ChatGPT

MeTTa and OpenCog Hyperon Guide

Introduction: OpenCog Hyperon is a next-generation cognitive framework, and **MeTTa (Meta Type Talk)** is its core programming language [1]. MeTTa is often described as a meta-programming or "language of thought" for AGI: it lets programs introspect and modify themselves, blend symbolic and sub-symbolic reasoning, and operate directly on rich knowledge graphs [2] [3]. The language was designed from the ground up to support declarative, functional, and logical computation over a central **Atomspace** hypergraph (a directed knowledge graph) [4] [3]. This guide introduces MeTTa for beginners, starting from functional programming basics, then covering MeTTa's unique features, non-determinism, and examples of use.

Functional Programming Primer

Functional programming builds programs by *applying and composing functions*, rather than by giving an explicit sequence of commands [5]. In this paradigm, **functions are first-class citizens**: you can pass functions as arguments, return them from other functions, or bind them to variables, just like any data [6]. Pure functions, which have no side effects (they don't modify external state) and always return the same output for the same inputs, are emphasized [7]. State is often immutable – instead of updating variables, functions create and return new values. This leads to code that is more predictable and easier to reason about. For example, summing a list of numbers in a functional style would use a function that returns a new total rather than looping with a changing counter.

Key points of functional style:
- **Immutability:** Data structures don't change; functions produce new outputs without side effects.
- **First-class functions:** You can assign functions to variables, pass them as arguments, and return them from other functions [6].
- **Composition and higher-order functions:** Common operations like `map`, `filter`, and `fold` are used to process lists. These functions take other functions as inputs to transform data. For example, `(map (lambda ($x) (+ $x 1)) (1 2 3))` would yield `(2 3 4)`, applying the increment function to each list element.
- **Declarative style:** You describe *what* to compute (e.g. "sum these values") rather than *how* to loop and accumulate. A pure function always yields the same result for given inputs [7], so you can reason about code modularly without worrying about hidden state.

These concepts underlie MeTTa's approach. In fact, MeTTa is very much a functional-language environment, extended with logic pattern-matching and graph-based data.

`let` and `let*` Constructs

In many functional languages (especially Lisps and Schemes), `let` expressions allow you to bind local variables. A `let` introduces a new scope where you give names to values. For example, in Scheme-style syntax:

```
(let ((x 3) (y 4))
  (+ x y))  ; yields 7, with x=3 and y=4 only inside this let
```

Here, `x` and `y` are both bound *in parallel* to 3 and 4, respectively. The result of `(+ x y)` is 7 within that local scope, and `x` / `y` disappear afterward. The bindings in a plain `let` happen simultaneously, so none of the right-hand sides can refer to the new names being defined.

By contrast, `let*` performs bindings **sequentially**. Each binding can use the variables defined by earlier bindings. For example:

```
(let* ((x 2)
       (y (* x 3)))   ; here x is already 2
  (+ x y))           ; yields 2 + 6 = 8
```

In this `let*`, `x` is first bound to 2, and then `y` is bound to `(* x 3)`, using the just-bound `x`. The result is 8. If we had used a plain `let` instead, the reference to `x` inside the definition of `y` would not see the new `x`, causing an error or undefined behavior. In practice, use `let` when all bindings are independent, and `let*` when later bindings need earlier ones. This keeps code flatter by avoiding nested scopes and lets you name intermediate results step by step.

The MeTTa Language

MeTTa is a novel **multi-paradigm** language built for knowledge-centric AI [8] [9]. It combines functional style, logical pattern matching, and probabilistic constructs, all operating over a unified **Atomspace** graph. The Atomspace is a **directed hypergraph**: every piece of data or code is an *atom* (a node or a link) in a graph [10]. This means, for example, that a symbolic term like `(Add 2 3)` is represented as a Link atom “Add” connected to the Number atoms 2 and 3. Code and data share the same graph structure, allowing introspection. Key features of MeTTa include:

- **Knowledge-Graph Integration:** MeTTa programs run *inside* the Atomspace graph, not on separate memory. All values (symbols, numbers, strings, tuples) are graph atoms. The evaluator queries and rewrites the graph during execution [10]. This makes MeTTa ideal for AI applications that require representing and manipulating rich, structured knowledge [10] [9]. For instance, you can encode not just primitive values but also complex relationships or concepts as subgraphs.

  ```
  graph TD
    subgraph Atomspace
      A(Symbol node "Add")
      B(Number node 2)
      C(Number node 3)
    end
    A --> B
    A --> C
  ```

  Example: An Atomspace with a Link atom `Add` connected to two number atoms (2 and 3). This could represent the expression `(Add 2 3)` all within the knowledge graph.

