Mathematical Expression Language Specification

Overview

This document defines the syntax, semantics, and practical usage of a mathematical expression language designed for symbolic computation, mathematical exploration, and interactive scripting.

The language features rich support for intervals, rationals, mixed numbers, pattern-matching functions, metadata-annotated objects, robust piping and mapping, and a regular system for containers, sets, maps, and matrices.

Table of Contents

---

1. Basic Assignment, Identity, and Assertion

Operation	Syntax	Meaning
Assignment	`x := 3`	Assign value 3 to `x`
Function Def	`f := (x, n := 5) -> x^n`	Define function with default
Boolean Test	`x ?= 3`	Is x equal to 3? (boolean)
Math Assertion	`x = 3`	Assertion/identity
Solve Equation	`x :=: 3x+2`	Solve for x
Inequality	`x < 5`, `x ?< 5`	Assertion; boolean test

- `?=`, `?<`, `?>`, `?<=`, `?>=` are all boolean comparisons.
- `=`, `<`, `>`, `<=`, `>=` are mathematical assertions (identities, inequalities).

- `:=:` denotes an equation to solve.

---

2. Numbers, Intervals, and Mixed Numbers

Type	Syntax	Meaning
Integer	`42`	Integer
Fraction	`2/3`	Rational p
Decimal	`1.23`	Interval [1.225, 1.235]
Repeating Dec	`0.#3`	1/3
Interval	`2:5`	[2,5] (closed)
Mixed Number	`1..3/4`	1 + 3/4 = 1.75
Interval Exp	`2:3 ^ 2`	[4, 9] (elementwise power)
Minkowski Pow	`2:3 ** 2`	[4, 9] (interval product)

- `a:b` denotes a closed interval from a to b.
- `^` applies elementwise on interval endpoints.
- `**` is interval/Minkowski product.
- Decimal notation without `#` is interpreted as an exact number in intervals: `1.23:1.34` = `1.23#0:1.34#0`.
- Repeating decimals: `0.#3` is 1/3, `1.23#56` is 1.235656...

2.a. Units and Quantities Extension

This section specifies support for units in the Mathematical Expression Language, including notation, arithmetic, conversion, user-defined units, and error handling.

Unit Syntax

- **Attach units to numbers with tildes:**
  - `3.2~m~` → 3.2 meters
  - `9.8~m/s^2~` → 9.8 meters per second squared
  - `2~kg*m^2/s^2~` → 2 joules (if so defined)
- **Complex units:** Any valid unit string between tildes (including `/`, `^`, `*`).

Arithmetic with Units

- **Addition/Subtraction:** Only allowed with *identical* units. Attempting to add/subtract numbers with incompatible units throws an error.
- **Multiplication/Division:** Units combine naturally:
  - `3~m~ * 2~s~ = 6~m*s~`

- **Exponentiation:** Units exponentiate:
  - `2~m~ ^ 3 = 8~m^3~`
- **Simplification:** Automatic if units cancel:
  - `5~m~ / 2~m~ = 2.5` (unitless)

Unit Conversion

- **Syntax:** `value~~new_unit/old_unit~`
  - E.g., `3.2~m~ ~~mi/m~` converts meters to miles
  - Double-tilde `~~` marks the conversion operation
  - The system automatically applies known conversion factors

Defining Units

- **Built-in command:** `Unit(name, definition)`
  - Example: `Unit("ly", 9.4607e15~m~)` ly for lightyear
  - Adds a user-defined unit or alias to the system's conversion table

Sample Usage

```
a := 10~m~
b := 3~m~
a + b         // 13~m~

c := 4~s~
a / c         // 2.5~m/s~

d := 20~m/s~
d * c         // 80~m~

speed := 100~km/h~
speed~~m/s~   // Convert to m/s

e := 5~kg~
f := 20~m/s^2~
force := e * f     // 100~kg*m/s^2~

// Error: incompatible units
a + c         // Error: cannot add meters and seconds
```

