Source: https://metta-lang.dev/docs/learn/learn.html

Learn MeTTa■

Tutorials available■

Introduction to evaluation in MeTTa

MeTTa from Ground Up: Patterns of Knowledge

Basics of Types and Metatypes

Stdlib overview

Basics of Functional Pragramming in MeTTa

Using MeTTa from Python

Metta-Motto

Minimal MeTTa

Source: https://metta-lang.dev/docs/playground/playground.html

Playground■

```
; put you code here
```

Source: https://metta-lang.dev/docs/learn/tutorials/eval_intro/intro.html

Introduction to evaluation in MeTTa■

Table of Contents■

Source: https://metta-lang.dev/docs/learn/tutorials/eval_intro/main_concepts.html

Main concepts■

Atoms and knowledge graphs■

MeTTa (Meta Type Talk) is a multi-paradigm language for declarative and functional computations over knowledge (meta)graphs.

Every MeTTa program lives inside of a particularAtomspace(or justSpaceif we don't insist on a particular internal representation). Atomspace is a part of theOpenCog (Hyperon)software ecosystem and it is essentially a knowledge database with the associated query engine to fetch and manipulate that knowledge. MeTTa programs can contain both factual knowledge and rules or functional code to perform reasoning on knowledge including programs themselves making the language fully self-reflective. One can draw an analogy with Prolog, which programs can also be considered as a knowledge base content, but with less introspective and more restrictive representation.

In an Atomspace, anAtomis a fundamental building block of all the data. In the context of graph representation, an Atom can be either a node or a link. In an Atomspace as metagraph, links can connect not only nodes, but other links, that is, they connect atoms, and they can connect any number of atoms (in contrast to ordinary graphs). In MeTTa as a programming language, atoms play the role of terms.

In the context of AI, Atoms can represent anything from objects, to concepts, to processes, functions or relationships. This enables the creation of rich, complex models of knowledge and reasoning.

Atom kinds and types■

There are 4 kinds of Atoms in MeTTa:

Symbol, which represents some idea or concept. Two symbols having the same name are considered equal and representing the same concept. Names of symbols can be arbitrary strings. Nearly anything can be a symbol, e.g.,A,f,known?,replace-me,■, etc.

Expression, which can encapsulate other atoms including other expressions. Basic MeTTa syntax is Scheme-like, e.g.(f A),(implies (human Socrates) (mortal Socrates)), etc.

Variable, which is used to create patterns (expressions with variables). Such patterns can be matched against other atoms to assign some specific binding to their variables. Variables are syntactically distinguished by a leading$, e.g.$x,$_,$my-argument, which tells the parser to convert a symbol to a variable. Patterns could be(Parent $x $y),(Implies (Human $x) (Mortal $x)),(:- (And (Implies $x $y) (Fact $x)) $y), or any other symbolic expression with variables. Such patterns get meaning when they are matched against expressions in the Atomspace.

Grounded, which represents sub-symbolic data in the Atomspace. It may contain any binary object, for example operation (including deep neural networks), collection or value. Grounded value type creators can define custom type, execution and matching logic for the value. There are some grounded atoms in the standard library to deal with numbers or strings, e.g.(+ 1 2)is an expression composed of a grounded atom+, which refers to an arithmetic operation, and1and2, which are grounded atoms containing specific values. Adding custom grounded atoms is a standard way for extending MeTTa and its interoperability.

Symbol,Variable,Groundedcan be considered as nodes, whileExpressioncan be considered as a generalized link. This interpretation of atoms plays an important role in MeTTa applications and Hyperon as a cognitive architecture, but is not essential for understanding MeTTa as a programming language.

MeTTa has optional typing, which is close enough to gradual dependent types, although with some peculiarities.%Undefined%is used for untyped expressions, while other types are represented as custom symbols and expressions.Symbol,Variable,Grounded, andExpressionare metatypes, which can be used to analyze MeTTa programs by themselves. They are subtypes ofAtom.

Special symbols■

There is a small number of built-in symbols which determine how a MeTTa program will be evaluated:

Equality symbol=defines evaluation rules for expressions and can be read as “can be evaluated as” or “can be reduced to”.

Colon symbol:is used for type declarations.

Arrow symbol->defines type restrictions for evaluable expressions.

These atoms are ofSymbolmetatype, and do not refer to particular binary objects unlikeGroundedatoms, but they are processed by the interpreter in a special way.

Source: https://metta-lang.dev/docs/learn/tutorials/eval_intro/basic_evaluation.html

Basic evaluation■

MeTTa programs■

Programs in MeTTa consist of a number of atoms (mostly expressions, but individual symbols or grounded atoms can also be put there). A MeTTa script is a textual representation of the program, which is parsed atom-by-atom, and put into a program Space.

In particular, binary objects wrapped into grounded atoms are constructed from their textual representation in the course of parsing. For example,+and1.05will be turned into grounded atoms containing corresponding operation and value. Particular grounded atoms and their textual representation is not a part of the core MeTTa language, but is defined in modules (both built-in and custom). How modules and grounded atoms are introduced is discussed in another tutorial.

If a programmer wants some atom to be evaluated immediately instead of adding it to the Space,!should be put before it. The result of evaluation will not be added to the Space, but will be included into the output result of the whole program.

MeTTa scripts can also have comments, starting with;, which will be ignored by the parser.

In the following program, the first two atoms will be added to the program space, while the next two expressions will be immediately evaluated and appear in the output.

```
; This line will be ignored.
Hello ; This symbol will be added to the Space
(Hello World) ; This expression will also be added
! (+ 1 2) ; This expression will be immediatedly evaluated
! (Hi there) ; as well as this one
```

If an expression starts with a grounded atom containing an operation, this operation is executed on the other elements of the tuple acting as its arguments.(+ 1 2)is naturally evaluated to3.

At the same time,(Hi there)is evaluated to itself, becauseHiis not a grounded operation, but just a custom symbol. It acts similar to a data constructor in Haskell (more on this in another tutorial). Let us consider how to do computations over symbolic expressions in MeTTa.

Equalities■

For a symbolic expression in MeTTa to be evaluated into something different from itself, an equality should be defined. Equality expressions work similar to function definitions in other languages. There is a number of important differences, though.

Let us consider a few examples.

A nullary function simply returns its body

```
(= (h) (Hello world))
! (h)
```

Some functions can accept only specific values of its argument. When this argument is passed, the right-hand side of the corresponding equality is returned

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

Note that(only-a B)is not reduced. In MeTTa, functions should not be total, and there is no hard boundary between a function and a data constructor. For example, consider this program:

```
! (respond me)
(= (respond me) (OK, I will respond))
! (respond me)
```

The first(respond me)will remain unchanged, while the second one will be transformed.

Functions can have variables as parameters, just like in other languages.

```
(= (duplicate $x) ($x $x))
! (duplicate A)
! (duplicate 1.05)
```

The passed arguments replace corresponding variables in the right-hand part of the equality.

Its arguments can be expressions with some structure

```
(= (swap (Pair $x $y)) (Pair $y $x))
! (swap (Pair A B)) ; evaluates to (Pair B A)
```

One may notice that this feature is similar to pattern matching in functional languages:

```
(= (Cdr (Cons $x $xs)) $xs)
! (Cdr (Cons A (Cons B Nil))) ; outputs (Cons B Nil)
```

But it is more general, because the structure of patterns can be arbitrary. In particular, patterns can contain the same variable encountered multiple times.

```
(= (check ($x $y $x)) ($x $y))
! (check (B A B)) ; reduced to (B A)
! (check (B A A)) ; not reduced
```

Functions can have multiple (nondeterministic) results. The following code will output both0and1

```
(= (bin) 0)
(= (bin) 1)
! (bin) ; both 0 and 1
```

Note that equations for functions are not mutually exclusive, and the following code will output two results (not onlycatched) in the last case

```
(= (f special-value) catched)
(= (f $x) $x)
! (f A) ; A
! (f special-value) ; both catched and special-value
```

Most importantly, variables can also be passed when calling a function, unlike imperative or functional languages. This will result in returning corresponding right-hand sides of equalities.

```
(= (brother Mike) Tom)
(= (brother Sam) Bob)
! (brother $x) ; just Tom and Bob are returned
! ((brother $x) is the brother of $x) ; the binding for $x is not lost
```

All these features are implemented using one mechanism, which is discussed later.

Evaluation chaining■

The result of the function is evaluated further both for symbolic and grounded operation:

```
(= (square $x) (* $x $x))
! (square 3)
```

Here,(square 3)is first reduced to(* 3 3), which, in turn, is evaluated to9by calling the grounded operation*.

In the following example,Seconddeconstructs the input list and returnsCarfor its tail, which is evaluated further

```
(= (Car (Cons $x $xs)) $x)
(= (Second (Cons $x $xs)) (Car $xs))
! (Second (Cons A (Cons B Nil))) ; outputs B
```

Arguments of functions will typically be evaluated before the function is called. How this behavior can be controlled is discussed in a separate tutorial. The following examples should be pretty straightforward:

```
! (* (+ 1 2) (- 8 3)) ; 15
(= (square $x) (* $x $x))
! (square (+ 2 3)) ; 25
(= (triple $x) ($x $x $x))
(= (grid3x3 $x) (triple (triple $x)))
! (grid3x3 (square (+ 1 2))) ; ((9 9 9) (9 9 9) (9 9 9))
```

This behavior is not different from other, especially functional, languages.

