OrgLang Reference

- Full Grammar specification

OrgLang Reference Manual

Introduction

Notation

Lexical analysis

Lexical analysis is the first stage of the OrgLang compiler. An OrgLang program is read as a sequence of characters, which are then grouped into meaningful units called **tokens**.

The lexer (or tokenizer) performs this transformation by scanning the source text from beginning to end. It identifies various types of tokens: - **Identifiers and Keywords**: Symbolic names for variables and functions.

- **Literals**: Constant values such as numbers and strings.
- **Operators**: Symbols representing computations or flows.
- **Delimiters**: Structural symbols like parentheses and semicolons.

Character encoding

OrgLang source files are expected to be encoded in **UTF-8**. While the current implementation primarily focuses on the ASCII subset for structural elements and identifiers, UTF-8 support ensures that strings and comments can contain any Unicode character.

Line structure

Comments

OrgLang supports two types of comments: single-line and multiline (block) comments.

Single-line comments start with the hash character (`#`) and extend to the end of the line. They are ignored by the compiler.

```
# This is a single-line comment
x : 42; # Comment after an expression
```

Multiline comments (also known as block comments) are enclosed in three consecutive hash characters (`###`).

> [WARNING] The multiline comment marker `###` must start at the first column of the line.

Everything between the opening and closing `###` markers is treated as a comment and ignored.

```
###
This is a multiline comment.
It can span multiple lines.
###
```

Blank lines

A line that contains only whitespace (spaces, tabs, and form feeds) is considered a blank line and is ignored by the compiler. Blank lines are recommended to separate logical blocks of code and improve readability.

Indentation

Unlike some other languages (like Python), indentation in OrgLang is generally not semantically significant. It is used primarily for readability and to reflect the structural hierarchy of nested blocks (e.g., inside `{}` or `[]`).

However, there are specific lexical rules where column position matters:

- The multiline comment marker `###` **must** start in the first column of the line.

Whitespace between tokens

Whitespace (spaces, tabs, and newlines) is used to separate tokens that would otherwise be joined. For example, `x : 42` requires whitespace around `:` if it were part of a larger word, but since `:` is a special symbol, it can often be used without spaces (e.g., `x:42`).

Certain symbols can be used without surrounding whitespace as they are recognized as distinct delimiters:

- `@`, `:`, `.`, `,`
- `(`, `)`, `[`, `]`, `{`, `}`

While not strictly required for these symbols, using whitespace is encouraged for visual clarity.

Identifiers and keywords

Identifiers

Identifiers (also referred to as names) are used to name variables, functions, and resources. In OrgLang, identifiers have a very flexible structure, allowing many symbols that are typically reserved for operators in other languages.

An identifier must start with a letter (case-sensitive `a-z` or `A-Z`), an underscore (`_`), or any of the following symbols:

- `!`, `$`, `%`, `&`, `*`, `-`, `+`, `=`, `^`, `~`, `?`, `/`, `<`, `>`, `|`

After the first character, an identifier can contain any combination of letters, underscores, digits (`0-9`), and the symbols listed above.

However, identifiers **cannot** contain the following structural delimiters:

- `@`, `:`, `.`, `,`, `;`, `(`, `)`, `[`, `]`, `{`, `}`

Examples of valid identifiers: - `variable_name`

- `isValid?`
- `++count`
- `>>`
- `my-module`
- `$price`

Restricted Names: Identifiers that match any of the language's Keywords are reserved and cannot be used as variable names.

> [NOTE] Since digits are allowed in identifiers but an identifier cannot *start* with a digit, the lexer can easily distinguish between numeric literals and names. For example, `42x` is not a valid identifier, but `x42` is.

Keywords

The following identifiers are reserved as keywords and have special meaning in the OrgLang language. They cannot be used as ordinary identifiers:

- `true`: Boolean truth value.
- `false`: Boolean falsehood value.
- `resource`: Used in resource definitions.
- `this`: Refers to the current function or block (used for recursion).
- `left`: Predefined name for the left operand in a binary operator.
- `right`: Predefined name for the right operand (or the operand of a prefix operator).

Reserved classes of identifiers

Literals

String literals

In OrgLang, strings are sequences of characters used for text representation. They can be defined as single-line or multiline literals.

Single-line Strings

Single-line strings are enclosed in double quotes (`"`). They must begin and end on the same line.

```
message : "Hello, OrgLang!";
```

Multiline Strings (DocStrings)

Multiline strings are enclosed in triple double quotes (`"""`). They can span multiple lines and are designed for large blocks of text or documentation.

To keep the source code clean, multiline strings automatically strip **common leading whitespace** (indentation) from all non-empty lines. The amount of whitespace removed is determined by the line with the least indentation.

```
# The resulting string will have no leading spaces on
        "Line 1" and "Line 2"
doc : """
    Line 1
    Line 2
""";
```

> [NOTE] Leading and trailing blank lines (usually surrounding the delimiters) are also stripped.

Strings as Tables

A fundamental design choice in OrgLang is that **Strings are semantically Tables** (ordered lists of characters).

- **Indexing**: A string can be indexed by integers starting at `0`. Accessing `s.0` returns the first character.
- **Table Properties**: Because they are tables, operators like `->` for iteration treat strings as a stream of characters.