Notes

- **Unit Parsing:** Supports algebraic units, e.g., `~kg*m^2/s^2~` or `~N~`

- **Conversion:** System simplifies/cancels units, recognizes canonical forms, and allows user extension
- **Units are first-class and checked at evaluation time**
- **User-defined units and conversion rules are supported**

2.b. Number System Extensions as Units

Overview Number system extensions—including complex numbers, algebraic roots, and general primitive elements—are modeled as symbolic "units" with explicit rules for arithmetic and simplification. This section explains their syntax, definition, arithmetic behavior, and conversion to real numbers.

Complex Numbers as Units

- **Syntax:**
  - `3 + 4~i~` (represents 3 + 4i)
- **Arithmetic:**
  - Addition/subtraction: combine like terms
    - `(3 + 4~i~) + (1 + 2~i~) = 4 + 6~i~`
  - Multiplication: FOIL and apply `~i*i~ = -1`
    - `(3 + 4~i~) * (1 + 2~i~) = 3*1 + 3*2~i~ + 4~i~*1 + 4~i~*2~i~ = 3 + 6~i~ + 4~i~ + 8~i*i~ = 3 + 10~i~ - 8 = -5 + 10~i~`
  - Exponentiation: Use binomial theorem and powers of `i`
- **Arithmetic rules for `i`:**
  - `~i*i~ = -1`
  - All standard complex rules apply

Algebraic Extensions as Units

- **Defining an algebraic primitive:**
  - `Primitive("sqrt2", Poly(1, 0, -2), 1:2)`
    - **Name:** `"sqrt2"`
    - **Minimal polynomial:** `Poly(1, 0, -2)` (i.e., x^2 - 2)
    - **Interval:** `1:2` (specifies which root to use)
- **Usage:**
  - `2 + 3~sqrt2~`
  - Arithmetic uses the minimal polynomial for simplification:

    * `~sqrt2~ * ~sqrt2~ = 2`
    * `(2 + 3~sqrt2~) + (1 - ~sqrt2~) = 3 + 2~sqrt2~`
    * `(2 + 3~sqrt2~) * (1 - ~sqrt2~)` expands and simplifies accordingly

- **General extensions:**
  - `Primitive("xi", Poly(1, 0, 0, -1), 0:2)` for a root of x^3 - 1

Approximate Conversion to Real Numbers

- **Syntax:**
  - `Real(2 + 3~sqrt2~, E-6)`
    * Converts the expression to a real number (float/interval) within 10^-6 accuracy.
- **Behavior:**
  - Evaluates the extension numerically to the specified precision.
  - Useful for plotting, numerical computation, etc.

Sample Usage

```
// Complex arithmetic
z := 3 + 4~i~
w := 1 + 2~i~
z + w       // 4 + 6~i~
z * w       // -5 + 10~i~

// Algebraic extension
Primitive("sqrt2", Poly(1,0,-2), 1:2)
x := 2 + 3~sqrt2~
y := 1 - ~sqrt2~
x + y       // 3 + 2~sqrt2~
x * y       // -1 + ~sqrt2~

// Real approximation
Real(x, E-6)  // Returns decimal value with 10^-6 accuracy
```

Defining General Extensions

- **Syntax:**
  - `Primitive("name", Poly(...), interval)`
    * `Primitive("sqrt3", Poly(1, 0, -3), 1:2)` defines sqrt 3.
    * `Primitive("xi", Poly(1, 0, 0, -1), 0:2)` defines a cube root of 1.

- **Usage:**
  - `a := 2 + ~xi~`
  - Arithmetic follows the defining polynomial for reduction.