Passing results of nondeterministic functions to other functions (both deterministic and nondeterministic) cause the outer functions to be evaluated on each result. Consider the following examples:

```
; nondeterministic function
(= (bin) 0)
(= (bin) 1)
; deterministic triple
(= (triple $x) ($x $x $x))
! (triple (bin)) ; (0 0 0) and (1 1 1)
; nondeterministic pair
(= (bin2) ((bin) (bin)))
! (bin2) ; (0 0), (0 1), (1 0), (1 1)
; deterministic summation
(= (sum ($x $y)) (+ $x $y))
(= (sum ($x $y $z)) (+ $x (+ $y $z)))
! (sum (triple (bin))) ; 0, 3
! (sum (bin2)) ; 0, 1, 1, 2
; nondeterministic increment
(= (inc-flip $x) (+ 0 $x))
(= (inc-flip $x) (+ 1 $x))
! (inc-flip 1) ; 1, 2
! (inc-flip (bin)) ; 0, 1, 1, 2
```

(triple (bin))produces only two results, bevause(bin)is evaluated first and then passed totriple, while(bin2)produces four results, because each(bin)in its body is evaluated independently. Deterministicsumsimply processes each nondeterministic value of its argument, whileinc-flipdoubles the number of input values.

Source: https://metta-lang.dev/docs/learn/tutorials/eval_intro/recursion.html

Recursion and control■

Basic recursion■

A natural way to represent repetitive computations in MeTTa is recursion like in traditional functional languages, especially for processing recursive data structures. Let us consider a very basic recursive function, which calculates the number of elements in the list.

```
(= (length ()) 0)
(= (length (:: $x $xs))
   (+ 1 (length $xs)))
! (length (:: A (:: B (:: C ()))))
```

The function has two cases, which are mutually exclusive de facto, and act as a conditional control structure. The base case returns0for an empty list(). Recursion itself takes place inside the second equality, in whichlengthis defined via itself on the deconstructed parameter.

Notice that we didn't define the recursive data structure (list) here, and used arbitrary atoms (::and()) as data constructors.lengthcan be called on anything, e.g.(length (hello world)), but this expression will simply be not reduced, because there are no suitable equalities for it. You can write your own version of length for otherConsandNilinstead of::and():

```
(= (length ...) 0)
(= (length ...)
   (+ 1 (length $xs)))
! (length (Cons A (Cons B (Cons C Nil))))
```

If a function expects specific subset of all possible expressions as input, types for corresponding atoms should be defined. However, we focus here on the basic evaluation process itself and leave types foranother tutorial.

Higher order functions■

Higher-order functions is a powerful abstraction, which naturally appears in MeTTa. Consider the following code:

```
(= (apply-twice $f $x)
   ($f ($f $x)))
(= (square $x) (* $x $x))
(= (duplicate $x) ($x $x))
! (apply-twice square 2) ; 16
! (apply-twice duplicate 2) ; ((2 2) (2 2))
! (apply-twice 1 2) ; (1 (1 2))
```

apply-twicetakes a function as its first parameter and applies it twice to its second parameter. In fact, it doesn't really care if it is a function or not. It simply constructs a corresponding expression for further evaluation.

Passing functions into recursive functions is very convenient for processing various collections. Consider the following basic example

```
(= (map $f ()) ())
(= (map $f (:: $x $xs))
   (:: ($f $x) (map $f $xs)))
(= (square $x) (* $x $x))
(= (twice $x) (* $x 2))
! (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) ())))
```

maptransforms a list by applying a given function (or constructor) to each element. There is a rich toolset of higher-order functions in functional programming. They are covered inanother tutorial.

Conditional statements■

Let us imagine that we want to implement the factorial operation. If we want to use grounded arithmetics, we will not be able to use pattern matching to deconstruct a grounded number and distinguish the base and recursive cases. We can write(= (fact 0) 1), but we cannot just write(= (fact $x) (* $x (fact (- $x 1)))). However, we can useif, which works much like if-then-else construction in any other language. Consider the following code

```
(= (factorial $x)
   (if (> $x 0)
       (* $x (factorial (- $x 1)))
       1))
! (factorial 5) ; 120
```

(factorial $x)will be reduced to(* $x (factorial (- $x 1)))if(> $x 0)isTrue, and to1otherwise.

It should be noted thatifdoesn't evaluate all its arguments, but "then" and "else" branches are evaluated only when needed.factorialwouldn't work otherwise, although this should be more obvious from the following code, which will not execute the infinite loop

```
(= (loop) (loop)) ; this is an infinite loop
! (if True Success (loop)) ; Success
```

Application ofiflooks like as an ordinary function application, andifis indeed implemented in pure MeTTa as a function. How it is done is discussed inanother tutorial.

Another conditional statement in MeTTa iscase, which pattern-matches the given atom against a number of patterns sequentially in a mutually exclusive way. A different version of the factorial operation can be implemented with it:

```
(= (factorial $x)
   (case $x
     ((0 1)
      ($_ (* $x (factorial (- $x 1)))))
   )
)
! (factorial 5) ; 120
```

In contrast toif,casedoesn't check logical conditions but performs pattern matching similar to application of a function with several equality definitions. Thus, their usage is somewhat different. For example, if one wants to zip two lists, it is convenient to distinguish two cases - when both lists are empty, and both lists are not empty. But when two lists are of different lengths, there will a situation when neither of these cases will be applicable, and the expression will not be reduced. Try to run this code:

```
(= (zip () ()) ())
(= (zip (:: $x $xs) (:: $y $ys))
   (:: ($x $y) (zip $xs $ys)))
! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ()))
! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) (zip (:: B ()) ()))
```

The non-matchable part remains unreduced. Of course, adding two equalities for(zip (:: $x $xs) ())and(zip () (:: $y $ys))could be used (you can try to add them in the above code), and it would be a more preferable way in some cases. However, usingcasehere could be more convenient:

```
(= (zip $list1 $list2)
   (case ($list1 $list2)
         ((() ()) ())
          (((:: $x $xs) (:: $y $ys)) (:: ($x $y) (zip $xs $ys)))
          ($else ERROR)
         )
   )
)
! (zip (:: A (:: B ())) (:: 1 (:: 2 ()))) ; (:: (A 1) (:: (B 2) ()))
! (zip (:: A (:: B ())) (:: 1 ())) ; (:: (A 1) ERROR)
```

Source: https://metta-lang.dev/docs/learn/tutorials/eval_intro/recursion_vars.html

Free variables and nondeterminism again, recursively■

A piece of logic■

We have already encounteredif, which reduces to different expressions depending on whether its first argument isTrueorFalse. They are returned by such grounded operations as>or==. There are also such common logical operations asand,or,notin MeTTa (see the stdlib tutorial for more information). Things start to get interesting, when we pass free variables into logical expressions.

Let us consider the following program.

```
; Some facts as very basic equalities
(= (croaks Fritz) True)
(= (eats_flies Fritz) True)
(= (croaks Sam) True)
(= (eats_flies Sam) False)
; If something croaks and eats flies, it is a frog.
; Note that if either (croaks $x) or (eats_flies $x)
; is false, (frog $x) is also false.
(= (frog $x)
   (and (croaks $x)
        (eats_flies $x)))
! (if (frog $x) ($x is Frog) ($x is-not Frog))
; (green $x) is true if (frog $x) is true,
; otherwise it is not calculated.
(= (green $x)
   (if (frog $x) True (empty)))
! (if (green $x) ($x is Green) ($x is-not Green))
```

There are some facts aboutFritzandSam, and there is a general rule about frogs. Just asking whether(frog $x)isTrue, we can infer thatFritzis aFrog, whileSamis not aFrog(detailed analysis of how it works is given in another tutorial).

(green $x)is defined in such a way that it isTruewhen(frog $x)isTrue. However, if(frog $x)isFalse, it returns(empty)(which is evaluated to an empty set of results, which is equivalent to not defining a function on the corresponding data). Running the above code reveals thatFritzis green, but we cannot say whetherSamis green or not.

Make the replacement in the above code with the naive version of(green $x).

```
(= (green $x)
   (if (frog $x) True (empty)))
(= (green $x) (frog $x))
```

This will result in(Sam is-not Green)to appear, which shows that(= (green $x) (frog $x))is not the same as logical implication even with boolean return values, although it is not precisely the same as equivalence (more on this in another tutorial).

You can also try to add(= (eats_flies Tod) True)into the set of facts.(green Tod)can be evaluated only partially (particular behavior is not fixed and might be different for different versions of MeTTa).

Recursion with nondeterminism■

Let us generalize generation of random binary pairs to binary lists of a given length. Examine the following program:

```
; random bit
(= (bin) 0)
(= (bin) 1)
; binary list
(= (gen-bin $n)
   (if (> $n 0)
       (:: (bin) (gen-bin (- $n 1)))
```

```
       ()))
! (gen-bin 3)
```

It will generate all the binary strings of length3. Similarly, functions to generate all the binary trees of the given depth, or all the strings up to a certain length can be written.

Try to write a function, which will output the binary list of the same length as an input list. You don't need to calculate the length of this list and to useif.

```
(= (bin) 0)
(= (bin) 1)
(= (gen-bin-list ()) ())
(= (gen-bin-list ...)
   ...)

! (gen-bin-list (:: 1 (:: 5 (:: 7 ()))))
```

Solving problems with recursive nondeterminism■

Let us put all the pieces together and solve the subset sum problem. In this problem, a list of integers is given, and one needs to find its elements whose sum will be equal to a given target sum. Candidate solutions in this problem can be represented as binary lists. Then, the sum of taken elements can be calculated as a sum of products of elements of two lists.

```
; random bit
(= (bin) 0)
(= (bin) 1)
; binary list with the same number of elements
(= (gen-bin-list ()) ())
(= (gen-bin-list (:: $x $xs))
   (:: (bin) (gen-bin-list $xs))
)
; sum of products of elements of two lists
(= (scalar-product () ()) 0)
(= (scalar-product (:: $x $xs) (:: $y $ys))
   (+ (* $x $y) (scalar-product $xs $ys))
)
; check the candidate solution
(= (test-solution $numbers $solution $target-sum)
   (if (== (scalar-product $numbers $solution)
           $target-sum)
       $solution
       (empty)
   )
)
; task
(= (task) (:: 8 (:: 3 (:: 10 (:: 17 ()))))))
! (test-solution (task) (gen-bin-list (task)) 20)
```

This solution is not scalable, but it illustrates the general idea of how nondeterminism and recursion can be combined for problem solving. Note that passing a variable instead of(gen-bin-list (task))will not work here. What is the difference with thefrogexample? The answer will be given in the next tutorial.

