 3]" - JPN postfix compiled: "[1, 2, 3, 3, "+"]"

Parameters:

Name	Type	Description	Default
`expression`	`Union[str, JSLExpression]`	JSL expression as string or parsed structure	*required*

Returns:

Type	Description
`JSLValue`	The result of evaluating the expression

Raises:

Type	Description
`JSLSyntaxError`	If the expression is malformed
`JSLRuntimeError`	If execution fails

Source code in `jsl/runner.py`