MeTTa Lecture Notes

📘 MeTTa Lecture Notes (Detailed)

1. 🌐 What is Atom Space?

- Think of **Atom Space** as a **knowledge graph database**.
- It stores **atoms**, which are the smallest units of knowledge.
- Atoms can be:
  - **Facts** → `(mike 25 2005 Male)`
  - **Relations** → `(john isParentOf pkm)`
  - **Nested structures** → `((pkm madharia) 60 2008 Female)`

👉 Imagine Atom Space like a **notebook** where you write down small knowledge facts. Later, you can **query** (ask questions) and **derive new knowledge** from them.

---

2. 📝 Storing Data in Atoms

Example 1: Simple Records

```
(pkm 23 2004 Male)
(mike 25 2005 Male)
(oo 25 2005 Male)
(john 26 2006 Female)
```

- General pattern:
  `(Name Age BirthYear Gender)`
- Each line is one **atom** in the database.

Example 2: Nested Records

```
((pkm madharia) 60 2008 Female)
```

- Here, `(pkm madharia)` is treated as a **compound entity**.
- Useful if names or identifiers have more than one part.

Example 3: Relationships (Graph Facts)

```
(john isParentOf pkm)
(mike isParentOf pkm)

(john isParentOf oo)
(mike isParentOf oo)
```

- This is where **graph structure** appears:
  - `john → pkm`
  - `mike → pkm`
  - `john → oo`
  - `mike → oo`

👉 We're slowly building a **family tree** inside Atom Space.

3. 🔍 Retrieving Data with `match`

How `match` works

```
(match Database Pattern Output)
```

- **Database** → usually `&self` (the current Atom Space).
- **Pattern** → what to look for (like a search filter).
- **Output** → what to return if pattern matches.

💡 Think of `match` like **SQL SELECT**, but functional and graph-based.

Example 1: Query pkm's details

```
!(match &self (pkm $x $y $z) ("EVERYthing is " $x))
```

- Pattern: `(pkm $x $y $z)` → match any record starting with `pkm`.
- Output: `"EVERYthing is " $x` → return text with `$x` (age).

✅ Output → `[("EVERYthing is " 23)]`

Example 2: Query by age

```
!(match &self (pkm 25 $y $z)
    ("user of name pkm with age 25 is" $y "year of birth"))
```

✅ Output → `[("user of name pkm with age 25 is" 2004 "year of birth")]`

Example 3: Find pkm's parents

```
!(match &self ($x isParentOf pkm)
    ("parent of pkm is " $x))
```

✅ Output → `[("parent of pkm is mike), ("parent of pkm is john")]`

Example 4: Find john's children

```
!(match &self (john isParentOf $x) $x)
```

✅ Output → `[oo, pkm]`

Example 5: Nested Match

```
!(match &self ($x isParentOf pkm)
```

```
    (match &self ($x $y $z $k) ($x $y $z $k)))
```

- First finds → `john` and `mike` (parents of `pkm`).
- Then retrieves their details.

✅ Output →

```
[(john 26 2006 Female),
 (mike 25 2005 Male)]
```

---

4. 🛠️ Defining Rules (Functions)

MeTTa allows **defining new knowledge** using functions (rules).

We use the syntax:

```
(= (FunctionName parameters) (FunctionBody))
```

Rule 1: Siblings

Two people are **siblings** if they share the same parent.

```
(= (isSibling $x $y)
   (match &self ($n isParentOf $x)
         (match &self ($n isParentOf $y) True)))
```

Breakdown:

1. Find all `$n` such that `$n isParentOf $x`.
2. For each such `$n`, check if `$n isParentOf $y`.
3. If yes → return `True`.