String Length and Orthogonality

Strings behave like tables when used in arithmetic contexts: - **Numeric Value**: When used with arithmetic operators (like `+` or `-`), a string evaluates to its **length**. - **No Concatenation**: Unlike many languages, the `+` operator does **not** concatenate strings. Instead, it adds their lengths (or a string's length to a number).

```
s1 : "ABC";
s2 : "DE";
res : s1 + s2; # res is 5 (3 + 2)
```

To join strings, use interpolation or specialized table operations (to be defined in the standard library).

Numeric literals

In OrgLang, all first-class numeric literals (integers and decimals) are designed to be implemented with **arbitrary precision**. This means that, by default, numbers are not limited by the standard 32-bit or 64-bit constraints of the underlying hardware, allowing for exact computations with very large or very precise values.

> [NOTE] While the language semantics favor arbitrary precision, future versions of the compiler will introduce support for specific machine types (like `int`, `long`, `float`, `double`) as internal optimizations. These will be used when the compiler can prove that the range and precision requirements are satisfied by the more efficient machine representations.

Integer literals

Integer literals are represented as a sequence of one or more digits (`0-9`).

Signed Integer Literals

An integer literal can be preceded by an optional sign character (`+` or `-`).

> [WARNING] To be treated as a single numeric literal, there **MUST NOT**
>
> be any whitespace between the sign and the digits.

- `-42`: A single integer literal token with value negative 42.
- `- 42`: Two tokens: the unary negation operator `-` followed by the integer literal `42`.

While the external behavior might often be similar, the distinction is important for the binding power of operators and lexer-level identification of values.

Decimal literals

Decimal literals represent non-integer numbers using a fixed decimal point notation. In OrgLang, these are distinct from the "floating point" types found in many other languages because they are designed for arbitrary precision and avoid the precision loss typical of binary floating point representations.

Syntax

A decimal literal consists of an integer part and a fractional part separated by a dot (`.`).

- **Digits on both sides**: For an unsigned number starting with a digit, there must be at least one digit on both sides of the dot (e.g., `3.14`).

- **Lexical Distinction**: A number followed by a dot without a subsequent digit (e.g., `1.`) is lexically interpreted as an Integer literal followed by the Dot operator.
- **No Scientific Notation**: OrgLang does not currently support scientific notation (e.g., `1e10`) in its literal syntax.

Signed Decimal Literals

Like integers, decimal literals can be preceded by an optional sign (`+` or `-`) with no intervening whitespace.

```
pi : 3.14159;
negative_small : -0.0001;
positive : +1.0;
```

Rational literals

Rational literals represent exact fractional numbers using a syntax that clearly separates the numerator and denominator. This ensures precision in calculations involving fractions.

Syntax

A rational literal is formed by two integer literals separated by a forward slash (`/`).

- **Numerator and Denominator**: Both the numerator and the denominator are integer literals (which can be positive or negative, as defined in Integer literals).
- **No Whitespace**: There must be no whitespace between the numerator, the slash, and the denominator.
- **Zero Denominator**: A zero in the denominator is syntactically valid but will result in a runtime error (division by zero) during evaluation.

Examples

```
one_half : 1/2;
three_quarters : 3/4;
negative_fraction : -1/2;
large_fraction : 123456789/987654321;
```

Table literals

Tables (also referred to as Lists) are the primary data structure in OrgLang. There is a single, unified model for table construction, whether using blocks or operators.

Construction: Blocks and Commas

A table can be constructed in two ways that produce the same semantic object:

1. **Blocks (`[]`)**: Square brackets group a sequence of statements or expressions into a Table. Elements are typically separated by whitespace.
2. **The Comma Operator (`,`)**: The comma is a binary operator that creates or extends a Table.

Because `[]` evaluates its contents and collects the results into a new Table, using commas inside brackets results in **nesting**.

```
# A simple table
t1 : [1 2 3];

# The same table using commas (outside brackets)
t2 : 1, 2, 3;

# NESTED: [1, 2, 3] creates a Table containing a Table
t3 : [1, 2, 3]; # Result is [[1 2 3]]
```

Implicit Indexing vs. Bindings

A Table consists of a sequence of elements. These elements are categorized into two types: - **Bindings**: Pairs created using the binding operator (`:`). These are accessed by their key and **do not** consume a numeric index slot. - **Positional Elements**: All other expressions. These are assigned implicit numeric indexes (`0, 1, 2...`) based on their order of occurrence among other positional elements.

Mixed Content and Indexing

When a Table contains both bindings and positional elements, the numeric indexes skip the bindings.

```
mixed : [10, "status" : "active", 20];

val0 : mixed.0;         # 10
status : mixed."status"; # "active"
val1 : mixed.1;         # 20 (NOT mixed.2)
```

Tables as Blocks

Since every Org file is itself a Table, the rules for table literals apply to the top-level structure of a program. A file containing `a:1; b:2;` is a Table where `a` and `b` can be accessed by name.

Laziness in Tables

Values within a table are **lazy by default**. They are represented as thunks and are only evaluated when accessed (e.g., using the `.` or `?` operators).

```
computation : [1 + 1, 2 * 2]; # Expressions are not
        evaluated yet
result : computation.0;       # 2 (evaluation happens here)
```

Operators

In OrgLang, almost every operation and structural construct is modeled as an **operator**. The language is designed to be highly orthogonal, with a minimal set of core rules that govern how these operators interact within expressions. Unlike many traditional languages that distinguish between operators, functions, and control structures, OrgLang treats nearly everything—from arithmetic to resource management and conditional evaluation—as an expression driven by operators.