Source: https://metta-lang.dev/docs/learn/tutorials/ground_up/intro.html

MeTTa from Ground Up: Patterns of Knowledge■

Table of Contents■

Source: https://metta-lang.dev/docs/learn/tutorials/ground_up/query_knowledge.html

Querying space content■

Introduction■

As a declarative language, MeTTa was designed for expressing complex relationships between entities of various types, performing computations on these relationships, and manipulating their structures. It allows programmers to specify AI algorithms and knowledge representations in a rich and flexible way. MeTTa code can be generated and processed in run-time by MeTTa programs themselves, which adds a lot of dynamism in working with complex data structures for AI tasks.

One of the main purposes of developing MeTTa was to operate over a knowledge metagraph called AtomSpace (or just Space), designed to store all sorts of knowledge, from raw sensory/motor data to linguistic and cultural knowledge, to abstract, mathematical, scientific or programming knowledge.

AtomSpace represents knowledge in the form of Atoms, the fundamental building block of all the data. Specifically, in the context of AI, an Atom can represent anything from objects, to concepts, to processes or relationships, to reasoning rules and algorithms.

While MeTTa may look like an ordinary language in certain aspects, it is built on top of operations over the knowledge metagraph, which is essential to understand how it works.

Knowledge declaration and matching query■

Let us look at a basic example of specifying relations between concepts, e.g., family relationships. While there are different ways to do this, in MeTTa, one can simply put expressions like the following into the program

```
(Parent Tom Bob)
```

This expression being put into the program space can be treated as the fact that Tom is Bob's parent. We startParentwith capitalPto distinguish it from a function, which we would prefer to start withpin this case, although this naming convention is not mandatory.

One can add more such expressions to the program space. But what can we do with such expressions? Thetutorialoverviewed the evaluation process of expressions, for which equalities are specified. But is there any use of expressions without equalities?

The core operation in MeTTa ismatching. It searches for all declared atoms corresponding to the given pattern and produces the output pattern. The process is similar to the manner in which one can search text strings with regular expressions, but it is for searching for subgraphs in a metagraph.

We can compose a query for matching using the grounded functionmatch. It expects three arguments:

a grounded atom referencing a Space;

pattern atom to be matched;

output pattern typically containing variables from the input pattern.

Basic examples■

Let us consider the following program

```
(Parent Bob Ann)
; This match will be successful
! (match &self (Parent Bob Ann) (Bob is Ann`s father))
; The following line will return []
! (match &self (Parent Bob Joe) (Bob is Joe`s father))
```

&selfis; a reference to the current program Space. We can refer to other Atomspaces, but we will cover it later. The second argument in the firstmatchexpression(Parent Bob Ann)is an expression to be matched against atoms in the current Space, and the third argument(Bob is Ann's father)is the atom to be returned if matching succeeded.

The program above will return[(Bob is Ann's father)]and[], since when the desired expression pattern wasn't foundmatchreturns nothing.

We can construct more interesting queries using variables. Let us consider the program

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

! (match &self (Parent $x Bob) $x) ; [Tom, Pam]
```

The pattern(Parent $x Bob), i.e. "Who are Bob's parents?", can be matched against two atoms (facts) in the Space, and corresponding bindings for$xwill be used to produce the result ofmatch. Here, we will get two matches[Tom, Pam], which can be viewed as anondeterministicevaluation ofmatch.

Please, note thatmatchdoesn't search in subexpressions. The following code will return[Ann]only:

```
(Parent Bob Ann)
(Parent Pam (Parent Bob Pat))

! (match &self (Parent Bob $x) $x) ; Ann
```

We can make even broader queries: "Who is a parent of whom?", or "Find$xand$ysuch that$xis a parent of$y".

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

! (match &self (Parent $x $y) ($x $y))
```

The output should contain the following pairs (the order can be different due to MeTTa's nondeterminism)[(Pat Bob), (Bob Ann), (Bob Pat), (Tom Bob), (Tom Liz), (Pat Pat)]. Can you add the query in the above program to retrieve only parents and children with same names?

Source: https://metta-lang.dev/docs/learn/tutorials/ground_up/unify_func.html

Functions and unification■

Function evaluation and matching■

As discussed in thetutorial, evaluable expressions can contain variables, and they are pattern-matched against left-hand side of equalities. In fact, evaluation of expressions can be understood as recursively constructing queries for equalities. Consider this code as an example

```
(= (only-a A) (Input A is accepted))
! (only-a A)
! (only-a B)
! (only-a $x)
```

Evaluation of(only-a A)can be thought of as execution of query(match &self; (= (only-a A) $result) $result).$resultwill be bound with the right-hand side of the function case (body), if the left-hand side matches with the expression under evaluation. Does it work for(only-a B)and(only-a $x)?

Let us check that the following program produces the same result:

```
(= (only-a A) (Input A is accepted))
! (match &self (= (only-a A) $result) $result)
! (match &self (= (only-a B) $result) $result)
! (match &self (= (only-a $x) $result) $result)
```

There is one difference.matchproduces the empty result in the second case, while the interpreter keeps this expression unreduced. The interpreter is performing some additional processing on top of such equality queries.

While allowing the MeTTa interpreter to construct equality queries automatically for evaluating expressions like(only-a A)is very convenient for functional programming, usingmatchdirectly allows for more compact knowledge representation and efficient queries glued together in a custom way.

It should also be noted that obtaining multiple results in queries to knowledge bases is very typical, and since the semantics of evaluating expressions in MeTTa is natively related to such queries, all evaluations in MeTTa are secretly or explicitly nondeterministic.

Let us analyze the following program:

```
(Parent Tom Bob)
(Parent Pam Bob)
(Parent Tom Liz)
(Parent Bob Ann)
(= (get-parent-entries $x $y)
   (match &self (Parent $x $y) (Parent $x $y)))
(= (get-parents $x)
   (match &self (Parent $y $x) $y))
! (get-parent-entries Tom $_)
! (get-parents Bob)
```

We can callmatchfrom an ordinary function, and we can still pass variable arguments to it, so(get-parent-entries Tom $_)is equivalent to(match &self; (Parent Tom $y) (Parent Tom $y)).

The result[(Parent Tom Liz), (Parent Tom Bob)]is not reduced further. It is convenient, when we want to represent pieces of knowledge and process them.

(get-parents Bob)returns[Tom, Pam]. Executingmatchfrom functions allows creating convenient functional abstractions while still working with declarative knowledge.

For example, how would you write a function, which returns grandparents of a given person?

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

```
(Parent Bob Pat)
(Parent Pat Jim)

(= (get-parents $x)
   (match &self (Parent $y $x) $y))
(= (get-grand-parents $x)
   (...))
! (get-grand-parents Pat)
```

From facts to rules■

One may notice that equality queries for functions suppose that there are free variables not only in the query, but also in the Atomspace entries. These entries can be not only function definitions, but other arbitrary expressions, which can be used to represent general knowledge or rules. For example, one can write

```
(Parent Tom Bob)
(Parent Bob Ann)
(Implies (Parent $x $y) (Child $y $x))
(= (deduce $B)
   (match &self (Implies $A $B)
          (match &self $A $B))
)
(= (conclude $A)
   (match &self (Implies $A $B)
          (match &self $A $B))
)
! (deduce (Child $x Tom))    ; [(Child Bob Tom)]
! (conclude (Parent Bob $y)) ; [(Child Ann Bob)]
```

IfChildandParentwere predicates returningTrueorFalse(as in thefrog example), we could somehow use=instead ofImplies. But here we don't evaluate the premise toTrueorFalse, but check that it is in the knowledge base. It makes inference better controllable. We can easily go from premises to conclusions withconclude, or to verify conclusions by searching for suitable premises withdeduce.

We will discuss different ways of introducing reasoning in MeTTa in more detail later. What we want to focus on now is that in both cases a query with variables is constructed, say,(Implies (Parent Bob $y) $B)and it should be matched against some entry in the knowledge base with variables as well, namely,(Implies (Parent $x $y) (Child $y $x))in our example. This operation is called unification, and it is available in MeTTa in addition tomatch.

Unification■

Functionunifyaccepts two patterns to be unified (matched together in such the way that shared variables in them get most general non-contradictory substitutions). The function is evaluated to its third argument if unification is successful and to the fourth argument otherwise. The following program shows the basic example.

```
! (unify (parent $x Bob) ; the first pattern
         (parent Tom $y) ; the second pattern
         ($x $y) ; the output for successful unification
         Fail) ; fallback
```

Here, we unify two expressions(parent $x Bob)and(parent Tom $y), and return a tuple($x $y)if unification succeeded. TheFailatom will be returned if there are no matches. Note that(unify (A $x) ($x B) Yes No)will be reduced toNo, because$xshould have the same binding in both patterns (and it cannot beAandBsimultaneously).

One of the first two arguments can be a reference to a Space as well. In this case, it will work likematchbut with an alternative option in the case of failed matching:

```
(Parent Tom Bob)
(Parent Bob Ann)
! (unify &self (Parent $x Bob) $x Fail) ; [Tom]
```

Here, we pass a reference to the current Space as the first argument, so the second expression(parent $x Bob)is matched against the whole set of declared knowledge.

Chained unification■

Let us analyze how(conclude (Parent Bob $y))from the above example is evaluated.