Notes

- **Extensions are first-class and behave as units in arithmetic.**
- **Simplification rules** (e.g., `~i*i~ = -1`, `~sqrt2~ * ~sqrt2~ = 2`) are built-in for each primitive.
- **Approximate real evaluation** can be requested explicitly via `Real(...)`.
- **Works for both complex and algebraic extensions, even simultaneously (e.g., a + bi + csqrt2).**

Why This Approach Is Powerful

- **Unified notation:** All extensions (imaginary, roots, etc.) are syntactically "units" with symbolic rules.
- **Extensible:** Users can define new number extensions at will.
- **Symbolic/numeric bridge:** Exact forms preserved until explicit real conversion.

2.c Real Numbers as Oracles, Metadata, and Access Convention

This specification extends the mathematical language with a precise notion of real numbers as *oracles*—function objects with rich metadata and a formal rule for how properties and content are accessed using dot (`.`) and underscore (`_`). The `Primary` property, accessible via metadata, always points to the main value or function.

Oracle Structure and Behavior

Definition

- **Oracle:** A function `(a:b, delta) -> (k, c:d)` representing a real number, where:
  - `a:b`: rational interval
  - `delta`: positive rational tolerance
  - Returns:
    - `(1, c:d)` if the real is in `(a:b)[delta]`, with `c:d \subset (a:b)[delta]`
    - `(0, c:d)` if not, and `c:d` is disjoint from `a:b`
  - `c:d` always contains the real number.

Properties (Axioms)

- **Range:** Output always `(k, c:d)`
- **Separation, Disjointness, Consistency, Closedness:** As in full mathematical definition

Example: Newton's Method Oracle (with `Self` and Dot Access)

```
newton_oracle := [
  (a:b, delta) -> {
    x := (a + b) / 2;
    Self.newton := Self.newton ++ [x];
    for i := 1 to 10 {
      x := x - (f(x) / f'(x));
      Self.newton := Self.newton ++ [x];
      if abs(f(x)) < delta { break; }
    }
    c := x - delta;
    d := x + delta;
    Self.history := Self.history ++ [c:d];
    return (1, c:d);
  },
  yes := init_interval,
  type := "oracle",
  newton := [],
  history := [],
  Primary := (a:b, delta) -> { ... }
]
```

- `Self.newton`: Newton iterate sequence
- `Self.history`: prophecy intervals (intervals returned)
- `Self.Primary`: always the function itself

General Usage and Examples

```
// Metadata
oracle.newton        // Newton iterates
oracle.type          // "oracle"
oracle.history       // All prophecy intervals
oracle.Primary       // The function itself

// Content
arr_3                // 3rd element in array
map_key              // value at key in map

// Dynamic access
arr[i]               // i-th element
```

```
oracle["history"]    // metadata if dynamic access is enabled
```

Example: Rational and Algebraic Oracles

```
// Rational oracle
q := 1/3
R_q := [
  (a:b, delta) -> (q IN a:b, q:q),
  yes := q:q,
  type := "oracle",
  Primary := (a:b, delta) -> (q IN a:b, q:q)
]

// sqrt(2) oracle
Primitive("sqrt2", Poly(1,0,-2), 1:2)
sqrt2_oracle := [
  (a:b, delta ? a > 0) :=> (2 IN a^2:b^2 ? (1, a:b) : (b^2 < 2 ? (0, b:2) : (0, 0:a))),
  (a:b, delta ? b < 0) :=> (0, 0:2),
  (a:b, delta) :=> (2 IN (0:b^2) ? (1, 0:b) : (0, b:2)),
  yes := 0:2,
  type := "oracle",
  Primary := (a:b, delta) -> { ... }
]
```

Arithmetic and Decimal Output

- Arithmetic operators (`+`, `-`, `*`, `/`) extended to oracles:
  - Result is an oracle if any operand is an oracle
  - Prophecy intervals computed algorithmically
- Decimal/interval approximation:
  - `oracle.decimal(tol := 1E-6)` returns an interval with width $<$ tolerance