Philosophy and Mechanics

OrgLang operators are strictly **unary** (prefix) or **binary** (infix). This strictness simplifies the language's grammar and execution model but introduces a different way of thinking about computation:

- **Binding Power**: The behavior of an expression is determined by the binding power (precedence) of its operators. Operators with higher binding power "pull" operands closer than those with lower power.

- **Everything is an Expression**: Operators don't just "perform actions"; they transform values and return new ones. This allows for deeply nested and highly expressive chains of computation.

Limitations and Patterns

The limitation to unary and binary forms (maximum of two operands) may seem restrictive compared to the variety of arities found in other languages. However, OrgLang overcomes this through several powerful patterns:

- **Tables as Parameters**: To pass multiple values to an operation that only accepts one operand (like a unary function call), those values are grouped into a Table literal. The operation then extracts exactly what it needs from the table.
- **Currying**: Binary operators can be used to "partially apply" data. An expression like `a op b` can return a new thunk or function that is "ready" to take more data later.
- **Abstractions**: Simple operators can be composed and bound to names, creating high-level abstractions that behave like complex built-in features in other languages.

By embracing these patterns, OrgLang achieves a high degree of expressiveness while maintaining a structurally simple core.

Arithmetic operators

Arithmetic operators perform standard mathematical calculations. In OrgLang, these operators are designed to work with arbitrary-precision Numeric literals.

OPERATOR	NAME	ARITY	DESCRIPTION
`+`	Addition	Binary	Returns the sum of two numbers.
`-`	Subtraction	Binary	Returns the difference

OPERATOR	NAME	ARITY	DESCRIPTION
			between two numbers.
`-`	Negation	Unary	Returns the additive inverse of a number.
`*`	Multiplication	Binary	Returns the product of two numbers.
`/`	Division	Binary	Returns the quotient of two numbers.
`**`	Power	Binary	Returns the left operand raised to the power of the right operand (Right-associative).

> [NOTE]
> **IMPLICIT COERCION**
> : Any arithmetic operator can be applied to Table literals and Strings, in which case their **size** is used as the numeric value. Additionally, Boolean literals are coerced to numbers: `true` is treated as `1`, and `false` as `0`.

Bitwise operators

OrgLang supports standard bitwise operations for integers:

- `&`: Bitwise AND
- `|`: Bitwise OR
- `^`: Bitwise XOR
- `~`: Bitwise NOT (Prefix)

- `<<` : Left Shift
- `>>` : Right Shift

Example:

```
10 & 2  // 2
10 | 5  // 15
10 ^ 5  // 15
~0      // -1
1 << 2  // 4
8 >> 1  // 4
```

Comparison operators

Comparison operators compare two values and always return a Boolean literal (`true` or `false`). OrgLang supports standard comparison operations, as well as automatic type coercion (e.g., comparing a string length to an integer).

OPERATOR	DESCRIPTION	EXAMPLE
`=`	Equal to	`x = y`
`<>` , `~=`	Not equal to	`x <> y`
`<`	Less than	`x < y`
`<=`	Less than or equal to	`x <= y`
`>`	Greater than	`x > y`
`>=`	Greater than or equal to	`x >= y`

> [NOTE]
> **IMPLICIT COERCION**
>
> : Comparison operators follow the same coercion rules as Arithmetic operators: Tables and Strings use their size, and Booleans are treated as `1` (`true`) or `0` (`false`).

> [WARNING]
> **COMPARISON CHAINING**
>
> : Since every comparison returns a Boolean, the result of a chain (e.g., `x < y < z`) is the result of the **last comparison** in the chain. This differs from languages where such a chain might be shorthand for `(x < y) && (y < z)`.

Boolean operators

Boolean operators are used to perform logical calculations.

OPERATOR	NAME	ARITY	DESCRIPTION
`!`	Logical NOT	Unary	Returns the logical negation (e.g., `! 0 = 1`).
`~`	Bitwise NOT	Unary	Returns the bitwise complement (e.g., `~ 0 = -1`).
`&&`	AND	Binary	Short-circuit logical AND (returns `true` only if both are `true`).
`||`	OR	Binary	Short-circuit logical OR (returns `true` if at least one is `true`).
`&`	Logical AND	Binary	**Non-short-circuit** logical AND.
`|`	Logical OR	Binary	

OPERATOR	NAME	ARITY	DESCRIPTION
			Non-short-circuit logical OR.
`^`	Logical XOR	Binary	Returns `true` if exactly one of the operands is `true`.

> [NOTE]
> **TRUTHINESS**
>
> : Boolean operators can be applied to Table literals and Strings. They follow a "size-based" truthiness rule: a size of `0` is treated as `false`, and every other value (size `> 0`) is treated as `true`.

Conditional operators

Conditional operators allow for selection and branching within expressions without traditional `if/else` statements.

OPERATOR	NAME	ARITY	DESCRIPTION
`.`	Dot Access	Binary	Static/ Positional access to a Table's elements or keys.
`?`	Selection Access	Binary	Conditional or dynamic selection from a Table. Evaluation-driven.
`??`	Error Check	Binary	Returns the right operand if the left

OPERATOR	NAME	ARITY	DESCRIPTION
			operand is an **Error**; otherwise, returns the left operand.
`?:`	Elvis Operator	Binary	Returns the right operand if the left operand is **"falsy"** (false, Error, or an empty Table/String); otherwise, returns the left operand.

Resource operators

Resource operators manage the lifecycle and data flow of Resources.

Resource Instantiation (`@`)

The prefix `@` operator is used to instantiate a resource. When applied to a resource name or literal, it executes the resource's `setup` block and returns a **Resource Instance**.

```
# Instantiate stdout
@stdout
```

Data Flow (`->`)

The binary `->` operator drives data from a source (left operand) to a sink (right operand).