- **Functional & Logical Paradigm:** MeTTa supports functional operations like arithmetic or recursion, **and** logical pattern matching in the same language. You define functions/rules using `=` and patterns (similar to Prolog clauses). For example, to define an "ancestor" relation, you can write:

```
(= (parent $X $Y) (inheritance $X $Y))
(= (ancestor $X $Y) (parent $X $Y))                              ; base
case
(= (ancestor $X $Y) (parent $X $Z) (ancestor $Z $Y))           ;
recursive case
```

  These pattern-based rules say that "X is a parent of Y" if the graph has `(inheritance X Y)`, and "X is an ancestor of Y" if X is a parent of Y *or* X is a parent of Z and Z is an ancestor of Y. The pattern variables like `$X, $Y, $Z` are unified against atoms in the graph. This declarative approach means MeTTa handles the search and matching for you.

- **Self-Modification:** MeTTa is self-reflective. Programs can generate and modify MeTTa code at runtime [2]. Because code is just data in the Atomspace, a MeTTa rule can assert new atoms or rewrite existing ones. This makes it straightforward to implement learning or meta-programming: for example, a rule could detect a new pattern and add a new inference rule to the graph [2] [10].

- **Gradual Dependent Type System:** MeTTa has an expressive type system with *gradual, dependent types* [11]. You can specify types like `(-> Number Number)` for functions, and use library functions (`is-function`, `type-cast`, etc.) to check or enforce types. This is advanced, but beginners can mostly work without explicit types. The type system provides safety (catching mismatches) when used.

- **Grounded Atoms / Neural Integration:** MeTTa supports *grounded atoms*, which are links in the Atomspace bound to external computations or data [12]. For example, a grounded atom could wrap a Python function or a neural network evaluator. This allows seamless neuro-symbolic reasoning: you can call a pre-trained neural net from inside MeTTa and then use the result in symbolic rules. For instance, a grounded atom `CatRecognizer` might take image data and return a confidence score; MeTTa rules can then pattern-match on that score.

- **Non-Determinism:** Uniquely, MeTTa functions can produce multiple possible results [13]. Instead of returning a single value, a function might nondeterministically yield a set of alternatives. Special built-in constructs `superpose`, `collapse` (and their `*-bind` variants) let you explore these branches. This effectively turns MeTTa into a search or inference engine, useful for things like combinatorics or probabilistic logic [13] (see next section).

Overall, MeTTa is essentially a "declarative knowledge-language" where you describe rules and let the engine handle matching, branching, and graph operations [13] [10]. It is sometimes called "Atomese 2" as a successor to OpenCog's Atomese [1]. Its design prioritizes concise knowledge representation and reasoning over traditional imperative control flow.

Non-Determinism in MeTTa

In most programming languages, a function returns exactly one output for given inputs. **MeTTa relaxes this:** a function can return *multiple* possible outcomes, representing a nondeterministic choice [13]. For example, if you write `(= (pick-one) (superpose (1 2 3)))`, then `!(pick-one)` will nondeterministically return 1, 2, or 3. In effect, each execution branch can pick a different result. Importantly, the *values* are fixed (1, 2, 3), but the *order* or *branching* is nondeterministic. One run might yield 2 first, another 1 first, etc.; MeTTa does not guarantee any order or priority.

To work with nondeterministic results, MeTTa provides:

- `superpose`: Converts a tuple into a nondeterministic result. For example, `!(superpose (A B C))` produces three branches: one yields `A`, another `B`, another `C` [14]. Each branch can continue independently.
- `collapse`: Gathers all alternatives back into one concrete tuple. For instance, `!(collapse (superpose (A B C)))` returns the single tuple `(A B C)` [15], collecting every branch. Use `collapse` when you want a deterministic summary of all possibilities.
- `collapse-bind`: When a MeTTa operation yields multiple bindings (e.g. different variable assignments), `collapse-bind` runs the operation and returns *all* results with their variable bindings. In notation, it returns a nondeterministic list of `(Value Bindings)` pairs. For example:

  ```
  (= (bin) 0)
  (= (bin) 1)
  !(collapse-bind (bin))  ; yields two branches: (0 {}) and (1 {})
  ```