Summary Table

Feature	Syntax/Example	Notes
Metadata	`yes`, `type`, `history`, `newton`, `Primary`	For bookkeeping and state
`Self`	`Self.history := ...`	Metadata management
`Primary`	`oracle.Primary`	Main function/object
Content	`arr_3`, `map_key`	Array/map access
Dynamic	`arr[i]`, `oracle["history"]`	Optional

Notes

- All arithmetic, printing, and interaction with real numbers goes through oracles—no floating point unless explicitly requested

3. Containers: Arrays, Sets, Maps, Tuples, Matrices

Type	Syntax	Meaning
Array	`[1, 2, 3]`	Standard JS-style array
Set	`{3, 4, 5}`	Set (unique elements)
Map/Object	`{a := 4, b := 5}`	Map with named keys
Tuple	`(3, 5, 6)`	Tuple (function argument grouping)
Matrix	`[1, 2; 3, 4]`	2x2 matrix
3D Matrix	`[1, 2; 3, 4 ;; 5, 6; 7, 8]`	2x2x2 tensor

- `{}` must be type-homogeneous (set, map, pattern-matching array, or system).
- Mixes like `{3, a := 5}` are not allowed (throws error).
- Use `[ ... , property := value ]` to add metadata/properties to any container.

---

4. Metadata and Properties

- Any container, number, function, or object can be wrapped in `[]` with extra properties:
  - `[obj, name := "foo"]` — attaches metadata property `name` to `obj`.
- **Property Access:**
  - `obj_name` — access content property/key `name` on object
  - `obj__meta` — access metadata property `meta`
  - `obj[x]` — content property at dynamic key `x`
  - `obj[;x]` — metadata property at dynamic key `x`
- **Equality** ignores metadata: `[3, meta:=4] == 3` is `true`.

Metadata and Dot/Underscore Rule

Formal Rule

- **Dot (`.`):** Access **metadata or properties**
  - `Self.newton` (Newton sequence)
  - `oracle.type` (object type)

  - `oracle.history` (interval/prophesy log)
  - `oracle.Primary` (the main value or function)

- **Underscore (`_`):** Access **content of arrays/maps**
  - `arr_3` (third element of an array)
  - `map_key` (value in map for key "key")

- **Dynamic Access:**
  - `arr[i]` for array content
  - `oracle["history"]` for dynamic metadata (optional, if implemented)

- **`Primary` property:**
  - Every object/function with metadata has `Primary` referencing its main value/function
  - For a function with metadata, `Self.Primary` is the function itself
  - For an object, it's the main value, with other fields in metadata

`Self` Keyword

- Inside any function/object, `Self` refers to that object's metadata (including `.Primary`)
- Allows oracles and advanced functions to manage their state and properties

Notes

- `Self` and dot/underscore conventions allow elegant, extensible object state and introspection
- The `Primary` property is always present in objects/functions with metadata for clear reference to the underlying value

---

5. Function Definition and Calls

- **Function Definition:**
  - `f := (x, n := 5) -> x^n + 1`
  - First arguments are positional, named/optional follow (with defaults).
  - Arrow `->` for body.

- **Calling:**
  - `f(2)` // $2^5 + 1 = 33$
  - `f(2, n := 3)` // $2^3 + 1 = 9$

- **Function Templates:**
  - `[f, n := 7]` — returns a version of `f` where n defaults to 7.
  - `g := [f, n := 5]`; `g(2)` $\rightarrow 33$

- **Metadata Templating:**
  - Pattern-matching and function arrays can carry metadata shared by all cases, unless locally overridden.

---

6. Pattern Matching and Case Functions

- Pattern-matching array syntax:

```
g := [
  (x ? x < 0) :=> -x,
  (x) :=> x
]
g(-5)    // 5
g(2)     // 2
```

- Multiple-case/templated pattern-matching:

```
powcases := [
  (x) :=> x^n,
  (x; n := 5) :=> x^(2n),
  n := 7
]
powcases(2)            // 2^7
powcases(2, n := 5)    // 2^10
```

- Precedence for named parameters: call arg > pattern-local > metadata default.

