Especificação de Design: OrgLang

Paulo Suderio

28 janeiro 2026

:TOC:

Language definition

Welcome to the OrgLang reference manual! This guide provides a comprehensive and user−friendly overview of the language, covering syntax, operators, data types, and constructs. OrgLang is designed to be expressive, flexible, and easy to use, supporting arithmetic operations, logic, table manipulation, and more.

Initial Design Goals

- Scripting language, used to fast development
- Very few types:
  - Error
  - Boolean
  - Number (Integer, Rational and Decimal subtypes)
  - Table (String subtype)
  - Operator
  - Resource
- Variables can hold any type (dynamic type, only values have type, not variables)
- Error is the basic value ; in (almost) any operation with error, error is returned
- Resources are, e.g., files, stdin, stdout, etc.
- Every Operator (i.e., Function) must be able to return a value for any type, or Error if the value is not supported.
- Extreme orthogonality of the basic operators. For instance, `[0] + [0 0] = 3` (uses the size of the Table). `"Hello" - 1 = 4`.

Basic Structure

File

A OrgLang program is a sequence of expressions, separated by semicolons (`;`). Example:

```orglang
x : 42;
y : x + 8;
result : y * 2;
```

Every source file returns a Table (more about it below) with the result of every expression and all assigned variables.

Assignments

A variable is assigned using `:`. The left–hand side is the identifier, and the right–hand side is an expression. An assignment is also an expression that returns its right–hand side value.

```orglang name : "John Doe"; age : 30; ```

Supported Literals

1. Integers:

```orglang x : 42; ```

1. Decimals:

```orglang pi : 3.14; ```

1. Rationals:

```orglang two_{thirds} : 2/3; ```

1. Strings:
2. Simple: Enclosed in double quotes (`"`).
3. Multiline (DocStrings): Enclosed in `"""`.

```orglang simpleString : "Hello, World!"; docString : """ Line 1 Line 2 """; ```

1. Booleans:

```orglang flag : true; ```

1. Error:

```orglang value : Error; # Actually, this is not valid code, since Error cannot be assigned. ```

1. Special Identifiers:
2. `this`: Refers to the whole expression of an operator. Used in recursion.
3. `right`: The right argument, or the argument of prefix operators.
4. `left`: The left argument, or the argument of postfix operators.

Operators

There are few primitive operators. Most operators are just a 'default library'.

Primitive

Operator	Description	Example
`.`	Table filter	`table.key`
`@`	Unit	`"Hello World" -> @stdin`
`?`	Test	`(1 = 0) ?`
`??`	Error Test	`x ?? 42`
`?:`	Elvis	`x ?: 42`
`->`	Broadcast Map	`[1 2 3] -> sum`
```

`–<`	Balanced Map	`[1 2 3] –< [sum1 sum2]`	
`–<>`	Join Map	`[1 2 3] –<> sum`	
`o`	Compose	`g o f`	
`\	> `	Partial application	`add1: 1 \ > +`

Arithmetic

Arithmetic operators always return a number (integer or decimal).

Operator	Description	Example
`+`	Addition	`3 + 2`
`–`	Subtraction	`5 – 1`
`*`	Multiplication	`4 * 2`
`/`	Division	`8 / 4`
`%`	Modulo	`5 % 2`
`^`	Exponentiation	`2 3`

Extended Assignment

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`

Logical

Logical operators always return a Boolean.

Operator	Description	Example
`&&`	Short–circuit AND	`true && false`
`\	\	`	Short–circuit OR	`true \	\ false`
`&`	Bitwise AND	`x & y`
`\	`	Bitwise OR	`x \	y`
`^`	Bitwise XOR	`x ^ y`

Comparison

Comparison operators always return a Boolean.

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`

Miscellaneous

Operator	Description	Example
`$`	String substitution	`"Hello $0" $ [42]`
`..`	Numeric range	`1..5`

Tables

1. Construction

   Tables are lists of elements or key–value pairs, constructed with square brackets (`[]`):

   ```orglang table1 : [1 2 3]; table2 : ["key": "value" "anotherKey": 42]; ```

2. Concatenation

   Commas (`,`) can be used to concatenate elements into tables:

   ```orglang result : [1 2], [3 4];
   ```

   If the left hand is a list, ',' adds the right hand argument. If it is not a list, it creates a new list with the two elements.
   ```

Functions and Operators

1. Defining Unary Operators

   Custom operators can be defined using `{}`:

   ```orglang increment : {right + 1}; ```

   Invocation

   ```orglang result : increment 5;
   ```

2. Defining Binary Operators

   ```orglang reverse$_{minus}$: {right – left}; ```

   Invocation

   ```orglang result : 2 reverse$_{minus}$ 6;

3. Recursion

   Every operator defines a `this` operator that references itself.

   ```orglang factorial : {[1 1].right ??  right * (this(right – 1))}; ```