Examples:

```
!(isSibling pkm oo)     ; [True, True]  (both have john and mike as parents)
!(isSibling pkm mike)   ; []            (mike is not a sibling, he's a parent)
```

Rule 2: Parent

A person is a **parent** if the fact `(X isParentOf Y)` exists.

```
(= (isParent $x $y)
  (match &self ($x isParentOf $y) True))
```

Examples:

```
!(isParent mike pkm)   ; [True]
!(isParent john oo)    ; [True]
```

---

5. ➕ Arithmetic in MeTTa

Yes, MeTTa can do arithmetic directly:

```
!(+ 2 3)   ; [5]
!(- 2 3)   ; [-1]
!(- 5 3)   ; [2]
```

👉 Shows that MeTTa is a **functional paradigm language**, not just a database query tool.

---

6. 🎯 Key Insights

- **Atom Space** = database (facts as graphs).
- **Atoms** = smallest knowledge units.
- **match** = query + pattern matching (like graph SQL).
- **Rules** = define relationships (e.g., siblings, parents).

- **No redundancy needed** → rules help derive info instead of storing duplicates.
- **Supports functional programming** → can mix logic + math.

💡 **Analogy**:

- Think of Atom Space as **Google Drive for knowledge**.
- Each file is an **atom**.
- `match` is the **search engine**.
- Rules are **shortcuts/macros** that save space and compute new knowledge automatically.

▼ 📘 MeTTa Notes — CRED Operations & Python Integration

---

1. 🗂️ Spaces in MeTTa

- A **Space** is like a **dynamic environment** where you can **add, remove, and update** atoms at runtime.
- By default, MeTTa provides `&self` → the current working space.
- But you can also **create new spaces** and manage them separately.

---

2. 🆕 Creating a New Space

To create a new space, we use:

```
!(bind! &new-space (new-space))
```

- **`bind!`** → binds a new space instance to a reference (here `&new-space`).
- **`new-space`** → creates a new empty space.

Example:

```
!(match &new-space $x $y)
```

✅ Output → `[()] []` (empty, since no atoms yet).

---

3. ➕ Adding Atoms

To insert new facts into a space:

```
!(add-atom &new-space (parent mike pkm))
!(add-atom &new-space (parent kiku oo))
```

Now query:

```
!(match &new-space (parent $x pkm) $x)
```

✅ Output → `[mike]`

```
!(match &new-space $x $x)
```

✅ Output → `[(parent kiku oo), (parent mike pkm)]`

👉 Facts are successfully stored in the new space.

---

4. ➖ Removing Atoms

To delete facts:

```
!(remove-atom &new-space (parent mike pkm))
!(match &new-space $x $x)
```

✅ Output → `[(parent kiku oo)]`

👉 The `(parent mike pkm)` fact is gone.

---

5. 🔄 Updating Atoms

There is no built-in **update** in MeTTa, but we can **define our own** by combining `remove-atom` and `add-atom`.

Custom Rule:

```
(= (update-atom $space $old-atom $new-atom)
  (let $x (remove-atom $space $old-atom)
      (add-atom $space $new-atom)))
```

Example:

```
!(update-atom &new-space (parent kiku oo) (parent kiku pkm))
!(match &new-space $x $x)
```

✅ Output → `[(parent kiku pkm)]`

👉 Old fact `(parent kiku oo)` replaced by `(parent kiku pkm)`.

---

6. ⚡ Control Flow in MeTTa

`let`

- Used to bind a temporary variable and reuse it in expressions.

```
!(let $x (+ 2 3) (* $x 4))
```

✅ Output → `[20]`

Explanation:

- `$x = (+ 2 3) = 5`
- Then compute `(* 5 4)` → `20`.

`let*`

- Used when we need **sequential variable substitutions** (chained computations).

```
!(let*
  (
    ($x (+ 1 3))   ; x = 4
    ($y (+ $x 2))  ; y = 6
    ($z (* $y 3))  ; z = 18
  )
  ($z))
```

✅ Output → `[18]`

---

8. ✅ Summary

- **Spaces** let us create isolated environments for data.
- **add-atom / remove-atom** allow dynamic changes.
- **update-atom** can be implemented using `remove-atom` + `add-atom`.
- **let / let**provide control flow and variable binding.
- **Python integration** allows using external libraries (NumPy, math, SciPy).
- Together, MeTTa + Python = **graph knowledge + numerical computation**.