---

7. Piping and Data Flow

- **Simple pipe (“):** Elixir-style; auto-feeds left as args (tuple unpacked):
  - `(3, 4) |> f` is `f(3, 4)`
- **Explicit pipe (“):** Requires explicit mapping:
  - `(3, 4) ||> f(_2, _1)` is `f(4, 3)`
  - Can access earlier pipeline values as `__1`, etc.
- **Arrays vs Tuples:**
  - `(a, b, c)` unpacks as multiple args
  - `[a, b, c]` passes as a single arg

---

8. Loops and Comprehensions

Syntax	Meaning
`for i := 1 to 10 { ... }`	Loop from 1 to 10
`for x in arr { ... }`	For-each over array
`while cond { ... }`	While loop
`repeat { ... } until cond`	Post-condition loop
`[f(x) for x in 1:10]`	Comprehension (array, set, etc.)

---

9. Symbolic Calculus Notation

- **Notation:**
  - `(''...)f[vars](ops)`
  - Examples:
    - `f'(x')`→df/dx
    - `''f[x, y]('y, x')`→ integrate by y, then differentiate by x
    - `f'(r')`→derivative along path r
  - Order of operations is left-to-right.
  - Integration: leading `'` before function
  - Differentiation: variable with trailing `'`

---

10. Systems of Equations & Inequalities

- Use `{ ... ; ... }` for system (semicolon separated)
- Example: `{ x^2 + y^2 :=: 1; x :=: y }`
  - Solution is tuple/array of tuples if finite, function if parametric.
  - `Solve(sys)`
- Can wrap system in `[]` to attach metadata/context.

---

11. Comments and Strings

Syntax	Meaning
`// ...`	Inline comment
`/* ... */`	Block comment

Syntax	Meaning
`/*** ... ***/`	Nestable block comment
`"string"`	Literal string
`` `Hello ${x}` ``	Interpolated string

---

12. Scope, Mutability, and Variable Rules

- **Variables are block-scoped, mutable.**
- Special scope access:
  - `@var` references variable outside block
  - `@var :@= ...` assigns at that scope
  - More `@` to climb higher scope levels
- Arrays, maps, sets, etc. are mutable by default.

---

13. Modules, Loading, and Help

- `Load("trig")` loads trig module with `SIN`, `COS`, etc.
- `Load("trig", "Tr")` loads with prefix: `TrSIN`, `TrCOS`, etc.
- `Execute(path)` runs file/script in context
- `Help` or `Help f` displays documentation

---

14. Examples and Tutorials

Variables and Numbers

```
x := 3
z := 1..2/3
I := 2:3
I ^ 2        // 4:9
```

Functions

```
f := (x, n := 5) -> x^n + 1
f(2)           // 33
f(2, n := 3)   // 9
```

Pattern-Matching Functions

```
g := [
  (x ? x < 0) :=> -x,
  (x) :=> x
]
g(-5)          // 5
g(2)           // 2
```

Metadata

```
arr := [1, 2, 3, name := "my array"]
arr__name      // "my array"
```

Sets, Arrays, Matrices

```
s := {1, 2, 3, 2, 3}
a := [1, 2, 3]
mat := [1, 2; 3, 4]
```

Piping

```
(3, 4) |> f        // f(3, 4)
(3, 4) ||> g(_2, _1) // g(4, 3)
```

Loops and Comprehensions

```
sum := 0
for i := 1 to 10 {
  sum := sum + i
}
squares := [i^2 for i in 1:10]
```

Systems

```
sys := {x^2 + y^2 :=: 1; y :=: x}
Solve(sys)
```

Symbolic Calculus

```
f'(x')               // Partial derivative w.r.t. x
''f[x, y]('y, x')    // Integrate by y, then differentiate by x
```

Notes

- This language is designed for interactive, symbolic, and mathematical computing.
- All syntax is case-insensitive except for variable/function names.
- Metadata is ignored in equality by default.
- All math is rational/interval-based for precision.
- Comments, whitespace, and newlines are freely allowed except where ambiguity would arise.