  Here `(bin)` can yield 0 or 1; `collapse-bind` produces both, pairing each value with its (empty) binding map [16].
- `superpose-bind`: Takes the output of `collapse-bind` (or any sequence of `(Value Bindings)` pairs) and performs a `superpose` on the values only, discarding the binding information. In effect, it converts the result of `collapse-bind` into a plain nondeterministic value list. For example:

  ```
  !(superpose-bind ((A {…}) (B {…})))  ; equivalent to (superpose (A B))
  ```

  So `superpose-bind` yields the atoms `A`, `B` nondeterministically, without the binding maps [17].

In practice, `superpose-bind` and `collapse-bind` are used together when writing MeTTa code that needs to preserve or examine variable bindings across branches. You often `collapse-bind` the top-level query to see all results with context, and use `superpose-bind` internally when applying nondeterminism under existing bindings. Think of `superpose`/`collapse` as working with pure values, while `superpose-bind`/`collapse-bind` handle value-plus-binding pairs.

Because of nondeterminism, you can represent searches or probabilistic choices compactly. For example, to solve a quadratic equation nondeterministically:

```
(= (solve-quadratic $a $b $c)
   (let* ((D (- (* $b $b) (* 4 $a $c))))
     (if (< D 0)
         (superpose)  ; yields no result if discriminant is negative
         (superpose
           ((/ (+ (- $b)  (sqrt-math D)) (* 2 $a))
            (/ (- (- $b)  (sqrt-math D)) (* 2 $a)))))))
```

Calling `!(collapse (solve-quadratic 1 -3 2))` will yield `(1 2)`, the two real roots of x^2-3x+2, collected into a tuple. Here `solve-quadratic` nondeterministically branches on the two solutions, and `collapse` gathers them.

Atomspace and Hyperon Integration

All MeTTa execution occurs within the **Atomspace**, Hyperon's central knowledge store. The Atomspace is a (distributed) directed hypergraph for representing atoms and their links [10]. Hyperon's *Distributed Atomspace (DAS)* scales this graph across machines, but MeTTa abstracts it so you use the same operations. When you define a MeTTa function, you are actually adding an **EqualityLink** atom to the Atomspace: its head is the function name and its body is the expression graph. Running a function performs pattern matching and graph rewrites in the Atomspace [10].

You manipulate the Atomspace directly via built-in grounded atoms:
- `add-atom`, `remove-atom`: assert or retract atoms in the graph.
- `match`: pattern-match a template against atoms in the Atomspace. For example:

```
(!(match ((parent John $X)) (println! "John has a child!") "No match"))
```

This looks for any atom `(parent John $X)`. If found, it runs the "then" branch (e.g. printing a message). Variables like `$X` get bound to actual values from the graph if a match is found.
- `for-each-in-atom`, `get-atoms`: iterate over or retrieve sets of atoms matching certain criteria.

In Hyperon, the Distributed Atomspace handles the complexity of sharding, replication, and persistence. You can trust that MeTTa's graph operations will work over large, decentralized datasets. In summary, MeTTa tightly couples with Hyperon's Atomspace: you write MeTTa code to query and rewrite the graph, and DAS takes care of the scale behind the scenes [10] [9].

Core Data Types in MeTTa

Everything in MeTTa is ultimately an atom in the graph. Basic atomic types include symbols (e.g. `Alice`), numbers (e.g. `42`), and strings (e.g. `"hello"`). Compound structures are built with **AtomLists** (tuples) and Links. For example, `(Add 2 3)` is a tuple where `Add` is linked to `2` and `3`. You can also form **Grounded atoms** which contain arbitrary external data or procedures; for instance, a grounded atom might hold a Python object or invoke a neural network.

The standard library provides predicates for types: e.g. `(is-function X)` checks if `X` is a function, or `(type-cast X T)` to enforce a type. MeTTa supports function types explicitly: you can annotate a function as `(: myFunc (-> Number Number))`. Usually you don't need to write many types, but they help catch errors. At runtime, MeTTa keeps track of types, and mismatches will cause evaluation to fail or error.