- **Source -> Sink**: Drives all data from the source into the sink until completion.
- **Iterator -> Function**: Creates a new projection (map) that will process elements lazily.

```
# Send a string to stdout
"Hello" -> @stdout;

# Send input through a transform to output
@stdin -> { args * 2 } -> @stdout;
```

Balanced Data Flow (`-<`)

The binary `-<` operator performs a balanced dispatch of data. It sends each element from the left source to exactly **one** of the available sinks in the Table on the right side, typically using a round-robin or load-balancing strategy.

- **Load Balancing**: If the right operand is a Table of sinks, elements are distributed among them.
- **Degeneration to `->`**: If the right operand contains only one sink, it behaves identically to the basic data flow operator (`->`).

```
# Distribute tasks between two workers
@tasks -< [worker1 worker2];
```

Join Data Flow (`-<>`)

The binary `-<>` operator acts as a synchronizing barrier. It is used to merge multiple data streams into a single flow of coordinated packets.

- **Synchronization**: It waits for **every source** in the left Table to produce at least one element.
- **Aggregation**: Once one element is received from each source, it combines them into a single Table and sends that Table as a single “pulse” to the right operand.

```
# Synchronize data from two sensors before processing
[sensor1 sensor2] -<> processor;
```

Assignment operators

In OrgLang, assignment is strictly an operation that binds a value to a name within a Table.

OPERATOR	NAME	DESCRIPTION
`:`	Binding	Binds the result of the right expression to the name specified on the left.

> [NOTE]
> **EXTENDED ASSIGNMENT**
>
> : OrgLang reserves the following operators for extended assignment (modification of existing bindings). These are **not yet implemented** in the current runtime.

OPERATOR	DESCRIPTION	EXAMPLE
`:+`	Addition and Assignment	`x :+ 2`
`:-`	Subtraction and Assignment	`x :- 1`
`:*`	Multiplication and Assignment	`x :* 3`
`:/`	Division and Assignment	`x :/ 4`
`:%`	Modulo and Assignment	`x :% 5`
`++`	Increment and Assignment	`++ x`
`--`	Decrement and Assignment	`-- x`
`:>>`	Right Shift and Assignment	`x :>> 5`
`:<<`	Left Shift and Assignment	`x :<< 5`
`:&`	AND and Assignment	`x :& y`
`:^`	XOR and Assignment	`x :^ y`
`:|`	OR and Assignment	`x :| y`

OPERATOR	DESCRIPTION	EXAMPLE
`:~`	Bitwise NOT and Assignment	`x :~ 1`

Operator definitions

OrgLang allows for the definition of custom operators and the refinement of existing ones using the **Binding Power** syntax. This syntax defines the left and right binding powers, determining the operator's precedence and associativity.

```
# Define a unary operator with prefix power 100
! : 100{ ... };

# Define a binary operator with left power 50 and right
        power 60
op : 50{ ... }60;
```

When an operator is called, the expression within the braces is evaluated. The operands are made available via `left` and `right`. - **`left`**: The left operand (for binary operators). For unary (prefix) operators, this is typically `Error` or `NULL`. - **`right`**: The right operand (for binary operators) or the single operand (for unary operators). - **`this`**: A reference to the operator function itself (useful for recursion).

> [WARNING]
> **STRICT BINDING POWER SYNTAX**
>
> : When defining custom binding powers, there **must not be any whitespace** between the number and the brace. - **Correct**: `op : 50{ ... }60;` - **Incorrect**: `op : 50 { ... } 60;`

Operators on operators

OrgLang provides higher-order operators that allow for the functional construction of logic by combining or specializing existing operators.

The `o` (compose) operator

The binary `o` operator performs **Functional Composition**. it merges two operators into a single, unified transformation.

- **Sequence**: In the expression `h : g o f`, the output of the right operator (`f`) becomes the input of the left operator (`g`).
- **Optimization**: The runtime attempts to fuse these operations into a single execution step to minimize intermediate overhead.

Arity-based Composition Rules (`h : g o f`):

The behavior of the composed operator `h` depends on the arity of `g` and `f`. The general rule is that the result of `f` always populates the `right` slot of `g`, and if `g` is binary, it retains the original `left` operand.

- **Unary `g` o Binary `f`**: `h` is a binary operator. `h(left, right)` evaluates as `g(f(left, right))`.
- **Binary `g` o Unary `f`**: `h` is a binary operator. `h(left, right)` evaluates as `g(left, f(right))`. This effectively uses `f` to pre-process the "main" argument while preserving the context in `left`.
- **Binary `g` o Binary `f`**: `h` is a binary operator. `h(left, right)` evaluates as `g(left, f(left, right))`.
- **Unary `g` o Unary `f`**: `h` is a unary operator. `h(right)` evaluates as `g(f(right))`.

```
# Compose increment and double
inc : { right + 1 };
double : { right * 2 };
inc_and_double : double o inc;

result : inc_and_double 5; # 12
```

The `|>` (partial application) operator

The binary `|>` operator, also known as the **Left Injector**, performs **Partial Application**. It "anchors" a value into the `left` slot of an operator, returning a new unary operator.

- **Specialization**: It allows you to create specialized versions of binary operators by fixing one of the operands.
- **Left Binding**: The value on the left of `|>` is bound to the `left` parameter of the operator on the right.

```
# Create a specialized 'add 10' function
add_ten : 10 |> +;

result : add_ten 5; # 15
```

Delimiters

Delimiters are structural symbols used for grouping expressions, constructing data structures, and defining blocks of code.