Sequence and Set Literals, Advanced Piping Operators

Sequence/Set Literal Syntax A highly expressive system for defining sequences (arrays, sets) that can be finite, infinite, recursive, filtered, generated, or stopped by custom conditions.

Operator Legend

- `|+n` *Add n to previous element (arithmetic sequence).* Example: `[2, |+2, |; 10]` → `[2, 4, 6, 8, 10]`
- `|*n` *Multiply previous element by n (geometric sequence).* Example: `[1, |*3, |; 6]` → `[1, 3, 9, 27, 81, 243]`
- `|:f` *Generator function by index (f receives the index).* Example: `[|: (i) -> i^2, |; 5]` → `[0, 1, 4, 9, 16]`
- `|>f` *Pipe previous value(s) into function f (for recursion).* Example: `[1, 1, |>(a, b) -> a + b, |; 7]` → `[1, 1, 2, 3, 5, 8, 13]` (Fibonacci)
- `|?p` *Predicate filter; keeps elements where predicate is true.* Example: `[1,2,3,4, |? (x) -> x % 2 ?= 0]` → `[2, 4]`
- `|;n` *Stop after n elements (finite sequence).* Example: `[2, |+2, |; 5]` → `[2, 4, 6, 8, 10]`
- `|;f` *Stop when function returns true (repeat-until style).* Function can take value and/or index. Example: `[2, |+2, |; (x) -> x > 10]` → `[2, 4, 6, 8, 10, 12]` (stops when x > 10)

Evaluation Rules

1. Start with explicit elements (if any).
2. Apply recursion/generation (`|+`, `|*`, `|:`, `|>`).
3. Filter with `|?`.
4. Stop with `|;` (number: after n elements, function: stop when true).
5. Operators and explicit elements separated by commas; order matters (left-to-right).

Example Usages

```
[2, |+2, |; 15]                  // [2, 4, 6, ..., 14]
[1, |*3, |; 10]                  // [1, 3, 9, 27, ...]
[1, 1, |>(a,b)->a+b, |; 7]       // [1, 1, 2, 3, 5, 8, 13]
[|: (i)->2*i, |; (x)->x>20]      // [0, 2, 4, ..., 20]
[2, 3, 4, 5, |+1, |; 10, |? (x)->x%2==0]
[2, |+2, |; (x,i)->i>=5]         // [2, 4, 6, 8, 10]
```

Piping Operators for Arrays and Sequences Piping can be used to apply functional transformations to arrays, sets, sequences, and other constructs. Each piping operator is specialized:

- `|>` *Feeds the array as arguments to a function.* Example: `[1, 2, 3] |> f` (calls `f(1, 2, 3)`)
- `|>>` *Maps function over array (like* ***`map`****).* Example: `[1, 2, 3] |>> f` → `[f(1), f(2), f(3)]`
- `|>?` *Filters array with predicate (like* ***`filter`****).* Example: `[1, 2, 3] |>? (x) -> x > 1` → `[2, 3]`
- `|>:` *Reduces array with function (like* ***`reduce`****).* Example: `[1, 2, 3] |>: (a, b) -> a + b` → `6`
- **All can be chained:** Example: `[1, 2, 3, 4, 5] |>> (x) -> x * 2 |>? (x) -> x < 7 |>: (a, b) -> a + b` (`[2, 4, 6]` summed to `12`)

Rules & Notes

- `|>` feeds the entire array/sequence as function arguments (destructures if function expects multiple args).
- `|>>` maps the function over the array (like Python's `map`).
- `|>?` filters the array (like Python's `filter`).
- `|>:` reduces/folds the array (like Python's `reduce`).
- All piping operators can be chained for expressive data pipelines:

  ```
  [1,2,3,4,5] |>> (x)->x*2 |>? (x)->x<7 |>: (a,b)->a+b   // (([2,4,6,8,10] filtered to [2
  ```

- **Generalization:** These piping operators can apply to sets, sequences, streams, etc.
- **Metadata:** If the array/set has metadata (e.g., from sequence literal syntax), it's preserved through the pipeline unless explicitly overridden.