Important built-in data types and functions include:
- **Numbers and math:** Infix arithmetic `(+ 2 3)`, `(* 5 6)`, etc., along with functions like `sqrt-math`, `pow-math`, `sin-math`. These return Atom numbers.
- **Tuples and lists:** A tuple `(1 2 3)` is an atom list. Use `map-atom`, `filter-atom`, `foldl-atom` to process tuples without manual recursion. For example, `(map-atom (lambda ($x) (* $x 2)) (1 2 3))` yields `(2 4 6)`. The `lambda` form defines an anonymous function. There are also set-like ops: `unique-atom`, `union-atom`, etc., which take *concrete* tuples.
- **Grounded atoms:** These are special atoms whose semantics is defined externally (in Python/C++). For example, one can register a Python function under a name and then call it as `(call-py fib $n)` in MeTTa. The Python API (see below) wraps results in `Atom` objects.
- **Quoting:** With `quote` and `unquote`, you can treat MeTTa code as data. For instance, `'(A B)` yields the literal tuple `(A B)` without evaluating `A` or `B`. This is useful for programmatically generating rules.
- **Error and control:** Constructs like `if`, `case`, `chain`, and error handlers (`if-error`, `return-on-error`) are available. For example, `(if (< $x 0) (println "Neg") (println "Pos"))` chooses a branch based on a condition.

In short, MeTTa's "values" are atoms in the Atomspace, which can be simple or complex. The evaluator manages types behind the scenes, so when writing code you typically manipulate atoms (using the above functions) and let MeTTa check consistency.

MeTTa Standard Library Highlights

The MeTTa standard library provides many built-in functions as grounded atoms. Some key ones for beginners:
- **Equality/Reduction (`=`):** The `=` construct itself defines rules/functions. For example, `(= (double $x) (* 2 $x))` defines a function `double`. There are identity helpers like `id`, `assertEqual`, etc., but most code just uses `=` for definitions.
- **Math:** Besides `+ - * /`, use `pow-math`, `sqrt-math`, etc. For example, `(* 2 3)` yields 6. For integer rounding: `floor-math`, `ceil-math`, `round-math`.
- **Nondeterminism:** `superpose`, `collapse` and their `-bind` variants manage multiple results [14] [15]. Use `superpose` to branch, and `collapse` to collect all branches. For example, `!(superpose (1 2 3))` returns one of 1, 2, or 3; `!(collapse (superpose (1 2 3)))` returns `(1 2 3)`.
- **List/Tuple Ops:** `map-atom`, `filter-atom`, `foldl-atom` operate on tuples of atoms. For instance, `(map-atom (lambda ($x) (+ $x 1)) (1 2 3))` gives `(2 3 4)`. Use `unique`, `union`, `intersection`, etc., on nondeterministic inputs (with results in a list), and the `-atom` versions on concrete tuples. For example, `!(unique (superpose (a b b c)))` returns `[a, b, c]`, whereas `!(unique-atom (a b b c))` returns `(a b c)` [18] [19].
- **Atomspace Interaction:** Besides `match` (see above), functions like `add-atom`, `remove-atom`, `get-atoms`, `for-each-in-atom` let you work with the Atomspace. These are crucial for building or querying knowledge.
- **Control Flow:** Besides `if`, there is `case`, `chain`, etc., for conditional and sequential logic. Debugging tools include `println!` (prints an atom immediately when the code is run) and error

handlers. Note: `println!` produces output at evaluation time (not returned as a function result), so use it sparingly for debugging.
- **Quoting:** `quote` (or a leading single-quote) prevents evaluation. E.g. `'(parent John Alice)` produces the literal pattern `(parent John Alice)` as data. Use quoting when you want to manipulate code itself rather than run it.

As a new user, focus on `=`, arithmetic, `superpose`/`collapse`, and basic list ops. These cover a lot of use cases.

Recursion and Iteration in MeTTa

MeTTa is a functional language, so it uses **recursion** instead of conventional loops. Here are common recursion patterns:

- **Direct recursion:** Define a base case and a recursive case using multiple `=` rules. Example (factorial):

  ```
  (= (fact 0) 1)
  (= (fact $n) (* $n (fact (- $n 1))))
  ```

  Calling `!(fact 5)` will recursively compute `fact 4`, `fact 3`, ... until reaching the base case `fact 0 = 1`.