At first,(match &self; (= (Parent Bob $y) $result) $result)is executed to evaluate the subexpression. But this query returns no result, because equalities forParentare not defined. Thus,(Parent Bob $y)remains unreduced.

Thus, the equality query for the whole expression(match &self; (= (conclude (Parent Bob $y)) $result) $result)is executed. The following two expressions (one is the query and another one is from Space) are unifiable:

```
(= (conclude (Parent Bob $y))
   $result)
(= (conclude       $A)
   (match &self (Implies $A $B)
          (match &self $A $B)))
```

$Awill be bound to(Parent Bob $y), and$resultwill be

```
(match &self (Implies (Parent Bob $y) $B)
             (match &self (Parent Bob $y) $B))
```

matchis executed directly as a grounded function (otherwise another equality query would be constructed) with(Implies (Parent Bob $y) $B)as a query. It unifies with the following entry in the Space:

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

One may notice that there could be some collisions of variable names, and the interpreter should deal with this. In overall,$xgets bound toBob, and$Bgets bound to(Child $y Bob). Since the output of thismatchis(match &self; (Parent Bob $y) $B), the expression for further evaluation becomes(match &self; (Parent Bob $y) (Child $y Bob)).

(Parent Bob $y)unifies with(Parent Bob Ann)yielding(Child Ann Bob)

Query(= (Child Ann Bob) $result)finds no matches, so(Child Ann Bob)is the final result.

The overall chain of transformations in the course of interpretation can be viewed as:

```
1. (conclude (Parent Bob $y))
2. (match &self (Implies (Parent Bob $y) $B)
          (match &self (Parent Bob $y) $B))
3. (match &self (Parent Bob $y) (Child $y Bob))
4. (Child Ann Bob)
```

These are not all the steps done by the interpreter, but they give the overall picture of what is really going on under the hood.

Source: https://metta-lang.dev/docs/learn/tutorials/ground_up/nested_queries.html

Nested queries and recursive graph traversal■

Composite queries■

We've already seen queries forconcludeanddeduce, which result is another query. At the same time, chaining of queries can be done in a more functional style with equalities as it could be done for

```
(= (get-grand-parents $x)
   ((get-parents (get-parents $x))))
```

Keeping knowledge declarative can be useful for implementing reasoning over it.

Imagine that we add more info on people like(Female Pam)or(Male Tom)into the knowledge base, and want to define more relations such assister. One can turn facts into equalities like(= (Female Pam) True)and use functional logic (as in thefrog example), but let us keep simple facts for now.

One can add more functions likeget-parents. A function forfemalewould be more convenient to represent as a filter, e.g.(= (female $x) (match &self; (Female $x) $x)), so it will be composable, e.g.(= (get-mother $x $y) (female (get-parents $x $y))).

One can do this by a composite query instead.

```
(Female Pam)
(Male Tom)
(Male Bob)
(Female Liz)
(Female Pat)
(Female Ann)
(Male Jim)
(Parent Tom Bob)
(Parent Pam Bob)
(Parent Tom Liz)
(Parent Bob Ann)
(Parent Bob Pat)
(Parent Pat Jim)

(= (get-sister $x)
   (match &self
     (, (Parent $y $x)
        (Parent $y $z)
        (Female $z))
     $z
   )
)
! (get-sister Bob)
```

Composite queries contain a few patterns (united by,into one expression), which should be satisfied simultaneously. Such queries can be efficient if the Atomspace query engine efficiently processes joints. This can be important for large knowledge bases. Otherwise, it is necessary to be careful about the order of nested queries of filters. For example, having(Female $z)with free variable$zas the innermost functional call or(match &self; (Female $z) ...)as the outermost query in a nested sequence of queries will be highly inefficient, because it will first extract all the females from the knowledge base, and only then will narrow down the set of the results.

Notice that the above program is imprecise. How can the mistake be fixed (check the sister ofLiz- one option would be to introduce(different $x $y)as a filter)? You can try implementing other relations likeunclein the above program or rewrite it in a functional way. Typically, we would like to represent such concepts using other derived concepts rather than monolithic composite queries (e.g. "Uncle is a brother of a parent" rather than "Uncle is a male child of a parent of a parent, but not the parent").

Recursion for graph traversal■

Let us define the predecessor relation:

```
 For any `$x` and `$z`: `$x` is a predecessor of `$z`
    if there is `$y` such that
       `$y` is a parent of `$z` and
       `$x` is a predecessor of `$y`
```

Recursion is a convenient way to represent such relations.

```
(Parent Tom Bob)
(Parent Pam Bob)
(Parent Tom Liz)
(Parent Bob Ann)
(Parent Bob Pat)
(Parent Pat Jim)
(Parent Jim Lil)

(= (parent $x $y) (match &self (Parent $x $y) $x))
(= (predecessor $x $z) (parent $x $z))
(= (predecessor $x $z) (predecessor $x (parent $y $z)))
; Who are predecessors of Lil
! (predecessor $x Lil)
```

Source: https://metta-lang.dev/docs/learn/tutorials/types_basics/intro.html

Basics of Types and Metatypes■

Table of Contents■

Source: https://metta-lang.dev/docs/learn/tutorials/types_basics/atom_types.html

Concrete types■

Types of symbols■

Atoms in MeTTa are typed. Types of atoms are also represented as atoms (typically, symbolic atoms and expressions). Expressions of the form(:)are used to assign types. For example, to designate that the symbol atomahas a custom typeAone needs to add the expression(: a A)to the space (program).

Note that sinceAhere is a symbol atom, it can also have a type, e.g.,(: A Type). The symbol atomTypeis conventionally used in MeTTa to denote the type of type atoms. However, it is not assigned automatically. That is, declaration(: a A)doesn't forceAto be of typeType.

When an atom has no assigned type, it has%Undefined%type. The value of%Undefined%type can be type-checked with any type required.

One can check the type of an atom withget-typefunction from stdlib.

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

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

Here, we declared typesAandBforaandbcorrespondingly, and typeTypeforA.get-typereturns the declared types or%Undefined%if no type information is provided for the symbol.

Types of expressions■

Consider the following program.

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

The type of expression(a b)will be(A B). The type of a tuple is a tuple of types of its elements. However, what if we want to apply a function to an argument? Usually, we want to check if the function argument is of appropriate type. Also, while function applications themselves are expressions, they are transformed in the course of evaluation, and the result has its own type. Basically, we want to be able to transform (or reduce) types of expressions before or without transforming expressions themselves.

Arrow->is a built-in symbol of the type system in MeTTa, which is used to create a function type, for example(: foo (-> A B)). This type signature says thatfoocan accept an argument of typeAand its result will be of typeB:

```
(: a A)
(: foo (-> A B))
! (get-type (foo a)) ; B
```

Let us note that

We didn't provide a body forfoo, so(foo a)is not reduced at all, and its typeBis derived purely from the types offooanda. It doesn't matter whetherfoois a real function or a data constructor.

Equality queries themselves don't care about the position of the function symbol in the tuple, and the following code is perfectly correct

```
(= ($1 infix-f $2) ($2 $1))
```

```
! (match &self (= (1 infix-f 2) $r) $r)
```

However, reduction of the type of a tuple is performed if itsfirstelement has an arrow (function) type. For convenience and by convention, the first element in a tuple is treated specially for function application.

Type-checking■

Types can protect against incorrectly constructed expressions including misuse of a function, when we want it to accept arguments of a certain type.

```
; This function accepts an atom of type A and returns an atom of type B
(: foo (-> A B))
(: a A)
(: b B)

! (foo a) ; no error
! (get-type (foo b)) ; no result
! (b foo) ; notice: no error
! (get-type (b foo)) ; (B (-> A B))
! (foo b) ; type error
```

We didn't define an equality forfoo, so(foo a)reduces to itself. However, an attempt to evaluate(foo b)results in the error expression. When we try to get the type of this expression with(get-type (foo b)), the result is empty meaning that this expression has no valid type.

Notice that evaluation of(b foo)doesn't produce an error. The arrow type offooin the second position of the tuple doesn't cause transformation of its type. Indeed,(get-type (b foo))produces(B (-> A B)).

Gradual typing■

Let us consider what types will expressions have, when some of their elements are%Undefined%. Run the following program to check the currently implemented behavior

```
(: foo (-> A B))
(: a A)
! (get-type (foo c))
! (get-type (g a))
```

Note thatgandcare of%Undefined%type, whilefooandaare typed. The result can be different depending on which type is not defined, of the function or its argument.

Multiple arguments■

Functions can have more than one argument. In their type signature, types of their parameters are listed first, and the return type is put at the end much like for functions with one argument.

The wrong order of arguments with different types as well as the wrong number of arguments will render the type of the whole expression to be empty (invalid).

```
; This function takes two atoms of type A and B and returns an atom of type C
(: foo2 (-> A B C))
(: a A)
(: b B)

! (get-type (foo2 a b)) ; C
! (get-type (foo2 b a)) ; empty
! (get-type (foo2 a)) ; empty
! (foo2 a c) ; no error
! (foo2 b a) ; type error (the interpreter stops on error)
! (foo2 c) ; would also be type error
```

Here, the atomcis of%Undefined%type and it can be matched against an atom of any other type. Thus,(foo2 a c)will not produce an error. However,(foo2 c)will not work because of wrong arity.

Also notice that it is not necessary to define an instance of typeC.foo2by itself acts as a constructor for this type.