General Notes

- Literal notation works for arrays, sets, streams, and can be used for infinite/lazy constructs (with `|;` or a predicate to make finite when desired).
- Operators separated by commas for clarity and order.
- Piping is orthogonal: any array/sequence/set can be piped to a function, mapped, filtered, or reduced.
- Filter, map, and reduce apply to other compatible constructs.
- Evaluation is lazy if generator, recursion, or stopping condition is present; otherwise, eager.

Example Combinations

```
// Map and sum all even numbers up to 20
[2, |+2, |; (x)->x>20] |>> (x)->x*2 |>: (a,b)->a+b

// First 10 squares, then filter those greater than 25
[|:(i)->i^2, |;10] |>? (x)->x>25

// Fibonacci sequence, take first value exceeding 10
[1,1,|>(a,b)->a+b,|; (x)->x>10]
```

Sequence Stopping and Generation Semantics

Stopping and Generation Operators

A dual system for sequence/array/set literal evaluation:

- **Eager/Immediate** execution with `|;`
- **Lazy/Generator** mode with `|^`

Operators (with list-style description)

- `|; n`
  - *Eager stopping after n elements.*
  - Example: `[2, |+2, |; 5]` → `[2, 4, 6, 8, 10]`
- `|; f`
  - *Eager stopping when function returns true.* Function receives `(x, i)`.
  - Example: `[2, |+2, |; (x) -> x > 10]` → `[2, 4, 6, 8, 10, 12]`
- `|^ n`
  - *Lazy/generator mode, limit to n elements if requested.*
  - Example: `[2, |+2, |^ 1000]` (generator, up to 1000 elements)

- `|^ f`
  - *Lazy/generator mode, stops generation if function returns true.* Function receives `(x, i)`.
  - Example: `[2, |+2, |^ (x, i) -> x > 100]` (generator, stops at x > 100 if accessed)
- `Self[i]`, `Self_3`
  - *Access current sequence/array within generator or stopping function for recursion/lookback.*

How To Use

- Use `|;` for full, immediate array generation up to the stop.
- Use `|^` for generator/stream semantics—sequence only produces values when accessed, up to the ceiling/limit.
- Both accept either a number or a function as argument.
- `Self[i]` and `Self_3` provide access to the current state of the sequence so far from within the function body.

Examples

Eager sequences:

```
[1, |+2, |; 5]                      // [1, 3, 5, 7, 9]
[2, |*3, |; (x) -> x > 100]         // [2, 6, 18, 54, 162]
```

Lazy/generator sequences:

```
[1, |+1, |^ 1000]                   // Up to 1000 elements on demand
[1, |+1, |^ (x, i) -> x > 100]      // Generates as needed, capped if x > 100
```

Self access inside functions:

```
[1, 1, |>(a, b) -> a + b, |; (x, i) -> i > 10] // Fibonacci, eager, 11 terms
[2, |+2, |^ (x, i) -> i > 100 && Self[i-1] % 5 == 0] // Lazy, stops if last is multiple of 5
```

Design Rationale

- `|;` is for eager (immediate) evaluation—builds the entire sequence now.
- `|^` is for generator/lazy evaluation—produces values on demand, never exceeds the limit/condition.
- Both types are essential for different programming and mathematical needs.
- All other sequence operators (`|+`, `|*`, `|:`, `|>`, `|?`, etc) are compatible and compose with these.