- **Pattern recursion:** You can write multiple patterns for the same function, as shown above with `ancestor`. This is similar to Prolog. For example:

  ```
  (= (ancestor $X $Y) (parent $X $Y))                    ; base
  (= (ancestor $X $Y) (parent $X $Z) (ancestor $Z $Y))   ; recursive
  ```

  MeTTa will try to match the first rule, and if not directly applicable, it will try the second, chaining the recursion.

- **Tail recursion:** If the recursive call is the last operation, MeTTa (like Lisp) can optimize it and reuse the stack, effectively acting like a loop. You can write tail-recursive versions by carrying along an accumulator.

- **Higher-order recursion:** You can often replace manual recursion over lists with `map-atom`, `filter-atom`, or `foldl-atom` (these are recursive in the library). For example, instead of writing your own loop to increment every element, just use `map-atom`. Fold operations like `foldl-atom` (left fold) can replace many accumulator-style recursions.

Always include a terminating base case to avoid infinite recursion. If a branch has no valid matches, MeTTa simply fails that branch. You can use `collapse` to gather all results of a search. For example, to list all solutions of an ancestor query: `!(collapse (ancestor John $Who))` might yield `(Mary Bob Alice Carol)`, all descendants of John.

Best Practices and Tips

- **Manage nondeterminism:** Uncontrolled nondeterminism can explode the search space. Use `collapse` when you want to stop forking and collect results. For instance, after a nondeterministic query, wrap it with `collapse` to get one tuple of answers.
- **Use library functions:** MeTTa's standard library is rich. Before writing complex recursion, check if a built-in (like `map-atom`, `filter-atom`, `unique-atom`, math functions, etc.) can do it. These are optimized and save time.
- **Ground heavy computations:** If some work is complex (e.g. heavy math or machine learning inference), implement it as a grounded atom in Python/C++ rather than in pure MeTTa rules. Pattern matching in MeTTa has overhead, so offloading heavy lifting can improve performance.
- **Quoting:** Only use `quote` when you need to treat code as data (e.g. generating rules on-the-fly). Unquoted symbols are treated as pattern variables or looked up as atoms. Remember the difference.
- **Debugging:** You can sprinkle `println!` calls in your code to output atoms during execution (since MeTTa prints them immediately). For example, `(!(println! "Debug: " $x))`. Also use `if-error` or `return-on-error` to catch unexpected cases. The `metta` CLI can often run in verbose mode for more logs.
- **Modularity:** Break logic into small named functions (each introduced with `=`). Give meaningful names and, if helpful, add type annotations (`(: funcName (-> Number Number))`). Each rule is independent, which makes reasoning about each case easier.

Installation and Setup

MeTTa comes bundled with Hyperon. To install locally, ensure you have Python 3.8+ and use pip:

```
pip install hyperon
```

This installs the `hyperon` Python package (which includes MeTTa). It adds a command-line tool `metta`. You can run a MeTTa program file like:

```
metta myprogram.metta
```

This will load `myprogram.metta` and evaluate any expressions it contains. For example, if `myprogram.metta` defines `(= (foo) 42)`, then invoking `metta myprogram.metta !(foo)` will print `42`.

Behind the scenes, Hyperon is still in pre-alpha, but works on all major platforms. For advanced use (or contributing), check the Hyperon GitHub and documentation for any platform-specific notes.

Using MeTTa from Python

Since Hyperon is a Python library, you can embed MeTTa inside Python code. For example:

```
from hyperon import MeTTa, Atom

mt = MeTTa()                    # Create MeTTa instance
```

```
mt.eval('(= (fib $n) (call-py fib $n))')  # Define fib in MeTTa to call
Python fib
# Define a Python function fib (in the same module):
def fib(n):
    if n < 2: return n
    return fib(n-1) + fib(n-2)
mt.register("fib", fib)          # Register Python function under name 'fib'
result = mt.call("fib", [Atom(6)])  # Call (fib 6) in MeTTa
print(result)  # should be Atom(8)
```

This shows how you can define MeTTa rules, register Python functions as callable, and evaluate MeTTa functions. The `Atom` class wraps values. You can also add atoms, query, and manipulate the Atomspace from Python. For more details, see the MeTTa tutorials (e.g. the Metta training repository has examples) and the Hyperon documentation.

Example Projects

Project 1: Family Heritage Tracker

Goal: Build a simple family-tree program in MeTTa to practice facts, rules, and recursion.