Parentheses `( )`

Parentheses are primarily used to **group expressions** and override the default precedence of operators.

```
res : (1 + 2) * 3; # 9
```

They are also used in function calls, although functionally `f(x)` is just `f` applied to the expression `(x)`.

Square Brackets `[ ]`

Square brackets are used to construct **Table literals**. They group a sequence of expressions, evaluate them, and collect the results into a new Table.

```
list : [1 2 3];
nested : [[1 2] 3];
```

Braces `{ }`

Braces are used to define **function bodies** and create **Operators**. The code inside braces is not executed immediately; instead, it is wrapped in an Operator (or thunk) that is evaluated when called.

```
# A simple function
add : { left + right };

# A thunk (parameter-less function)
thunk : { 1 + 1 };
```

Data model

The Data Model defines the fundamental entities and their relationships within OrgLang. It describes how information is represented, organized, and manipulated by the runtime. OrgLang is built on a foundation of extreme orthogonality and high-level abstractions, where complex behaviors emerge from the interaction of a small set of primitive types and universal operators.

Values and types

In OrgLang, information is represented as **Values**. A Value is a piece of data that can be bound to a name, passed as an argument to an operator, or returned as the result of an expression.

The language uses a **Dynamic Typing** model. This means that variables (bindings) do not have types; only the Values themselves carry type information. A variable can hold a Number at one point and a Table later in the execution.

Values vs. Objects

Unlike many object-oriented languages, OrgLang does not strictly distinguish between “primitive values” and “objects.” Every entity, from a simple Integer to a complex Resource Instance, is a first-class Value. Even Errors and Operators are treated as Values that can be manipulated and stored.

First-Class Expressions

Because OrgLang is built on a late-binding, lazy evaluation model, any piece of code enclosed in braces `{}` is itself a Value—an **Operator**. This allows logic to be passed around as data, forming the basis for the language's "Compositional" nature.

Extreme Orthogonality

A hallmark of OrgLang values is their predictable behavior across different operators. For instance, the addition operator `+` is defined for all types:

- Adding two Numbers produces their sum.
- Adding a Table to a Number uses the Table's size.
- Adding two Tables returns the sum of their sizes.

This consistency reduces the need for "special cases" and allows for highly generic code.

The standard type hierarchy

OrgLang organizes its types into a logical hierarchy. While the runtime may implement these as a flat set of structures for performance, semantically they follow this inheritance pattern:

- Expression
    - Error
    - Name
    - Table
        - String
    - Number
        - Integer
        - Rational
        - Decimal
    - Boolean
    - Operator
        - Unary
        - Binary
        - Nullary

Special names

main

One of the special names is `main`. It is a special name because it is the entry point of the program. An org executable will look for a global name `main` and execute it. If `main` is not found, the program will exit with an error.

Execution model

The Execution Model describes how OrgLang programs are evaluated, how names are resolved, and how state is managed over time. The model is centered around the concept of **Persistent Tables** and **Lazy Evaluation**.

Naming and binding

In OrgLang, naming is not a separate storage mechanism but a structural property of Tables.

Everything is a Table

Every scope in OrgLang—whether it's the global file scope, a code block `{ }`, or a module loaded from another file—is semantically a Table. When you perform an assignment using the binding operator `:`, you are performing a key-value insertion into the **Current Table**.

Dynamic Binding and Shadowing

Bindings are resolved dynamically based on lexical scope. When an identifier is evaluated, the runtime looks it up in the current table. If not found, it traverses upward through parent tables (e.g., from an operator's internal scope to the file's global scope). If you assign to a name that already exists in the current scope, the new value shadows (updates) the previous binding.

Evaluation of Bindings (Laziness)

A core feature of OrgLang is that table entries are **Lazy** by default. When you bind an expression to a name, the expression is wrapped in a "thunk" and stored. Evaluation only occurs when the name is explicitly accessed via the Dot Access or Selection Access operators.

```
x : 1 + 2; # 'x' stores the expression { 1 + 2 }
y : x;     # 'y' now also stores the same thunk
result : x; # Accessing 'x' triggers evaluation, result
        becomes 3
```

Errors

Errors in OrgLang are not "exceptions" that interrupt the flow of control; they are **First-Class Values** that participate in the data flow.

Error Propagation

Most operators in OrgLang are "Error-Aware." If any operand of a binary or unary operation is an Error value, the operator does not perform its standard calculation. Instead, it immediately returns the Error value. This allows errors to propagate naturally through complex expressions until they reach a handler or the program's output.

Error Generation

Crucially, **`Error` is not a literal** in OrgLang. You cannot write `x : Error` or `Error + 1` in your source code, as the word `Error` is a type name, not a value constructor. To force the generation of an Error value, you must perform an operation that is mathematically or logically invalid.

```
# Forcing an error through division by zero
val : 1 / 0; # 'val' now holds an Error value
```

Error Handling

Specifically designed operators like `??` (Error Check) and `?:` (Elvis) allow the programmer to detect and recover from Error values. These are the only operators that do not automatically propagate errors from their left operand.

```
# Propagating an error generated from invalid math
( (1/0) + 1 ) * 2; # Returns Error

# Handling an error
val : (1/0) ?? 0; # Returns 0
```

Terminal Signaling

If an Error value is returned by the `main` entry point or remains as the result of a top-level expression, the runtime typically signals this to the user via the system's standard error stream (stderr).