▼ 🐍 Python Integration with MeTTa

Why integrate Python?

- MeTTa is great for knowledge representation, but it lacks **libraries for math, ML, and science**.
- Python provides **NumPy, SciPy, math, etc.** → we can call them inside MeTTa.

Example 1: Using Python Function

```
!(py-atom math.ceil)
!((py-atom math.ceil) 3.14)
```

✅ Output → `[4]`

👉 Here, `math.ceil` (from Python) is bound as a MeTTa function.

Example 2: Full Python Script

```
from hyperon import MeTTa
import numpy as np

# Cosine similarity function
def cosine_similarity(v1, v2):
    v1, v2 = np.array(v1), np.array(v2)
    return float(np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2)))

# Start MeTTa
metta = MeTTa()

# Register Python function
metta.register_python_function(cosine_similarity, "cosine-sim")

# Run MeTTa queries
print(metta.run("!((cosine-sim [1 0] [0 1]))"))  # [[0.0]]
print(metta.run("!((cosine-sim [1 1] [1 1]))"))  # [[1.0]]
```

👉 Output:

```
[[0.0]]
[[1.0]]
```

▼ MeTTa Console Output & Debugging

1. 📤 Console Output in MeTTa

There are two main ways to output to the console:

1. `println!` → prints results directly to console.
2. `trace!` → prints intermediate steps (good for debugging).

2. 📝 Example: Defining Facts

```
(is gentalmen a)
(is lady b)
(is lady c)
```

3. 🖨️ Using `println!`

Example:

```
!(println! (match &self (is lady $x) (lady $x)))
```

✅ Console Output:

```
(lady c)
(lady b)
```

```

```

✅ Final Result:

```
[(), ()]
```

👉 Explanation:

- `println!` prints matches `(lady c)` and `(lady b)` to the console.
- The result of `println!` is `[()]` (since printing returns an empty result).

Example with Match

```
!(match &self (is gentalmen $x) ("gentalmen is " $x))
```

✅ Output:

```
[("gentalmen is " a)]
```

---

4. 🔎 Using `trace!`

- `trace!` is used for **debugging**.
- It prints extra information while keeping the main result.

Example:

```
(= (isgentalmen $x)
   (trace! (Inpute $x)
         (outpute: (match &self (is gentalmen $x) True))))
```

```
!(isgentalmen a)
```

✅ Console Output:

```
(Inpute a)
```

✅ Final Result:

```
[(outpute: True)]
```

👉 Explanation:

- `trace!` first prints `(Inpute a)`.
- Then evaluates the expression → returns `[(outpute: True)]`.

---

5. 🔁 Debugging Recursive Functions

We can also combine `println!` and recursion for step-by-step tracing.

Example: Factorial Function

```
(= (factorial $num)
    (if (== $num 0)         ; base case
        1
        (let*               ; recursive case
          (
            ($next (- $num 1))
            ($result (* $num $next))
            (() (println! ((next is $next) (current is $num) (result is $result))))
          )
          $result
        )
```

```
    )
)
```

Run:

```
!(factorial 5)
```

✅ Console Output:

```
((next is 4) (current is 5) (result is 20))
```

✅ Final Result:

```
[20]
```

👉 Explanation:

- When `$num = 5`, MeTTa prints intermediate values:
  - `next = 4`
  - `current = 5`
  - `result = 20`
- Then it returns the final result `[20]`.

---

6. ✅ Summary

- Use **`println!`** for direct console printing.
- Use **`trace!`** for debugging intermediate values while keeping output intact.
- Works well for:
  - Query debugging

- Pattern matching
- Recursive function tracing

Learing from docs

Recursion

Perfect 👍 I'll explain this recursion part of **MeTTa** in the same **style as your earlier notes** — step by step, simple, with comments and examples.

🌀 Basic Recursion in MeTTa

Recursion = A function calling itself until a **base case** is reached.