1. **Define parent-child facts:** In a `.metta` file, add atoms for each parent-child pair:

   ```
   (inheritance Mary Alice)
   (inheritance John Mary)
   (inheritance John Bob)
   (inheritance Bob Carol)
   ```

   Each `(inheritance P C)` atom says P is a parent of C. These are just data atoms in the Atomspace.

2. **Parent rule:** Define a rule so `(parent X Y)` matches when `inheritance(X, Y)` exists:

   ```
   (= (parent $X $Y) (inheritance $X $Y))
   ```

   Now `(parent John Mary)` will succeed because `(inheritance John Mary)` is in the graph.

3. **Ancestor rule (recursive):** Using two rules:

   ```
   (= (ancestor $X $Y) (parent $X $Y))       ; base case
   (= (ancestor $X $Y) (parent $X $Z) (ancestor $Z $Y))
   ```

   The first says “X is an ancestor of Y if X is a parent of Y.” The second says “X is an ancestor of Y if X is a parent of Z and Z is an ancestor of Y.” Together they recursively capture all descendants.

4. **Queries:** To find all descendants of `John`, you can query with collapse:

   ```
   !(collapse (ancestor John $Who))
   ```

   This will return a tuple of all individuals who are descendants of John (e.g. `(Mary Bob Alice Carol)`). A simple boolean check `!(ancestor John Alice)` will succeed (implicitly returning `()` for true).

5. **Explanation:** Each `=` definition is a rule pattern. Variables `$X, $Y, $Z` match actual atoms. This is analogous to Prolog clauses. When you run a query, MeTTa unifies patterns against the Atomspace.

```
graph LR
  John --> Mary
  Mary --> Alice
  John --> Bob
  Bob --> Carol
```

Visualization: Each arrow (e.g. `John → Mary`) represents an `(inheritance John Mary)` atom in the Atomspace.

This project teaches basic MeTTa syntax (`=` rules), asserting facts, and recursion. You can extend it by adding more families or attributes. For instance, write a function to list all ancestors of someone, or check if any ancestor satisfies a condition (using `collapse` to collect results).

Project 2: Nondeterministic Math and Python Integration

Goal: Use MeTTa's nondeterminism and Python interop.

1. **Nondeterministic choice:** Define a function that picks a number from 1–5 at random:

   ```
   (= (pick-one) (superpose (1 2 3 4 5)))
   ```

   Running `!(pick-one)` will return one of 1–5 nondeterministically. To get all options, use collapse:

   ```
   !(collapse (pick-one))  ; yields (1 2 3 4 5)
   ```

2. **Math functions:** For example, define a square-root function:

   ```
   (= (mysqrt $x) (sqrt-math $x))
   ```

   Then `!(mysqrt 16)` returns `4`. MeTTa also has `pow-math`, `log-math`, etc.

Example: Solve a quadratic deterministically:

```
(= (solve-quadratic $a $b $c)
   (let* ((D (- (* $b $b) (* 4 $a $c))))
     (if (< D 0)
         (superpose)   ; no solutions
         (superpose
           ((/ (+ (- $b)  (sqrt-math D)) (* 2 $a))
            (/ (- (- $b)  (sqrt-math D)) (* 2 $a)))))))
```

Then `!(collapse (solve-quadratic 1 -3 2))` yields `(1 2)`, the two real roots of x^2-3x+2.

1. **Python function call:** Suppose you have a Python Fibonacci function in `fib.py`:

   ```
   def fib(n):
       if n < 2: return n
       return fib(n-1) + fib(n-2)
   ```

   In Python, you can integrate it with MeTTa:

   ```
   from hyperon import MeTTa, Atom

   mt = MeTTa()
   mt.register("fib", fib)   # register the Python function under the name
   "fib"
   mt.eval('(= (fib $n) (call-py fib $n))')  # Define MeTTa rule to call it
   result = mt.call("fib", [Atom(6)])       # invoke (fib 6)
   print(result)  # prints Atom(8), since fib(6)=8
   ```

   This demonstrates calling external code from MeTTa. You can similarly register neural network classes or other heavy computations as grounded atoms.