**Strings and Substitutions**

1. Positional Substitution

   Use `$N` to substitute values from a list:

   ```orglang message : "Value: $0, Double: $1" $ [42, 84]; ```

2. Variable Substitution

   Use `$var` to substitute the value of a declared variable:

   ```orglang x : 10; message : "The value is $x"; ```

**Comments**

1. Single−Line

   Start with `#`:

   ```orglang
   ```

   ***Block

   Enclosed in `###`:

   ```orglang ### This is a block comment. ### ```

Resource and Flux

`@` (The Resource Atom)

The `@` operator is the Lifter. It takes a symbol or a string and "prompts" the runtime to associate it with a system effect or a resource handle.

1. `@stdout`: A sink for the standard output.
2. `@file`: A constructor that turns a path into a streamable handle.
   ```

3. Visual logic: Think of `@` as the boundary. Anything with a `@` touches the "outside world" (the OS).

`–>` (The Pulse / Flow)

The `–>` operator is the Stream Binder. It moves data from a source to a destination.

1. `source –> sink`: Every time `source` produces a "pulse" of data, it is pushed into `sink`.
2. `"Hello World" –> @stdout` treats the string as a single–pulse stream and sends it to the console.

`|>` (The Anchor / Left Injector)

The `|>` operator is the Partial Applier. It takes a value and "anchors" it into the `left` slot of the following expression, returning a new function.

1. `("Hello " |> concat)`: This creates a new unary function that already has `"Hello "` as its `left` value. It is now waiting for a `right` value to complete the concatenation.

`o` (The Composer)

The `o` operator is the Kleisli Composer. it merges two functions or resources into a single pipeline **before** any data flows through them.

1. `h : g o f`: Creates a new transformation where the output of `f` becomes the input of `g`.

`–<>` (The Barrier / Join)

The `–<>` operator is the Synchronizer. It is used when a flow has been split (forked). It tells the runtime: "Do not proceed to the next node until **all** previous parallel branches have completed their current frame."

```rust // A flows into B and C in parallel. // –<> ensures D only starts after B and C finish. A –> [ B C ] –<> D;
```

Filosofia de Design

Link da conversa original: https://gemini.google.com/share/776a9615fd64

Princípios Core:

1. Ortogonalidade: Poucas primitivas que combinam perfeitamente.
2. Determinismo de Memória: Uso de Arenas com ciclo de vida atrelado ao escopo (Teardown).
3. Fluxo como Topologia: Sincronização via grafos (sintaxe DOT) em vez de mutexes manuais.
4. Tudo é Stream: De arquivos a relógios, tudo segue uma interface de canal/fluxo.

Estrutura da Linguagem

1. Funções Puras (Lógica)

Não possuem efeitos colaterais. Utilizam slots posicionais `left` e `right` para evitar a nomeação desnecessária de variáveis (Point−free style).

Unárias: Operam sobre `right`.

Binárias: Operam sobre `left` e `right`.

Nulária: Não possuem argumentos.

2. Recursos

Gerenciam o ciclo de vida dos recursos através do operador `@`.

Setup: Ocorre na abertura do script/expressão.

Step: Ocorre a cada execução do recurso.

Teardown: Limpeza garantida ao fim do escopo (fechar FDs, resetar Arenas).

3. Topologia de Fluxo (dot language)

Utiliza setas `->` / `-<>` para definir como os dados pulsam entre os nós. O paralelismo é implícito e gerenciado por *Backpressure*.

Primitivas Atômicas e de Sistema

Primitiva	Abstração	Papel no Runtime
—	—	—
@handle	Canal de I/O	Interface única para Arquivos e Sockets.
@mem	Canal de Espaço	Memória endereçável tratada como um stream seekable.
@signal	Canal de Evento	Gatilhos temporais (`@clock`) ou do sistema (`@metadata`).
@sys	Canal de Evento	Invocação direta de syscall (ex: read, write, open).

Decisões de Implementação

Compilador (Frontend)

Linguagem: Go (pela velocidade de desenvolvimento e facilidade com CLI/Grafos).

Estratégia de Parsing: Pratt Parser (Top−Down Operator Precedence).

Gestão de Grafos: Biblioteca `dominikbraun/graph` para validação de topologia e detecção de deadlocks.

Alvo (Backend/Runtime)

Linguagem Alvo: C99 / Zig.

Modelo de I/O:

Linux: `io_uring` para I/O assíncrono de alto desempenho.

Windows: `IOCP` (I/O Completion Ports) via Overlapped I/O.

Memória: Mapeamento direto de Arenas para `mmap` (Unix) ou `MapViewOfFile` (Windows) para persistência zero−copy.

Próximos Passos

1. Implementar o Lexer básico em Go usando `text/scanner`.
2. Estruturar a tabela de precedência para o Pratt Parser (NUD/LED).
3. Definir o Header de Runtime em C para gestão das Arenas e Slots `left` / `right`.