In **MeTTa**, recursion is the natural way to handle **repeated computations** and especially **recursive data structures** (like lists, trees, etc.).

Example: Counting the Length of a List

We want to build a function `length` that counts how many elements are in a list.

1. Base Case

```
(= (length ()) 0)
```

- `()` → empty list
- If the list is empty → the length is `0`

This is the **stopping condition** of recursion.

---

2. Recursive Case

```
(= (length (:: $x $xs))
  (+ 1 (length $xs)))
```

- `(:: $x $xs)` → list constructor
  - `$x` → the first element (head)
  - `$xs` → the rest of the list (tail)
- Rule: Length of a list = **1 (for the first element)** + **length of the rest of the list**

---

3. Running It

```
!(length (:: A (:: B (:: C ()))))
```

Step-by-step:

1. `(:: A (:: B (:: C ())))` → represents the list `[A, B, C]`
2. `(length (:: A ...))` → applies recursive rule:
   - `(+ 1 (length (:: B (:: C ()))))`
3. Again:
   - `(+ 1 (length (:: C ())))`
4. Finally:
   - `(+ 1 (length ()))` → base case = `0`
5. Computation:
   - `(+ 1 (+ 1 (+ 1 0)))`
   - Result = `3`

✅ Output:

```
[3]
```

⚡ How This Works

- **Two rules only**:
  - Base case → `(length ()) = 0`
  - Recursive case → `(:: $x $xs)` → add 1 + recurse on `$xs`
- These two cases are **mutually exclusive**:
  - If the input is `()` → only the base case matches
  - If the input is `(:: something ...)` → only recursive case matches

This acts like an **if-else condition**.

🔧 Important Notes

- In MeTTa, we used `(:: ...)` and `()` as **data constructors** (like `Cons` and `Nil` in functional programming).
- **We didn't predefine them as a list type** → they are just atoms.
- Example:

  ```
  !(length (hello world))
  ```

  → No matching rule, so it **won't reduce**.

👉 If you want a different style (say `Cons` and `Nil` instead of `::` and `()`), just redefine rules with those.

✅ Summary

- **Recursion in MeTTa** = function defined in terms of itself.
- **Base case** → stops recursion.
- **Recursive case** → breaks problem into smaller sub-problems.
- Example `length` shows how to process recursive data structures like lists.

Would you like me to also **extend your notes** by rewriting this example in the same **structured Markdown format** we've been building (with sections like "Concept", "Code", "Explanation", "Output") so it fits smoothly into your master notes?

▼ **map**

Got it 👍 Let's carefully break this down just like before — simple explanations, structured, with **code + explanation + output**.

🌟 Higher Order Functions & Conditionals in MeTTa

1. 🔼 What are Higher-Order Functions?

- A **higher-order function** = a function that **takes another function as input** or **returns a function**.
- In MeTTa, this naturally works because functions and atoms are all treated the same way.

Example: `apply-twice`

```
(= (apply-twice $f $x)
   ($f ($f $x)))

(= (square $x) (* $x $x))
(= (duplicate $x) ($x $x))
```

👉 Running:

```
!(apply-twice square 2)       ; 16
!(apply-twice duplicate 2)    ; ((2 2) (2 2))
!(apply-twice 1 2)            ; (1 (1 2))
```

🔎 Explanation:

- `(apply-twice $f $x)` → just applies `$f` two times:
  - First: `$f $x`
  - Then again: `$f ($f $x)`
- `square` works fine (`2 → 4 → 16`)
- `duplicate` makes pairs (`2 → (2 2) → ((2 2) (2 2))`)
- If `$f` is not a function (e.g. `1`) → it just constructs `(1 (1 2))` because MeTTa doesn't check types strictly.