Arithmetic conversions

Arithmetic expressions in OrgLang are designed to be highly predictable and permissive, adhering to the principle of **extreme orthogonality**. Arithmetic operators (`+`, `-`, `*`, `/`, `%`) always aim to return a numeric value (Integer, Rational, or Decimal) by coercing their operands if necessary.

Coercion of Non-Numeric Types

When an arithmetic operator is applied to a non-numeric type, it is automatically coerced into a Number before the operation is performed:

- **Tables and Strings**: Coerced to their **size** (the number of elements or characters).
- **Booleans**: Coerced to `1` for `true` and `0` for `false`.

```
# Extreme orthogonality examples
"Hello" + 1;      # 5 + 1 = 6
[10 20 30] * 2;   # 3 * 2 = 6
true + true;      # 1 + 1 = 2
```

Division Rules

OrgLang handles division between Integers with special care to maintain precision without prematurely forcing floating-point representation.

- **Exact Division**: If one Integer divides another perfectly with no remainder, the result is an **Integer**.
- **Inexact Division**: If there is a remainder, the result is a **Rational**.

```
result1 : 4 / 2; # result1 is Integer 2
result2 : 3 / 2; # result2 is Rational 3/2
```

Numeric Promotion

When operations involve different numeric subtypes, the result is promoted to the most general type:

- Operations involving a **Decimal** typically produce a **Decimal**.
- Operations involving a **Rational** and an **Integer** typically produce a **Rational**.

Atoms

In the Lisp tradition, **Atoms** are the fundamental, indivisible building blocks of OrgLang expressions. An atom represents a specific value or a name that cannot be further broken down by the parser without changing its meaning.

Lexical Space and Separation

OrgLang uses whitespace (spaces, tabs, newlines) to separate tokens. The rules for where space is required, optional, or forbidden are critical to distinguishing between atoms and operators.

- **Mandatory Spaces**: Space is required to separate two atoms that would otherwise merge into a single identifier (e.g., `x y` is two names, `xy` is one).

- **Optional Spaces**: Spaces are optional around delimiters (`( )`, `[ ]`, `{ }`, `,`, `;`) and structural operators (`.`, `:`, `@`, `->`). For example, `(1+1)` is equivalent to `(1 + 1)`.
- **Forbidden Spaces**:
  - **Signed Numbers**: There must be **no space** between a leading sign and the digits for it to be parsed as a negative or positive number atom (e.g., `-1` is a Number, `- 1` is the unary negation operator applied to `1`).
  - **Binding Power**: There must be **no space** between an integer and the braces when defining custom binding powers (e.g., `700{ ... }701`).

Names (Identifiers)

Names are tokens used to refer to bindings in a Table.

- **Characters**: Names can contain letters, digits, underscores, and most symbols (e.g., `isValid?`, `counter_1`, `set!`).
- **Start Rule**: A name cannot start with a digit (which would initiate a Number).
- **Exclusions**: Symbols used as delimiters or structural operators (`( )`, `[ ]`, `{ }`, `,`, `.`, `:`, `@`, `;`) cannot be part of a name.

Literals

Literals are atoms that represent fixed values.

Numbers

Numbers are represented in three subtypes: - **Integers**: Sequences of digits, optionally preceded by a sign (`42`, `-10`). - **Decimals**: Digits containing a decimal point (`3.14`, `-.5`). - **Rationals**: Represented as a ratio of two integers (`2/3`).

Booleans

The keywords `true` and `false` are the only two Boolean atoms.

Strings

Strings are represented by text enclosed in double quotes:

- **Simple**: `"Single line string"`.
- **Multiline**: `"""Triple quotes for multiline text"""`.

Strings are semantically **Tables** where each character is a value indexed by its position.

Atoms in Tables

One of the most important aspects of Atoms in OrgLang is how they interact with Table literals `[ ]`. While the source code of a program is itself a Table where expressions are separated by semicolons `;`, a Table literal uses **space** as its primary separator.

Because spaces are also used to separate atoms within an expression (like `1 + 1`), the use of space inside a Table literal can be ambiguous. In OrgLang, the **space in a Table literal acts as an element separator**, effectively terminating the current expression.

- **Atomic Gathering**: Within `[ ]`, the runtime treats each space-separated sequence as a distinct element if not explicitly grouped.
- **Binding Greediness**: The binding operator `:` inside a Table literal is “greedy” and attaches the name on its left to the immediate next atom on its right. It does not automatically consume subsequent atoms if they are separated by spaces.

```
# Likely an error if you wanted 'a' to be 2:
# 1. A binding/pair (a: 1)
# 2. An operator (+)
# 3. An integer (1)
# Resulting Table: [0: +, 1: 1, a: 1]
table_split : [a: 1 + 1];

# Correct: results in a Table with one specific entry
# The parentheses group '1 + 1' into a single Atomic
        Expression
table_ok : [a: (1 + 1)];
```

Parenthesized forms

Parentheses `( )` are used to group expressions to override precedence. A parenthesized expression is treated as a single atomic unit (an **Atomic Expression**) during the evaluation of the outer expression or when used as an element in a Table.

Advantages and Limitations

The atomic model of OrgLang provides a unique balance of simplicity and expressive power:

- **Advantage: Extreme Orthogonality**: Because every atom (even Operators and Errors) is a first-class Value, the language's core operators work consistently across all data types.
- **Advantage: Structural Purity**: Every source file is semantically a Table literal, making the relationship between code and data perfectly transparent.
- **Limitation: Lexical Sensitivity**: The reliance on space as an element separator in Tables means developers must be mindful of grouping when mixing spaces and operators.
- **Limitation: Precedence Quirks**: Some operators, like unary negation, have lower precedence than exponentiation (`-1**2 = -1`), which preserves mathematical convention but may surprise users coming from languages where unary operators are always highest.