What will be the type of a function with zero arguments? Its type expression will have only the return type after->, e.g.

```
(: a A)
(: const-a (-> A))
(= (const-a) a)
```

Nested expressions■

Types of nested expressions are inferred from innermost expressions outside. You can try nesting typed expressions in the sandbox below and see what goes wrong.

```
(: foo (-> A B))
(: bar (-> B B A))
(: a A)
! (get-type (bar (foo a) (foo a)))
! (get-type (foo (bar (foo a) (foo a))))
```

Note that type signatures can be nested expressions by themselves:

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

! (get-type (foo-pair a b)) ; empty
! (get-type (foo-pair (a b))) ; C
```

As was mentioned above, an arrow type of the atom, which is not the first in the tuple, will not cause type reduction. Thus, one may apply a function to another function (or a data constructor):

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

! (get-type (foo bar)) ; C
! (get-type (foo (bar a))) ; empty
```

Here, the type ofbarmatches the type of the first parameter offoo. Thus,(foo bar)is a well-typed expression, which overall type corresponds to the return type offoo, namely,C.

(foo (bar a)), in turn, is badly typed, because the type of(bar a)is reduced toB, which does not correspond to(-> A B)expected byfoo.

Similarly, the return type of a function can be an arbitrary expression including arrow types. Try to construct a well-typed expression involving all the following symbols

```
(: foo (-> C (-> A B)))
(: bar (-> B A))
(: a A)
(: c C)

! (get-type (...))
```

We intentionally don't provide function bodies here to underline that typing imposes purely structural restrictions on expressions, which don't require understanding the semantics of functions. In the example above,fooaccepts an atom of typeC. Thus,(foo c)is well-typed, and its reduced type is(-> A B). This is an arrow type meaning that we can put this expression at the first position of a tuple (function application), and it will expect an atom of typeA. Thus,((foo c) a)should be well-typed, and its reduced type will beB. Thus, we can applybarto it. Will(bar ((foo c) a))be indeed well-typed?

Grounded atoms■

Grounded atoms are also typed. One can check their types withget-typeas well:

```
! (get-type 1) ; Number
```

```
! (get-type 1.1) ; Number
! (get-type +) ; (-> Number Number Number)
! (get-type (+ 1 2.1)) ; Number
```

As the example shows,1and1.1both are ofNumbertype, although their data-level representation can be different.+accepts two arguments ofNumbertype and returns the result of the same type. Thus,Numberis repeated three times in its type signature.

Let us note once again that the argument ofget-typeis not evaluated, andget-typereturns an inferred type of expression. In particular, when we try to apply+to the argument of a wrong type, the result is the error expression (which by itself is well-typed), butget-typereturns the empty result instead of returning the type of the error message:

```
(: a A)
! (get-type (+ 1 a)) ; empty
! (get-type (+ 1 b)) ; Number
! (+ 1 b) ; no error, not reduced
! (+ 1 a) ; type error
```

In this program, we also tried to see the type of application of the grounded function to the argument of%Undefined%type. Such the expression type-checks. However, it is not reduced in the course of evaluation. Thus, grounded functions work as partial functions or expression constructors in such cases. MeTTa is a symbolic language, and the possibility to construct expressions for further analysis is one of its main features. Ultimately, grounded functions should not differ from symbolically defined functions in this regard.

Source: https://metta-lang.dev/docs/learn/tutorials/types_basics/parametric_types.html

Recursive and parametric types■

Recursive data types■

All types allow constructing recursive expressions, when there is at least one function accepting and returning values of this type. This is true for arithmetic expressions or compositions of operations over strings. Say, any expression like(+ (- 3 1) (* 2 (+ 3 4)))will be ofNumbertype. We expect that the result of evaluation of such expressions will have the same type as the reduced type of the expression itself.

However, in some cases, we don't even want such expressions to be reduced, but want to consider them as instances of the reduced type. Consider the simple example of Peano numbers:

```
(: Z Nat) ; Z is "zero"
(: S (-> Nat Nat)) ; S "constructs" the next number
! (S Z) ; this is "one"
! (S (S Z)) ; this is "two"
! (get-type (S (S (S Z)))) ; Nat
! (get-type (S S)) ; not Nat
```

We didn't define the type ofNatitself. One may prefer to add(: Nat Type)for clarity.

In the code above,Sdoes nothing. It could be a grounded function, which adds1to the given number in some binary representation. Instead,(S some-nat)is not reduced and serves itself to represent the next natural number. It doesn't actually important thatSis not a function, and(S some-nat)is not calculated. In fact, it could be. What really matters is that instances ofNatcan be deconstructed and pattern-matched.

The following code shows, howNatas a recursive data type is processed by pattern matching.

```
(: Z Nat)
(: S (-> Nat Nat))
(: Greater (-> Nat Nat Bool))
(= (Greater (S $x) Z)
   True)
(= (Greater Z $x)
   False)
(= (Greater (S $x) (S $y))
   (Greater $x $y))
! (Greater (S Z) (S Z)) ; False
! (Greater (S (S Z)) (S Z)) ; True
```

While this implementation is inefficient for computations, it is more suitable for reasoning.

More practical use of recursive data structures is in the form of containers to store data. We already constructed them in the previous tutorials, but without types. Let us add typing information and define the type of list of numbers:

```
(: NilNum ListNum)
(: ConsNum (-> Number ListNum ListNum))
! (get-type (ConsNum 1 (ConsNum 2 (ConsNum 3 NilNum)))) ; ListNum
! (ConsNum 1 (ConsNum "S" NilNum)) ; BadType
```

The type reduction for such expressions is rather straightforward: the type of(ConsNum 3 NilNum)is reduced toListNum, sinceConsNumis of(-> Number ListNum ListNum)type and its arguments are ofNumberandListNumtypes. Consequently,(ConsNum 2 (...))is reduced toListNumagain for the same reason, and so on. For the second case,(ConsNum "S" NilNum)is badly typed.

Such expressions can be recursively processed as was done in thetutorial. Adding type information makes the purpose of the corresponding functions clearer and allows detecting mistakes.

Parametric types■

Type expressions can contain variables. Type-checking for such types is implemented and can be understood via pattern-matching. Let us consider some basic examples.

Stdlib contains a comparison operator==. The following code

```
! (get-type ==)
! (== 1 "S")
```

will reveal that(== 1 +)is badly typed, and the reason is that==has the type(-> $t $t Bool). This means that the arguments can be of an arbitrary but same type. Type-checking and reduction can be understood here as an attempt to unify(-> $t $t Bool)with(-> Number String $result).

It deserves noting that the output type can also be variable, e.g.

```
(: apply (-> (-> $tx $ty) $tx $ty))
(= (apply $f $x) ($f $x))
! (apply not False) ; True
! (get-type (apply not False)) ; Bool
! (unify (-> (-> $tx  $ty)  $tx  $ty)
         (-> (-> Bool Bool) Bool $result)
         $result
         BadType) ; Bool
! (apply not 1) ; BadType
```

nothas(-> Bool Bool)type andFalseis ofBooltype. Thus, arguments of(apply not False)suppose that the function type signature should be unified with(-> (-> Bool Bool) Bool $result). This results in binding both$txand$tytoBool, and the output type ($ty) also becomesBool.

In thetutorial, we definedapply-twice, which takes the function as an argument and applies it two time to the second argument. But what if the output type of the function is different from its input type? Can it be applied to the result of its own application? Try to specify the type ofapply-twiceto catch the error in the last expression:

```
(: apply-twice (-> ? ? ?))
(= (apply-twice $f $x)
   ($f ($f $x)))
(: greater-than-0 (-> Number Bool))
(= (greater-than-0 $x) (> $x 0))
! (get-type (apply-twice not True)) ; should be [Bool]
! (get-type (apply-twice greater-than-0 1)) ; should be []
```

Besides defining higher-order functions, parametric types are useful for recursive data structures. One of the most common examples isList. How can we define it as a container of elements of an arbitrary but same type? We can parameterize the typeListitself with the type of its elements:

```
(: Nil (List $t))
(: Cons (-> $t (List $t) (List $t)))
! (get-type (Cons 1 (Cons 2 Nil)))
! (get-type (Cons False (Cons True Nil)))
! (get-type (Cons + (Cons - Nil)))
! (get-type (Cons True (Cons 1 Nil)))
```

Let us consider how the type of(Cons 2 Nil)is derived. These arguments ofConssuppose its type signature to be undergo the following unification:

```
! (unify (-> $t     (List $t) (List $t))
         (-> Number (List $t) $result)
         $result
         BadType)
```

$tgets bound toNumber, and the output type(List $t)becomes(List Number).

Then, the outerConsin(Cons 1 (Cons 2 Nil))receives the arguments of typesNumberand(List Number), which can be simultaneously unified with$tand(List $t)producing(List Number)as the output type once again.

In contrast, the outerConsin(Cons True (Cons 1 Nil))receivesBooland(List Number). Apparently,$tin(-> $t (List $t) (List $t))cannot be bound to bothBoolandNumberresulting in type error.

Functions can receive arguments of parametric types, and type-checking will help to catch possible mistakes. Consider the following example

```
(: Nil (List $t))
(: Cons (-> $t (List $t) (List $t)))
(: first (-> (List $t) $t))
(: append (-> (List $t) (List $t) (List $t)))
! (get-type
    (+ 1
      (first (append (Cons 1 Nil)
                     (Cons 2 Nil)))))
```

We don't need function bodies for type-checking.firstreturns the first element of(List $t)-typed list, and this element should be of$ttype.appendconcatenates two lists with elements of the same type and produces the list of elements of this type as well. When we start considering a specific expression and unify types of its elements with type signatures of corresponding functions, variables in types get bindings. Apparently, types of(Cons 1 Nil)and(Cons 2 Nil)are reduced to(List Number). Then,(append (...) (...))gets the same type, while the type of(first (...))is reduced toNumber. You can experiment with making the expression badly typed in the code above and see, at which point the error is detected.

Functional programming with types is discussed in more detail inthis tutorial. However, types in MeTTa are more general than generalized algebraic data types and are similar to dependent types. The use of such advanced types is elaborated inthis tutorial, in particular, in application to knowledge representation and reasoning.