Summary

- Use `|;` to force immediate, finite arrays.
- Use `|^` to create safe, lazy generators for infinite or large sequences.
- Both support custom logic via `(x, i)` functions.
- `Self[i]` and `Self_3` are available for advanced recursion or context-sensitive generation.

Double Quote String Literal Specification

Rule Overview

- **Delimiter:**
  - String literals are enclosed by N consecutive double quotes (", "", """, etc.), for any N >= 1.
  - The string starts after the opening N quotes and ends at the next N consecutive double quotes.
- **Literal Content:**
  - Any sequence of M consecutive double quotes where M < N is preserved literally within the string.
  - No run of double quotes is collapsed or reduced inside the string—**all are kept as written**.
- **Trimming (Optional):**
  - If there are spaces between the opening delimiter and the first non-space character, or between the last non-space character and the closing delimiter,one space is trimmed from the resulting string.

Examples

Input	Result	Notes
`""Hi "Jake"!""`	`Hi "Jake"!`	2-quote delimiter, single quote is literal
`"""She said: ""Hello""."""`	`She said: ""Hello"".`	3-quote delimiter, 2-quotes preserved
`""""Wow,"" he said."""`	`Wow,"" he said.`	4-quote delimiter, 2-quotes preserved
`""  "Hi"  ""`	`"Hi"`	Spaces trimmed at both ends
`"""  ""Hello""  """`	`""Hello""`	3-quote delimiter, 2-quotes preserved, spaces trimmed

---

Parsing Algorithm

1. Count N consecutive quotes at the start.
2. Read all subsequent characters as the content of the string.
3. The first occurrence of N consecutive quotes marks the end of the string.
4. If there are spaces directly inside the delimiter at either end with a quote at the other side, one space is trimmed from the result.
5. All internal quote runs of fewer than N quotes are kept as written.

Advantages

- **No escapes needed for quotes:** Use a delimiter longer than any run inside.
- **Literal clarity:** No collapsing or ambiguity; what you see is what you get.
- **Flexible for any content:** Write as many quotes as you like inside the string by picking a longer delimiter.
- **Spaces at the ends:** Optionally remove unwanted spaces with intuitive syntax.

Braces and Code Block Disambiguation

Rule Summary

- `{ ... }`
  - Used for sets, maps, or system constructs.
  - Example: `{3}` is a set containing 3.
  - Example: `{a := 4, b := 5}` is a map.
- `{{ ... }}`
  - Used for code blocks that can be assigned, passed as values, or returned from functions.
  - No space is allowed between the two opening braces.
  - Example: `{{ 3 }}` is a code block whose value is 3.
- `{ { ... } }`
  - A set containing a set (or other set construct).
  - There must be a space after the opening brace to distinguish from a code block.
  - Example: `{ {3} }` is a set whose single element is the set `{3}`.

Clarification Table

Syntax	Meaning	Notes
`{3}`	Set containing 3	
`{{3}}`	Code block, result is 3	No space between braces
`{ {3} }`	Set containing `{3}`	Space after first brace required
`{{3; 4}}`	Code block, result is 4	Multiple statements, result is last
`{ {3}, 4 }`	Set containing `{3}` and 4	

Parsing and Style Guidance

- **Double braces (`{{ ... }}`) without a space always indicate a code block.**
- **Single braces are never a code block.**
- **A space between `{` and `{` (e.g., `{ { ... } }`) always means a set containing a set (or similar construct).**
- Code blocks should return the value of their last statement.
- Code blocks can be passed to functions (e.g., for flow control or as arguments).
- Parsers may issue a warning if ambiguous or unintended spacing is detected after an opening brace.

Examples

```
foo := {{ x := 2; y := 3; x + y; }}        // code block assigned to foo, returns 5
bar := {3, 4, 5}                           // set containing 3, 4, and 5
nested := { {3} }                          // set containing the set {3}
If(a ?= 2, {{ y := 7; x + y; }}, {{ x := 9; y := 1; x * y; }})  // code blocks as branches
```