Example: `map` (classic higher-order function)

```
(= (map $f ()) ())
(= (map $f (:: $x $xs))
  (:: ($f $x) (map $f $xs)))

(= (square $x) (* $x $x))
(= (twice $x) (* $x 2))
```

👉 Running:

```
!(map square (:: 1 (:: 2 (:: 3 ()))))
; (:: 1 (:: 4 (:: 9 ())))

!(map twice (:: 1 (:: 2 (:: 3 ()))))
; (:: 2 (:: 4 (:: 6 ())))

!(map A (:: 1 (:: 2 (:: 3 ()))))
; (:: (A 1) (:: (A 2) (:: (A 3) ())))
```

🔎 Explanation:

- `map` applies a function `$f` to **each element of a list**.
- Base case → empty list returns `()`
- Recursive case → apply `$f` to first element, then recurse on the rest.

---

2. 🔁 Conditional Statements

a) Using `if`

```
(= (factorial $x)
   (if (> $x 0)
       (* $x (factorial (- $x 1)))
       1))
```

👉 Running:

```
!(factorial 5) ; 120
```

🔎 Explanation:

- If `$x > 0` → multiply `$x` by factorial of `$x - 1` (recursive)
- Else → return `1` (base case).
- Important: `if` **is lazy** → it only evaluates the needed branch (prevents infinite loops).

Example:

```
(= (loop) (loop)) ; infinite loop
!(if True Success (loop)) ; Success
```

✅ Works because `(loop)` is never evaluated.

---

b) Using `case`

```
(= (factorial $x)
   (case $x
     ((0 1)
      ($_ (* $x (factorial (- $x 1)))))
   ))
```

👉 Running:

```
!(factorial 5) ; 120
```

🔎 Explanation:

- `case` = pattern matching across multiple possibilities.
- If `$x = 0` → return `1`
- Otherwise → fallback rule applies (`$x (fact $x-1)`).

---

c) `zip` Example with `case`

```
(= (zip $list1 $list2)
   (case ($list1 $list2)
       (((() ()) ())                        ; both empty
        (((:: $x $xs) (:: $y $ys)) (:: ($x $y) (zip $xs $ys))) ; pair elements
        ($else ERROR)                        ; fallback
   ))
)
```

👉 Running:

```
!(zip (:: A (:: B ())) (:: 1 (:: 2 ())))
; (:: (A 1) (:: (B 2) ()))

!(zip (:: A (:: B ())) (:: 1 ()))
; (:: (A 1) ERROR)
```

🔎 Explanation:

- `case` checks patterns in order:
  1. Both lists empty → return `()`
  2. Both non-empty → pair `(A 1)` then recurse
  3. Otherwise → ERROR (lists unequal length)

---

3. ❓ What is `::` in MeTTa?

- `::` is just an **atom used as a list constructor** (like `Cons` in Lisp or Haskell).
- It builds a **linked list structure**:

```
(:: A (:: B (:: C ())))
```

means → `[A, B, C]`

- `()` represents an **empty list** (like `Nil`).

So:

- Base case: `()` (end of list)
- Recursive case: `(:: head tail)`

This allows recursion to walk through a list element by element.

✅ **Summary:**

- Higher-order functions → pass functions around (`apply-twice`, `map`).
- Conditionals:
  - `if` → works like if-else, lazy evaluation.
  - `case` → pattern matching with multiple branches.
- `::` → constructor to build recursive lists (`(:: A (:: B ()))`).

Do you want me to **integrate this higher-order + conditionals section** into your existing structured notes (so it all flows as one PDF/Markdown later), or keep it as a separate tutorial-style chunk?

▼ condetional

🔷 Conditional Statements in MeTTa

In most programming languages, we have `if-else` to check conditions.

In **MeTTa**, the same idea applies, but with a slightly different flavor:

```
(if <condition> <when-true> <when-false>)
```

- `<condition>` → must evaluate to `True` or `False` (or something equivalent).
- `<when-true>` → returned if condition is `True`.
- `<when-false>` → returned if condition is `False`.

Example:

```
(if (== 2 2) "Yes" "No")
```

👉 Output: `"Yes"`

✅ Deterministic `if`

When a condition **always evaluates to a single truth value**, the `if` statement behaves deterministically (only one possible outcome).

```
(if (== 5 3) True False)
```

👉 Always returns `False` (deterministic).