Source: https://metta-lang.dev/docs/learn/tutorials/types_basics/metatypes.html

Metatypes■

Peeking into metatypes■

In MeTTa, we may need to analyze the structure of atoms themselves. Thetutorialstarts with introducing four kinds of atoms -Symbol,Expression,Variable,Grounded. We refer to them asmetatypes. One can useget-metatypeto retrieve the metatype of an atom

```
! (get-metatype 1) ; Grounded
! (get-metatype +) ; Grounded
! (get-metatype (+ 1 2)) ; Expression
! (get-metatype a) ; Symbol
! (get-metatype (a b)) ; Expression
! (get-metatype $x) ; Variable
```

How to process atoms depending on their metatypes is discussed in another tutorial. In this tutorial, we discuss one particular metatype, which is widely utilized in MeTTa to control the order of evaluation. You should have noticed that arguments of some functions are not reduced before the function is called. This is true forget-typeandget-metatypefunctions. Let us check their type signatures:

```
! (get-type get-type) ; (-> Atom Atom)
! (get-type get-metatype) ; (-> Atom Atom)
```

Here,Atomis a supertype forSymbol,Expression,Variable,Grounded. While metatypes can appear in ordinary type signatures, they should not be assigned explicitly, e.g.(: a Expression), except for the following special case.

Atomis treated specially by the interpreter - if a function expects an argument ofAtomtype, this argument is not reduced before passing to the function. This is why, say,(get-metatype (+ 1 2))returnsExpression. It is worth noting thatAtomas a return result will have no special effect. WhileAtomas the return type could prevent the result from further evaluation, this feature is not implemented in the current version of MeTTa.

Using arguments ofAtomtype is essential for meta-programming and self-reflection in MeTTa. However, it has a lot of other more common uses.

Quoting MeTTa code■

We encountered error expressions. These expressions can contain unreduced atoms, becauseErrorexpects the arguments ofAtomtype:

```
! (get-type Error) ; (-> Atom Atom ErrorType)
! (get-metatype Error) ; just Symbol
! (get-type (Error Foo Boo)) ; ErrorType
! (Error (+ 1 2) (+ 1 +)) ; arguments are not evaluated
```

Erroris not a grounded atom, it is just a symbol. It doesn't even have defined equalities, so it works just an expression constructor, which prevents its arguments from being evaluated and which has a return type, which can be used to catch errors.

Another very simple constructor from stdlib isquote, which is defined just as(: quote (-> Atom Atom)). It does nothing except of wrapping its argument and preventing it from being evaluated.

```
! (get-type quote)
! (quote (+ 1 2))
! (get-type if)
```

Some programming languages introducequoteas a special symbol known by the interpreter (otherwise its argument would be evaluated). Consequently, any term should be quoted, when we want to avoid evaluating it. However,quoteis an ordinary symbol in MeTTa. What is specially treated is theAtommetatype for arguments. It appears to be convenient not only for extensive work

with MeTTa programs in MeTTa itself (for code generation and analysis, automatic programming, meta-programming, genetic programming and such), but also for implementing traditional control statements.

ifunder the hood■

As was mentioned in thetutorial, theifstatement in MeTTa works much like if-then-else construction in any other language.ifis not an ordinary function and typically requires a special treatment in interpreters or compilers to avoid evaluation of branches not triggered by the condition.

However, its implementation in MeTTa can be done with the following equalities

```
(= (if True $then $else) $then)
(= (if False $then $else) $else)
```

The trick is to have the type signature with the first argument typedBool, and the next two arguments typedAtom. The first argument typedBoolcan be an expression to evaluate like(> a 0), or aTrue/Falsevalue. TheAtom-types arguments$thenand$elsewill not be evaluated while passing into theiffunction. However, once theif-expression has been reduced to either of them, the interpreter will chain its evaluation to obtain the final result.

Consider the following example

```
(: my-if (-> Bool Atom Atom Atom))
(= (my-if True $then $else) $then)
(= (my-if False $then $else) $else)
(= (loop) (loop))
(= (OK) OK!)
! (my-if (> 0 1) (loop) (OK))
```

If you comment out the type definition, then the program will go into an infinite loop trying to evaluate all the arguments ofmy-if. Lazy model of computation could automatically postpone evaluation of$thenand$elseexpressions until they are not required, but it is not currently implemented.

Can you imagine how a "sequential and" function can be written, which evaluates its second argument, only if the first argument isTrue?

```
(: seq-and (-> ... ... Bool))
(= (seq-and ... ...) ...)
(= (seq-and ... ...) ...)
(: loop (-> Bool Bool))
! (seq-and False (loop)) ; should be False
! (seq-and True True) ; should be True
```

Apparently, in the proposed setting, the first argument should be evaluated, so its type should beBool, while the second argument shouldn't be immediately evaluated. What will be the whole solution?

Transforming expressions■

One may want to useAtom-typed arguments not only for just avoiding computations or quoting expressions, but to modify them before evaluation.

Let us consider a very simple example with swapping the arguments of a function. The code below will give-7as a result

```
(: swap-arguments-atom (-> Atom Atom))
(= (swap-arguments-atom ($op $arg1 $arg2))
   ($op $arg2 $arg1)
)
! (swap-arguments-atom (- 15 8))
```

At the same time, the same code without typing will not work properly and will return[(swap-arguments 7)], because(- 15 8)will be reduced by the interpreter before passing to theswap-argumentsand will not be pattern-matched against($op $arg1 $arg2)

```
(= (swap-arguments ($op $arg1 $arg2))
   ($op $arg2 $arg1)
)
! (swap-arguments (- 15 8))
```

One more example of using theAtomtype is comparing expressions

```
; `atom-eq` returns True, when arguments are identical
; (can be unified with the same variable)
(: atom-eq (-> Atom Atom Bool))
(= (atom-eq $x $x) True)

; These expressions are identical:
! (atom-eq (+ 1 2) (+ 1 2))

; the following will not be reduced because the expressions are not the same
; (even though the result of their evaluation would be)
! (atom-eq 3 (+ 1 2))
```

Source: https://metta-lang.dev/docs/learn/tutorials/types_basics/match_control.html

Controlling pattern matching■

Both standard and custom functions in MeTTa can haveAtom-typed arguments, which will not be reduced before these functions are evaluated. But we may want to call them on a result of another function call. What is the best way to do this? Before answering this question, let us considermatchin more detail.

Type signature of thematchfunction■

Pattern matching is the core operation in MeTTa, and it is implemented using thematchfunction, which locates all atoms in the given Space that match the provided pattern and generates the output pattern.

Let us recall that thematchfunction has three arguments:

a grounded atom referencing a Space;

a pattern to be matched against atoms in the Space (query);

an output pattern typically containing variables from the input pattern.

Consider the type ofmatch:

```
! (get-type match)
```

The second and the third arguments are ofAtomtype. Thus, the input and the output pattern are passed tomatchas is, without reduction. Preventing reduction of the input pattern is essentially needed for the possibility to use any pattern for matching. The output pattern is instantiated bymatchand returned, and only then it is evaluated further by the interpreter.

in-and-out behavior ofmatch■

In the following example,(Green $who)is evaluated toTruefor$whobound toToddue to the corresponding equality.

```
(Green Sam)
(= (Green Tod) True)
! ($who (Green $who)) ; (Tod True)
! (match &self (Green $who) $who) ; Sam
```

However,(Green $who)is not reduced when passed tomatch, and the query returnsSam, without utilizing the equality because(Green Sam)is added to the Space.

Let us verify that the result ofmatchwill be evaluated further. In the following example,matchfirst finds two entries satisfying the pattern(Green $who)and instantiates the output pattern on the base of each of them, but only(Frog Sam)is evaluated toTrueon the base of one available equality, while(Frog Tod)remains unreduced.

```
(Green Sam)
(Green Tod)
(= (Frog Sam) True)
! (match &self (Green $who) (Frog $who)) ; [True, (Frog Tod)]
```

We can verify that instantiation of the output pattern happens before its evaluation:

```
(Green Sam)
(= (Frog Sam) True)
! (match &self (Green $who) (quote (Frog $who)))
```

Here,(Green $who)is matched against(Green Sam),$whogets bound toSam, and then it is substituted to the output pattern yielding(quote (Frog Sam)), in which(Frog Sam)is not reduced further toTrue, becausequotealso expectsAtom. Thus,matchcan be thought of as transformation of the input pattern to the output pattern. It performs no additional evaluation of patterns by itself.

Returning output patterns with substituted variables before further evaluation is very convenient for nested queries. Consider the following example:

```
(Green Sam)
(Likes Sam Emi)
(Likes Tod Kat)
! (match &self (Green $who)
    (match &self (Likes $who $x) $x))
! (match &self (Green $who)
    (match &self (Likes $boo $x) $x))
! (match &self (Likes $who $x)
    (match &self (Green $x) $x))
! (match &self (Likes $who $x)
    (match &self (Green $boo) $boo))
```

The output of the outer query is another query. The inner query is not evaluated by itself, but instantiated as the output of the outer query.

In the first case,$whogets bound toSamand the pattern in the second query becomes(Frog Sam $x), which has only one match, so the output isEmi.

In the second case,$whois not used in the inner query, and there are two results, because the pattern of the second query remains(Likes $boo $x).

In the third case, there are no results, because the outer query produces two results, but neither(Green Emi)nor(Green Kat)are in the Space.

In the last case,Samis returned two times. The outer query returns two results, and although its variables are not used in the inner query, it is evaluated twice.

Patterns are not type-checked■

Functions withAtom-typed parameters can accept atoms of any other type, including badly typed expressions, which are not supposed to be reduced. As it was mentioned earlier, this behavior can be useful in different situations. Indeed, why couldn't we, say, quote a badly typed expression as an incorrect example?