Unary arithmetic and bitwise operations

Unary operators in OrgLang are prefix operators that associate with the immediate expression to their right. They follow the principle of extreme orthogonality, coercing non-numeric types to numbers when necessary.

Negation (`-`)

The unary negation operator reverses the sign of a numeric value.

- **Integers**: Returns a negative or positive Integer.

- **Decimals**: Returns a Decimal with the sign reversed.
- **Precedence Note**: Unary negation has **lower precedence** than exponentiation. This means `-1**2` is evaluated as `-(1**2)`, resulting in `-1`, which aligns with standard mathematical notation.

Increment and Decrement (`++`, `--`)

These operators perform primitive arithmetic addition or subtraction of 1.

- `++x` is semantically equivalent to `x + 1`.
- `--x` is semantically equivalent to `x - 1`.
- Note: These are prefix operators and do not have "postfix" variants in the core language.

Bitwise NOT (`~`)

The bitwise NOT operator returns the bitwise complement of a number.

- **Coercion**: Non-integers are coerced to Integers before the bitwise inversion occurs.
- **Result**: Always returns an Integer. For example, `~0` results in `-1` (using two's complement representation).

Logical NOT (`!`)

The logical NOT operator performs a truthiness check and returns a Boolean.

- **Truthiness**: `0`, `none`, and empty tables/strings are typically considered falsey. All other values are truthy.
- **Result**: Returns `true` if the operand is falsey, and `false` otherwise.

Binary arithmetic operations

Binary arithmetic operations in OrgLang are designed to be permissive and mathematically intuitive. They operate on three numeric types: **Integer**, **Rational**, and **Decimal**.

Standard Operators

- **Addition (`+`)**: Performs numeric addition. If both operands are integers and the result is within integer range, results in an **Integer**. If any operand is a Decimal, results in a **Decimal**.
- **Subtraction (`-`)**: Performs numeric subtraction. Same promotion rules as addition.
- **Multiplication (`*`)**: Performs numeric multiplication.
- **Division (`/`)**: Follows specialized precision rules:
  - **Exact**: `4 / 2` results in **Integer** `2`.
  - **Inexact**: `3 / 2` results in **Rational** `3/2`.
  - **Decimal**: If either operand is a Decimal, the result is a **Decimal**.
- **Modulo (`%`)**: Returns the remainder of division. Typically used with Integers.

Exponentiation (`**`)

The exponentiation operator raises the left operand to the power of the right operand.

- **Precedence**: Higher than unary negation. Thus, `-1**2` is parsed as `-(1**2)` resulting in `-1`.
- **Promotion**: Often results in a **Decimal** if the power is fractional or negative, unless the result can be exactly represented as an Integer or Rational.

Numeric Coercion

Following the principle of **extreme orthogonality**, binary operators automatically coerce non-numeric types into Numbers:

- **Tables/Strings**: Their **size** is used as the numeric value.
- **Booleans**: `true` becomes `1`, `false` becomes `0`.
- **Errors**: Propagate through the operation (the result of any arithmetic with an Error is an Error).

```
# Examples of extreme orthogonality
[10 20] + [30]; # 2 + 1 = 3
"abc" * 2;      # 3 * 2 = 6
true + 1;       # 1 + 1 = 2
```

Shifting operations

Shifting operations in OrgLang are bitwise operations that operate on numeric values, treating them as bit patterns.

Left Shift (`<<`)

The left shift operator moves the bits of the left operand to the left by the number of positions specified by the right operand.

- **Coercion**: Non-integers are coerced to Integers via their numeric representation or size.
- **Arithmetic Effect**: Shifting an integer left by N bits is equivalent to multiplying it by 2^N.
- **Result**: Always an Integer.

Right Shift (`>>`)

The right shift operator moves the bits of the left operand to the right by the number of positions specified by the right operand.

- **Coercion**: Standard numeric coercion applies.
- **Arithmetic Effect**: Shifting an integer right by N bits is equivalent to integer division by 2^N.

- **Result**: Always an Integer.

```
# Examples of shifting
1 << 3;    # 1 * 8 = 8
16 >> 2;   # 16 / 4 = 4
"abcd" << 1; # 4 << 1 = 8 (uses string size)
```

Binary bitwise operations

Binary bitwise operators in OrgLang perform bit-by-bit operations on their operands. Like other arithmetic-adjacent operators, they follow the principle of **extreme orthogonality**, coercing non-numeric types to Integers.

Bitwise AND (`&`)

Returns a number where each bit is `1` only if the corresponding bits of both operands are `1`.

- **Coercion**: Standard numeric/size coercion applies.
- **Result**: Always an Integer.

Bitwise OR (`|`)

Returns a number where each bit is `1` if at least one of the corresponding bits of the operands is `1`.

- **Result**: Always an Integer.

Bitwise XOR (`^`)

Returns a number where each bit is `1` if the corresponding bits of the operands are different.