🔀 Non-deterministic `if`

In MeTTa, conditions can have **multiple possible outcomes** if they rely on facts or nondeterministic rules.

This means the same `if` could yield **multiple results**, depending on which rule gets matched.

Example (from your frog case 🐸):

```
(= (frog $x)
  (and (croaks $x)
      (eats_flies $x)))

(= (croaks Fritz) True)
```

```
(= (eats_flies Fritz) True)
(= (croaks Sam) True)
(= (eats_flies Sam) False)

!(if (frog $x) ($x is Frog) ($x is-not Frog))
```

- For **Fritz** → `(frog Fritz)` is `True` → output: `(Fritz is Frog)`
- For **Sam** → `(frog Sam)` is `False` → output: `(Sam is-not Frog)`

This is **non-determinism** because `$x` was a **free variable**.

MeTTa tries different substitutions (`Fritz`, `Sam`, ...) and evaluates each possibility.

🔷 Logical Operations

MeTTa provides standard logic operators (like in functional languages):

- `(and A B)` → True if **both** are true
- `(or A B)` → True if **either** is true
- `(not A)` → True if **A is False**

Used inside `if` or recursive rules.

🔷 Deterministic vs Non-deterministic (Side by Side)

Type	**Definition**	**Example**	**Behavior**
Deterministic	Always gives one fixed result for same input	`(if (== 1 1) True False)`	Always returns `True`
Non-deterministic	Can give multiple possible results depending on facts / free vars	`(if (frog $x) ($x is Frog) ($x is-not Frog))`	Returns **different answers** for `Fritz` and `Sam`

🔷 Why non-determinism is powerful?

- It lets MeTTa **explore multiple branches of computation automatically**.
- That's why you can solve problems like **subset sum** by generating many candidates (`gen-bin-list`) and letting MeTTa filter the right ones with conditions.
- Traditional languages need explicit loops; MeTTa uses **logical search** + recursion.

✅ In simple words:

- **Deterministic if** = one path, always same result.
- **Non-deterministic if** = multiple possible worlds explored.

Would you like me to **extend your Markdown notes** and add a whole new **"Conditionals & Determinism" section** (like we did for recursion + console output), so later we can export everything into a polished PDF?

▼ match and unification

Perfect — you've reached the **hardcore part of MeTTa** now: evaluation, pattern-matching, and unification.

Let's carefully unpack this step by step in the same **"explain like I'm a beginner"** way we've been doing.

🔷 1. Function Evaluation & Matching

In MeTTa, when you call something like:

```
(only-a A)
```

the interpreter tries to **match** this expression against the **left-hand side of equalities** you have defined.

```
(= (only-a A) (Input A is accepted))
```

👉 The idea is:

- If the input **matches** `(only-a A)`,

  then replace it with `(Input A is accepted)`

So:

- `(only-a A)` → ✅ matches → output `(Input A is accepted)`
- `(only-a B)` → ❌ doesn't match → no result (unreduced expression)
- `(only-a $x)` → ✅ `$x` is a free variable, so it can match A, B, etc.

Behind the scenes

You can think of:

```
! (only-a A)
```

as secretly being rewritten to:

```
! (match &self (= (only-a A) $result) $result)
```

Here's what those keywords mean:

- **`match`** → search in a knowledge base (space) for something that fits.
- **`&self`** → means "search inside the current space (this file/program)".
- **`$result`** → variable that will capture whatever the RHS is.

So `(only-a A)` becomes a **query** asking:

"Is there a rule `(= (only-a A) something)` in the current knowledge space?"

🔷 2. Facts vs Rules

You can also store **facts** directly:

```
(Parent Tom Bob)
(Parent Pam Bob)
```

```
(Parent Tom Liz)
(Parent Bob Ann)
```

These are just **data entries** in the knowledge base.

Now you can query them with `match`.