It should be noted, though, that providing specific types for function parameters and simultaneously indicating that the corresponding arguments should not be reduced could be useful in other cases. Unfortunately, it is currently not possible to provide a specific type and a metatype simultaneously (which isone of the known issues).

At the same time,matchis a very basic function, which should not be restricted in its ability to both accept and return "incorrect" expressions. Thus, one should keep in mind thatmatchdoes not perform type-checking on its arguments, which is intentional and expected.

The following program contains a badly typed expression, which can still be pattern-matched (andmatchcan accept a badly typed pattern):

```
(+ 1 False)
! (match &self (+ 1 False) OK) ; OK
! (match &self (+ 1 $x) $x) ; False
```

It can be useful to deal with "wrong" MeTTa programs on a meta-level in MeTTa itself, so this behavior ofmatchallows us to write code that analyzes badly typed expressions within MeTTa.

Type of=■

MeTTa programs typically contain many equalities. But is there a guarantee that the function will indeed return the declared type? This is achieved by requiring that both parts of equalities are of the same type. Consider the following code:

```
(: foo (-> Number Bool))
(= (foo $x) (+ $x 1))
! (get-type (foo $x)) ; Bool
! (get-type (+ $x 1)) ; Number
```

```
! (get-type =) ; (-> $t#nnnn $t#nnnn Atom)
! (= (foo $x) (+ $x 1)) ; BadType
```

We declared the type offooto be(-> Number Bool). On the base of this definition, the type of(foo $x)can be reduced toBool, which is the expected type of its result. However, the type of its body(+ $x 1)is reduced toNumber. If we get the type of=, we will see that both its arguments should be of the same type. The result type of=isAtom, since it is not a function (unless someone adds an equality over equalities, which is permissible). If one tries to "execute" this equality, it will indeed return the type error.

Programs can contain badly typed expressions as we discussed earlier. However, this may permit badly defined functions.! (pragma! type-check auto)can be used to enable automatic detection of such errors:

```
! (pragma! type-check auto) ; ()
(: foo (-> Number Bool))
(= (foo $x) (+ $x 1)) ; BadType
```

This pragma option turns on type-checking of expressions before adding them to the Space (without evaluation of the expression itself).

let's evaluate■

Sometimes we need to evaluate an expression before passing it to a function, which expectsAtom-typed arguments. What is the best way to do this?

One trick could be to write a wrapper function like this

```
(= (call-by-value $f $arg)
   ($f $arg))
! (call-by-value quote (+ 1 2)) ; (quote 3)
```

Arguments of this function are not declared to be ofAtomtype, so they are evaluated before the function is called. Then, the function simply passes its evaluated argument to the given function. However, it is not needed to write such a wrapper function, because there is a more convenient way with the use of operationletfrom stdlib.

lettakes three arguments:

a variable atom (or, more generally, a pattern)

an expression to be evaluated and bound to the variable (or, more generally, matched against the pattern in the first argument)

the output expression (which typically contains a variable to be substituted)

```
! (let $x (+ 1 2) (quote $x)) ; (quote 3)
(: Z Nat)
! (get-metatype (get-type Z)) ; (get-type Z) is Expression
! (let $x (get-type Z) (get-metatype $x)) ; Nat is Symbol
```

One may also want to evaluate some subexpression before constructing an expression for pattern-matching

```
(= (age Bob) 5)
(= (age Sam) 8)
(= (age Ann) 3)
(= (age Tom) 5)
(= (of-same-age $who)
   (let $age (age $who)
        (match &self (= (age $other) $age)
               $other)))
! (of-same-age Bob) ; [Bob, Tom]
; without `of-same-age`:
! (let $age (age Bob)
       (match &self (= (age $other) $age)
              $other)) ; also [Bob, Tom]
! (match &self (= (age $other) (age Bob))
```

```
         $other) ; does not pattern-match
; evaluating the whole pattern is a bad idea
! (let $pattern (= (age $other) (age Bob))
       $pattern) ; [(= 5 5), (= 8 5), (= 5 5), (= 3 5)]
! (let $pattern (= (age $other) (age Bob))
       (match &self $pattern $other)) ; does not pattern-match
```

It can be seen thatlethelps to evaluate(age Bob)before constructing a pattern for retrieval. However, evaluating the whole pattern is typically a bad idea. That is why patterns inmatchare ofAtomtype, andletis used when something should be evaluated beforehand.

As was remarked before,letcan accept a pattern instead of a single variable. More detailed information onlettogether with other functions from stdlib are provided inthe next tutorial.

Unit type■

Unitis a type that has exactly one possible valueunitserving as a return value for functions, which return "nothing". However, from the type-theoretic point of view, mappings to the empty set do not exist (they are non-constructive), while mappings to the one-element set do exist, and returning the only element of this set yields zero information, that is, "nothing". This is equivalent tovoidin such imperative languages as C++.

In MeTTa, the empty expression()is used for the unit value, which is the only instance of the type(->). A function, which doesn't return anything meaningful but which is still supposed to be a valid function, should return()unless a custom unit type is defined for it.

In practice, this()value is used as the return type for grounded functions with side effects (unless these side effects are not described in a special way, e.g., with monads). For example, the functionadd-atomadds an atom to the Space, and returns().

When it is necessary to execute such a side-effect function and then to return some value, or to chain it with subsequent execution of another side-effect function, it is convenient to use the following construction based onlet:(let () (side-effect-function) (evaluate-next)). If(side-effect-function)returns(), it is matched with the pattern()inlet-expression (one can use a variable instead of()as well), and then(evaluate-next)is executed.

Let us consider a simple knowledge base for a personal assistant system. The knowledge base contains information about the tasks the user is supposed to do. A new atom in this context would be a new task.

```
(= (message-to-user $task)
   (Today you have $task))
(= (add-task-and-notify $task)
    (let () (add-atom &self (TASK $task))
            (message-to-user $task))
)
! (get-type add-atom) ; (-> hyperon::space::DynSpace Atom (->))
! (add-task-and-notify (Something to do))
! (match &self (TASK $t) $t) # (Somthing to do)
```

Theadd-task-and-notifyfunction adds a$taskatom into the current Space using theadd-atomfunction and then calls another function which returns a message to notify the user about the new task. Please, notice the type signature ofadd-atom.

Source: https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/intro.html

Standard Library Overview■

In this section we will look at the main functions of the standard library, which are part of the standard distribution of MeTTa.

Table of Contents■

Source:
https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/basic_grounded_functions.html

Basic grounded functions■

Arithmetic operators andNumbertype■

Arithmetic operations in MeTTa are grounded functions and use the prefix notation where the operator comes before the operands. MeTTa arithmetic works with atoms ofNumbertype, which can store floating-point numbers as well as integers under the hood, and you can mix them in your calculations. The type of binary arithmetic operations is(-> Number Number Number).

```
; Addition
! (+ 1 3) ; 4

; Subtraction
! (- 6 2.2) ; 3.8

; Multiplication
! (* 7.3 9) ; 65.7

; Division
! (/ 25 5) ; 5 or 5.0

; Modulus
! (% 24 5) ; 4
```

In the current implementation arithmetic operations support only two numerical arguments, expressions with more than two arguments like!(+ 1 2 3 4)will result in a type error (IncorrectNumberOfArguments). One should use an explicit nested expression in that case

```
! (+ 1 (+ 2 (+ 3 4))) ; 10
! (- 8 (/ 6.4 4)) ; 6.4
```

Numbers in MeTTa are presented as grounded atoms with the predefinedNumbertype. Evaluation of ill-typed expressions produces an error expression. Notice, however, that arithmetic expressions with atoms of%Undefined%type will not be reduced.

```
! (+ 2 S) ; (+ 2 S)
! (+ 2 "8") ; BadType
```

Other common mathematical operations likesqr,sqrt,abs,pow,min,max,log2,ln, etc. are not included in the standard library as grounded symbols at the moment. But they can beimported from Python directly.

Comparison operations■

Comparison operations implemented in stdlib are also grounded operations. There are four operations<,>,<=,>=of(-> Number Number Bool)type.

```
; Less than
! (< 1 3)

; Greater than
! (> 3 2)

; Less than or equal to
! (<= 5 6.2)

; Greater than or equal to
! (>= 4 (+ 2 (* 3 5)))
```

Once again, passing ordinary symbols to grounded operations will not cause errors, and the expression simply remains unreduced, if it type-checks. Thus, it is generally a good practice to

ensure the types of atoms being compared are what the comparison operators expect to prevent unexpected results or errors.

```
! (> $x (+ 8 2)) ; Inner expression is reduced, but the outer is not
! (>= 4 (+ Q 2)) ; Reduction stops in the inner expression
(: R CustomType)
! (>= 4 R) ; BadType
```

The==operation is implemented to work with both grounded and symbol atoms and expressions (while remaining a grounded operation). Its type is(-> $t $t Bool). Its arguments are evaluated before executing the operation itself.

```
! (== 4 (+ 2 2)) ; True
! (== "This is a string" "Just a string") ; False
! (== (A B) (A B)) ; True
! (== (A B) (A (B C))) ; False
```

Unlike,==will not remain unreduced if one of its arguments is grounded, while another is not. Instead, it will returnFalseif the expression is well-typed.

```
! (== 4 (+ Q 2)) ; False
(: R CustomType)
! (== 4 R) ; BadType
```

Logical operations andBooltype■

Logical operations in MeTTa can be (and with some build options are) implemented purely symbolically. However, the Python version of stdlib contains their grounded implementation for better interoperability with Python. In particular, numeric comparison operations directly execute corresponding operations in Python and wrap the resultingboolvalue into a grounded atom. The grounded implementation is intended for subsymbolic and purely functional processing, while custom logic systems for reasoning are supposed to be implemented symbolically in MeTTa itself.