2. **Advanced math example:** Combine nondeterminism and math. For instance, find Pythagorean triples:

   ```
   (= (pythagorean-triple)
      (superpose
        ((a b c)
         (= (+ (* a a) (* b b)) (* c c))
         (> a 0) (> b 0) (> c 0))))
   ```

   Running `!(collapse (pythagorean-triple))` will nondeterministically search for `(a, b, c)` satisfying $a^2 + b^2 = c^2$, yielding tuples like `(3 4 5)`, `(6 8 10)`, etc., if explored sufficiently.

```
flowchart LR
  Start --> Choice1[Branch A=3,B=4]
  Start --> Choice2[Branch A=6,B=8]
  Choice1 --> Check[(A^2 + B^2 == C^2)?]
  Choice2 --> Check
  Check --> Valid[Collect (3,4,5) etc]
```

Flowchart: MeTTa explores different branches (assigning values to variables) and checks the condition. The `superpose` operator creates parallel branches, and `collapse` merges them.

This project shows how MeTTa can handle multiple possibilities in parallel and leverage Python for heavy work. It illustrates using `superpose`/`collapse`, built-in math, and the mechanism for calling Python functions.

Project 3: Neuro-Symbolic Integration Demo

Goal: Combine symbolic rules with a simple neural component.

1. **Define a grounded neural atom:** In Python, you might create a class inheriting from `GroundedAtom`. For example:

   ```
   from hyperon import MeTTa, GroundedAtom, Atom

   class CatRecognizer(GroundedAtom):
       def __call__(self, inputs):
           image_atom = inputs[0]
           # Imagine processing image_atom.pyval() with a neural net; here
   we fake it
           confidence = 0.8  # pretend 80% confident it's a cat
           return Atom(confidence)  # return as an Atom
   mt = MeTTa()
   mt.register_grounded_atom("CatRecognizer", CatRecognizer)
   ```

   Now `(CatRecognizer $img)` in MeTTa will invoke the Python code and return an `Atom` containing 0.8.

2. **MeTTa rules using the classifier:** In a `.metta` file, write rules that use this:

   ```
   (= (is-cat $img) (CatRecognizer $img))
   (= (respond-if-cat $img $msg)
      (is-cat $img)
      (if (> (CatRecognizer $img) 0.5)
          $msg
          ""))  ; returns $msg only if confidence > 0.5
   ```

   The first rule wraps the grounded classifier. The second says: if the image is a cat (confidence > 0.5), then respond with `$msg`; otherwise return an empty string.

3. **Symbolic inference:** Suppose we have a symbolic fact like `(responds Alice $greeting)`. We can combine it with the image classification:

```
(= (interaction $person $img)
   (is-cat $img)
   (responds $person (respond-if-cat $img "I see a cat!")))
```

   If Alice shows an image and `CatRecognizer` is triggered, the rule chain will eventually produce the message "I see a cat!" for Alice.

This example shows how MeTTa can incorporate neural (sub-symbolic) results into symbolic logic. The grounded atom runs the neural network (or here a stub returning 0.8), and MeTTa pattern matching and conditionals decide how to respond based on that result. MeTTa thus bridges raw data processing (neural nets) and high-level reasoning in one framework [3].

Sources

The above explanations are synthesized from the latest MeTTa and Hyperon documentation. Key references include the official MeTTa docs and announcements by the OpenCog/SingularityNET team [10] [3] [20], as well as the MeTTa Standard Library guide [21] [14]. Installation details are drawn from the Hyperon (MeTTa) PyPI page [20]. These sources outline MeTTa's design philosophy (knowledge-graph integration, self-modifying code, nondeterminism, etc.) and provide the primitives summarized above. Each quoted statement is cited from the relevant documentation.

---

[1] [20] hyperon·PyPI
https://pypi.org/project/hyperon/

[2] [3] [8] [11] [12] [13] MeTTa Programming Language - ASI | Artificial Superintelligence Alliance
https://superintelligence.io/portfolio/metta-programming-language/

[4] [9] [10] MeTTa in a Nutshell: Exploring the Language of AGI | by SingularityNET | SingularityNET | Medium
https://medium.com/singularitynet/metta-in-a-nutshell-exploring-the-language-of-agi-8d344c15b573

[5] [6] [7] Functional programming - Wikipedia
https://en.wikipedia.org/wiki/Functional_programming

[14] [15] [16] [17] [21] Non-deterministic Computation — MeTTa Standard Library 0.1 documentation
https://metta-stdlib.readthedocs.io/en/latest/non_deterministic_computation.html

[18] [19] Set Operations — MeTTa Standard Library 0.1 documentation
https://metta-stdlib.readthedocs.io/en/latest/set_operations.html