Example

```
(= (get-parent-entries $x $y)
  (match &self (Parent $x $y) (Parent $x $y)))
```

- This says: "To get parent entries, match facts of the form `(Parent $x $y)` and return them."

So:

```
!(get-parent-entries Tom $_)
```

= “Find all children of Tom”.

→ result: `(Parent Tom Liz)` and `(Parent Tom Bob)`.

Another function:

```
(= (get-parents $x)
  (match &self (Parent $y $x) $y))
```

This asks:

“Find all `y` such that `(Parent y x)` exists.”

So:

```
!(get-parents Bob)
```

→ `[Tom, Pam]`.

🔷 3. From Facts to Rules

Facts only state relationships, but rules let you **derive new facts**.

```
(Implies (Parent $x $y) (Child $y $x))
```

This says:

"If `$x` is parent of `$y`, then `$y` is a child of `$x`."

Deduction Function

```
(= (deduce $B)
  (match &self (Implies $A $B)
       (match &self $A $B)))
```

Process:

1. Look for rules of the form `(Implies $A $B)`
2. If the premise `$A` holds, then return `$B`

Example:

```
!(deduce (Child $x Tom))
```

Steps:

- Search `(Implies $A (Child $x Tom))`
- Find `(Implies (Parent $x $y) (Child $y $x))`
- Match `$y` with Tom → `$x = Bob`
- Result: `(Child Bob Tom)`

✅ Now we derived new knowledge.

🔷 4. Unification (`unify`)

Matching is one-directional:

- "Does the input expression match a stored pattern?"

Unification is two-directional:

- "Can two patterns be made equal by assigning variables?"

Example

```
! (unify (parent $x Bob)   ; first pattern
         (parent Tom $y)   ; second pattern
         ($x $y)           ; output if success
         Fail)             ; else
```

- Compare `(parent $x Bob)` and `(parent Tom $y)`
- `$x` must be `Tom`, `$y` must be `Bob`
- ✅ Works → result `(Tom Bob)`

Failure Case

```
! (unify (A $x) ($x B) Yes No)
```

- Here, `$x` must be both `A` and `B` → contradiction
- ❌ Unification fails → result `No`

Using `&self` with `unify`

```
(Parent Tom Bob)
(Parent Bob Ann)

! (unify &self (Parent $x Bob) $x Fail)
```

This means:

- Search in the current space for `(Parent $x Bob)`

- Find `(Parent Tom Bob)`
- ✅ Result: `[Tom]`

🔷 5. Chained Unification Example

Query:

```
!(conclude (Parent Bob $y))
```

Unfolds step by step:

1. Try to evaluate `(conclude (Parent Bob $y))`
2. Match with rule:

   ```
   (= (conclude $A)
     (match &self (Implies $A $B)
          (match &self $A $B)))
   ```

   → `$A = (Parent Bob $y)`
3. Becomes:

   ```
   (match &self (Implies (Parent Bob $y) $B)
             (match &self (Parent Bob $y) $B))
   ```
4. This unifies with knowledge:

   ```
   (Implies (Parent $x $y) (Child $y $x))
   ```

   → `$x = Bob`, `$B = (Child $y Bob)`
5. Becomes:

   ```
   (match &self (Parent Bob $y) (Child $y Bob))
   ```
6. Fact `(Parent Bob Ann)` matches → `$y = Ann`

→ result `(Child Ann Bob)`

✅ Final Answer: `(Child Ann Bob)`

🔷 6. Summary of Keywords

- **`match &self (pattern) (result)`** → search in current space for matching pattern.
- **`&self`** → the current knowledge base.
- **`Implies (A) (B)`** → if A is true, then B is true (a rule).
- **`unify expr1 expr2 success fail`** → try to unify two patterns.
- **facts** like `(Parent Tom Bob)` → raw data entries.
- **rules** like `(Implies (Parent $x $y) (Child $y $x))` → derive new knowledge.