- **Result**: Always an Integer.

```
# Examples of bitwise operations
5 & 3;    # 101 & 011 = 001 (1)
5 | 3;    # 101 | 011 = 111 (7)
5 ^ 3;    # 101 ^ 011 = 110 (6)
"abc" & 7; # 3 & 7 = 3 (uses string size)
```

Comparisons

Comparison operators in OrgLang are used to determine the relationship between two values. They primarily operate on numeric values but, following the **extreme orthogonality** principle, will coerce non-numeric types to their numeric equivalent (usually their size).

Comparison Operators

OPERATOR	DESCRIPTION	EXAMPLE
`=`	Equal to	`x = y`
`<>` , `~=`	Not equal to	`x <> y`
`<`	Less than	`x < y`
`<=`	Less than or equal to	`x <= y`
`>`	Greater than	`x > y`
`>=`	Greater than or equal to	`x >= y`

- **Result**: All comparison operators return a **Boolean** (`true` or `false`).
- **Equality Limitation**: OrgLang does **not** allow the specialization of equality. There are no plans to support custom equality methods for user-defined tables or operators; equality behavior is baked into the runtime for core types.
- **Numeric Comparisons**:
    - **Integer vs. Rational**: Comparison is **exact**. The Integer is treated as a Rational with a denominator of 1.
    - **Decimal vs. Others**: If any operand is a **Decimal**, the other operand is promoted to a Decimal (inexact) before the comparison is performed.

    - **Decimal vs. Decimal**: Compared based on their approximate numeric values.

- **Structural Comparisons**: For strings or tables, comparisons currently default to evaluating their **sizes**.

Comparison Chaining

OrgLang allows comparison operators to be chained together, such as `x < y < z`.

- **Semantics**: In a chain of comparisons, each operation is evaluated in sequence. However, unlike languages like Python (where `x < y < z` is `(x < y) and (y < z)`), OrgLang's current execution model evaluates the chain left-to-right, and the **result of the entire chain is the result of the last comparison**.
- **Example**: `3 < 5 < 2` would evaluate `3 < 5` (returning `true`, which is `1`), then evaluate `1 < 2`, resulting in `true`.

> [WARNING] Because of how chaining works, users should be cautious. If a mathematical range check is desired, explicit logical ands should be used once available (e.g., `(x < y) && (y < z)`).

Numeric Coercion in Comparisons

As with arithmetic, non-numeric types are coerced to numbers before comparison:

- **Strings/Tables**: Coerced to their **length/size**.
- **Booleans**: `true` is `1`, `false` is `0`.

```
# Examples of orthogonal comparisons
"apple" > "pear"; # 5 > 4 = true
[1 2] = 2;        # 2 = 2 = true
true < 2;         # 1 < 2 = true
```

Boolean operations

Boolean operations in OrgLang are used to combine or invert boolean values. They follow the principle of **extreme orthogonality**, meaning they can be applied to any type by first determining its **truthiness**.

Truthiness Rules

In OrgLang, nearly every value can be evaluated in a boolean context.

- **Truthy**: Any non-zero Number, any non-empty Table (including non-empty Strings), and any active Resource or Operator.
- **Falsey**: The number `0`, empty Tables `[]`, empty Strings `""`, and **Error**.

Logical Operators

OPERATOR	DESCRIPTION	RESULT TYPE
`&&`	Logical AND	Boolean
`||`	Logical OR	Boolean
`!`	Logical NOT	Boolean

- **Short-circuiting**: Per the language design, `&&` and `||` are **short-circuiting** operators. If the result can be determined by the left operand (e.g., `false && ...` or `true || ...`), the right operand is not evaluated.
    - *Implementation Note*: In the current prototype, these may behave strictly; however, programs should be written assuming short-circuiting behavior.
- **Negation (`!`)**: Returns `true` if the operand is falsey, and `false` if it is truthy.

```
# Examples of boolean logic
(1 = 1) && (2 = 2); # true
0 || "hello";       # true (0 is falsey, non-empty string
        is truthy)
! [ ];              # true (empty table is falsey)
! "non-empty";      # false
```

Conditional expressions

OrgLang provides three powerful operators for handling conditions and defaults, aligning with its table-centric and expression-based philosophy.

Table Conditional (`?`)

The `?` operator acts as a conditional lookup. It reverses the standard access order: `condition ? table`.

- **Behavior**: It evaluates the `condition` (left operand) and uses the result as a **key** to look up a value in the `table` (right operand).
- **Usage**: Commonly used with a table containing `true` and `false` keys to mimic an if-else expression.
- **Lazy Evaluation**: Since table values are lazy (thunks), only the selected branch is evaluated.

```
result : (x > 0) ? [true: "Positive" false: "Non-
        positive"];
```

Elvis Operator (`?:`)

The `?:` operator is a “falsy” coalescing operator.

- **Behavior**: Returns the left operand if it is **truthy**. If the left operand is **falsey** (0, empty string, empty table, or Error), it returns the right operand.
- **Usage**: Providing defaults for potentially empty or invalid values.

```
name : input_name ?: "Guest"; # Use "Guest" if input_name
        is empty or error
```

Error Coalescing (`??`)

The `??` operator is a specialized “Error” coalescing operator, similar to Null Coalescing in other languages.

- **Behavior**: Returns the left operand if it is **not** an Error. If the left operand is an **Error**, it returns the right operand.
- **Distinction**: Unlike `?:`, it preserves other falsey values like `0` or `""`.

```
# If calc_value returns 0, result is 0. If it returns
        Error, result is 10.
result : calc_value() ?? 10;
```

Lambdas

Expression lists

Evaluation order

Operator precedence

Assignment

Resources

Operator definitions

Module definitions

File input

Interactive input

Expression input

Full Grammar specification