👉 In short:

- `match` = query facts/rules (like a database lookup).
- `unify` = try to glue two patterns by finding variable bindings.
- Rules (`Implies`) let you move from facts to conclusions.

Do you want me to also **draw a flow diagram** (facts → match → unify → result) so it becomes super clear when we make your final notes PDF?

▼ Data type and type casting

Perfect, you've now reached the **type system in MeTTa** — this is one of the most important topics because it ties together everything you learned earlier (atoms, functions, unification, evaluation). Let me break this down for you in a **structured, beginner-friendly way**.

🔷 1. Types of Symbols (Atoms)

Every atom in MeTTa can have a **type**.

Types themselves are also atoms.

The syntax is:

```
(: <atom> <type>)
```

✅ Example:

```
(: a A)   ; a is of type A
(: b B)   ; b is of type B
(: A Type) ; A is a Type
```

Notes:

- If you don't assign a type → atom has type `%Undefined%`.
- `%Undefined%` can match with anything (like a wildcard).
- You can check type with `get-type`.

👉 Example:

```
! (get-type a) ; A
! (get-type b) ; B
! (get-type c) ; %Undefined%
! (get-type A) ; Type
! (get-type B) ; %Undefined%
```

🔷 2. Types of Expressions

If you combine atoms into a tuple:

```
(: a A)
(: b B)

! (get-type (a b)) ; (A B)
```

- Type of `(a b)` = tuple of element types → `(A B)`

So **tuple type = tuple of types of its elements**.

🔷 3. Function Types

Now the important part: **function typing**.

MeTTa uses `->` to define function signatures.

```
(: foo (-> A B))
```

This means:

- `foo` takes an input of type `A`
- and produces an output of type `B`.

Example

```
(: a A)
(: foo (-> A B))

! (get-type (foo a)) ; B
```

Explanation:

- Even though `foo` has no body defined,
- `(foo a)` is recognized as type `B` (purely from type signatures).

🔷 4. Function Application vs Tuples

MeTTa treats **the first element in a tuple specially**:

- If the first element has a function type → treat the tuple as **function application**.
- Otherwise → treat it as a plain tuple.

Example

```
(: foo (-> A B))
(: a A)
(: b B)

! (foo a)        ; ✅ works (type checks)
! (foo b)        ; ❌ type error
! (get-type (b foo)) ; (B (-> A B)) -> just a tuple, not application
```

Key point:

- `(foo a)` = function application (special case).
- `(b foo)` = just a tuple, no error, no function call.

🔷 5. Type Errors

If you use the wrong type:

```
(: foo (-> A B))
(: a A)
(: b B)

! (foo b)  ; type error
```

Why?

- `foo` expects type `A`,
- but got type `B`.

So evaluation returns an **error expression**.

And `get-type` on it returns nothing.

🔷 6. Gradual Typing (%Undefined%)

When some atoms are untyped (`%Undefined%`), MeTTa tries to work with what it has.

Example:

```
(: foo (-> A B))
(: a A)

! (get-type (foo c)) ; depends (c is %Undefined%)
! (get-type (g a))   ; depends (g is %Undefined%)
```

- If **function is typed** but argument is `%Undefined%` → output type can still be derived sometimes.
- If **function is `%Undefined%`** but argument is typed → result type is `%Undefined%`.

This is called **gradual typing**: the system tolerates missing type info and infers what it can.

🔷 7. Summary

- `(: x T)` → assign type `T` to atom `x`.
- `%Undefined%` = no type info (matches with anything).
- `get-type` → retrieves the type of an atom/expression.
- Tuple type = tuple of element types.
- `>` → defines function signature (input → output type).
- If first element has arrow type → tuple is treated as function call.
- Wrong types → type error.
- Gradual typing lets `%Undefined%` coexist with typed atoms.

👉 Bro, now we've covered **atoms, facts, rules, unification, matching, AND typing**.

This is basically the **full beginner-to-intermediate foundation of MeTTa**.

Do you want me to **merge this section into your structured Markdown notes** (so it will be included in the final PDF with all other topics)?