Logical operations in stdlib deal withTrueandFalsevalues ofBooltype, and have signatures(-> Bool Bool)and(-> Bool Bool Bool)for unary and binary cases.

```
; Test if both the given expressions are True
! (and (> 4 2) (== "This is a string" "Just a string")) ; False

; Test if any of the given expressions is True
! (or (> 4 2) (== "This is a string" "Just a string")) ; True

; Negates the result of a given Bool value
! (not (== 5 5)) ; False
! (not (and (> 4 2) (< 4 3))) ; True
```

Source: https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/console_output.html

Console output and debugging■

All values obtained during evaluation of the MeTTa program or script are collected and returned. The whole program can be treated as a function. If a stand-alone program is executed viaa command-line runnerorREPLthese results are printed at the end. This printing will not happen if MeTTa is used viaits API.

However, MeTTa has two functions to send information to the console output:println!andtrace!. They can be used by developers for displaying messages and logging information during the evaluation process, in particular, for debugging purposes.

Print a line■

Theprintln!function is used to print a line of text to the console. Its type signature is(-> %Undefined% (->)).

The function accepts only a single argument, but multiple values can be printed by enclosing them within parentheses to form a single atom:

```
! (println! "This is a string")
! (println! ($v1 "string" 5))
```

Note thatprintln!returns theunitvalue(). Beside printing to stdout, the program will return two units due toprintln!evaluation.

The argument ofprintln!is evaluated beforeprintln!is called (its type is notAtombut%Undefined%), so the following code

```
(Parent Bob Ann)
! (match &self (Parent Bob Ann) (Ann is Bob`s child))
! (println! (match &self (Parent Bob Ann) (Bob is Ann`s parent)))
```

will print(Bob is Ann's parent)to stdout. Note that this result is printed before all the evaluation results (starting with thematchexpressions) are returned.

Trace log■

trace!accepts two arguments, the first is the atom to print, and the second is the atom to return. Both are evaluated before passing totrace!, which type is(-> %Undefined% $a $a), meaning that the reduced type of the wholetrace!expression is the same as the reduced type of the second argument:

```
! (get-type (trace! (Expecting 3) (+ 1 2))) ; Number
```

trace!can be considered as a syntactic sugar for the following construction usingprintln!andlet

```
(: my-trace (-> %Undefined% $a $a))
(= (my-trace $out $res)
   (let () (println! $out) $res))
! (my-trace (Expecting 3) (+ 1 2))
```

It can be used as a debugging tool that allows printing out a message to the terminal, along with valuating an atom.

```
(Parent Bob Ann)
! (trace! "Who is Anna`s parent?" ; print this expression
   (match &self (Parent $x Ann)
          ($x is Ann`s parent)))  ; return the result of this expression
!(trace! "Who is Bob`s child?"  ; print this expression
   (match &self (Parent Bob $x)
          ($x is Bob`s child))) ; return the result of this expression
```

The first argument does not have to be a pure string, which makestrace!work fine on its own

```
(Parent Bob Ann)
! (trace! ((Expected: (Bob is Ann`s parent))
           (Got: (match &self (Parent $x Ann) ($x is Ann`s parent)))
          )
   ())
```

Quote■

Quotation wasalready introducedas a tool for evaluation control. Let us recap thatquoteis just a symbol with(-> Atom Atom)type without equalities (i.e., a constructor). In some versions of MeTTa and its stdlib,quotecan be defined as(= (quote $atom) NotReducible), where the symbolNotReducibleexplicitly tells the interpreter that the expression should not be reduced.

The following is the basic example of the effect ofquote:

```
(Fruit apple)
(= (fruit $x)
   (match &self (Fruit $x) $x))
! (fruit $x) ; apple
! (quote (fruit $x)) ; (quote (fruit $x))
```

There is a useful combination oftrace!,quote, andletfor printing an expression together with its evaluation result, which is then returned.

```
(: trace-eval (-> Atom Atom))
(= (trace-eval $expr)
   (let $result $expr
        (trace! (EVAL: (quote $expr) --> $result)
                $result)))
(Fruit apple)
(= (fruit $x)
   (match &self (Fruit $x) $x))
; (EVAL: (quote (fruit $x)) --> apple) is printed to stdout
! (Overall result is (trace-eval (fruit $x))) ; (Overall result is apple)
```

In this code,trace-evalaccepts$exprofAtomtype, so it is not evaluated before getting totrace-eval.(let $result $expr ...)stores the result of evaluation of$exprinto$result, and then prints both of them usingtrace!((quote $expr)is used to avoid reduction of$exprbefore passing totrace!) and returns$result. The latter allows wrappingtrace-evalinto other expressions, which results in the behavior, which would take place without such wrapping, except for additional console output.

Another pattern of usingtrace!withquoteandletis to add tracing to the function itself. We first calculate the result (if needed), and then usetrace!to print some debugging information and return the result:

```
(= (add-bin $x)
   (let $r (+ $x 1)
        (trace! (quote ((add-bin $x) is $r))
                $r)))
(= (add-bin $x)
   (trace! (quote ((add-bin $x) is $x))
           $x))
; (quote ((add-bin 1) is 1)) and (quote ((add-bin 1) is 2)) will be printed
! (add-bin 1) ; [1, 2]
```

Without quotation an atom such as(add-bin $x)evaluated fromtrace!would result in an infinite loop, butquoteprevents the wrapped atom from being interpreted.

In the following code(test 1)would be evaluated fromtrace!and would result in an infinite loop

```
(= (test 1) (trace! (test 1) 1))
(= (test 1) (trace! (test 0) 0))
! (test 1)
```

Asserts■

MeTTa has a couple of assert operations that allow a program to check if a certain condition is true and return an error-expression if it is not.

assertEqualcompares (sets of) results of evaluation of two expressions. Its type is(-> Atom Atom Atom), so it interprets expressions internally and can compare erroneous expressions. If sets of results are equal, it outputs the unit value().

```
(Parent Bob Ann)
! (assertEqual
   (match &self (Parent $x Ann) $x)
   (unify (Parent $x Ann) (Parent Bob $y) $x Failed)) ; ()
! (assertEqual (+ 1 2) 3) ; ()
! (assertEqual (+ 1 2) (+ 1 4)) ; Error-expression
```

WhileassertEqualis convenient when we have two expressions to be reduced to the same result, it is quite common that we want to check if the evaluated expression has a very specific result. Imagine the situation when one wants to be sure that some expression, say(+ 1 x), isnotreduced. It will make no sense to use(assertEqual (+ 1 x) (+ 1 x)).

Also, if the result of evaluation is nondeterministic, and the set of supposed outcomes is known, one would need to turn this set into a nondeterministic result as well in order to useassertEqual. It can be done withsuperpose, but both issues are covered by the following assert function.

assertEqualToResulthas the same type asassertEqual, namely(-> Atom Atom Atom), and it evaluates the first expression. However, it doesn't evaluate the second expression, but considers it a set of expected results of the first expression.

```
(Parent Bob Ann)
(Parent Pam Ann)
! (assertEqualToResult
    (match &self (Parent $x Ann) $x)
    (Bob Pam))  ; ()
(= (bin) 0)
(= (bin) 1)
! (assertEqualToResult (bin) (0 1)) ; ()
! (assertEqualToResult (+ 1 2) (3)) ; ()
! (assertEqualToResult
    (+ 1 untyped-symbol)
   ((+ 1 untyped-symbol))) ; ()
! (assertEqualToResult (+ 1 2) ((+ 1 2))) ; Error
```

Let us notice a few things:

We have to take the result into brackets, e.g.,(assertEqualToResult (+ 1 2) (3))vs(assertEqual (+ 1 2) 3), because the second argument ofassertEqualToResultis a set of results even if this set contains one element.

As a consequence, a non-reducible expression also gets additional brackets as the second argument, e.g.,((+ 1 untyped-symbol)). It is also a one-element set of the results.

The second argument is indeed not evaluated. The last assert yields an error, because(+ 1 2)is reduced to3. Notice3as what we got instead of expected (for the sake of the example)(+ 1 2).

Source:
https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/superpose_collapse.html

Handling nondeterministic results■

Superpose■

In previous tutorials we saw thatmatchalong with any other function can return multiple (nondeterministic) as well as empty results. If you need to get a nondeterministic result explicitly, use thesuperposefunction, which turns a tuple into a nondeterministic result. It is an stdlib function of(-> Expression Atom)type.

However, it is typically recommended to avoid using it. For example, in the following program

```
(= (bin) 0)
(= (bin) 1)
(= (bin2) (superpose (0 1)))
! (bin) ; [0, 1]
! (bin2) ; [0, 1]
```

binandbin2do similar job. However,binis evaluated using one equality query, whilebin2requires additional evaluation ofsuperpose. Also, one may argue thatbinis more modular and more suitable for meta-programming and evaluation control.

One may want to usesuperposeto execute several operations. However, the order of execution is not guaranteed. And again, one can try thinking about writing multiple equalities for a function, inside whichsuperposeseems to be suitable.

However,superposecan still be convenient in some cases. For example, one can pass nondeterministic expressions to any function (both grounded and symbolic, built-in and custom) and get multiple results. In the following example, writing a nondeterministic function returning3,4,5would be inconvenient:

```
! (+ 2 (superpose (3 4 5))) ; [5, 6, 7]
```

Here, nondeterminism works like a map over a set of elements.

Another example, where usingsuperposeexplicitly is useful is for checking a set of nondeterministic results withassertEqual, when both arguments still require evaluation (soassertEqualToResultis not convenient to apply). In the following example, we want to check that we didn't forget any equality for(color), but we may not be interested what exact value they are reduced to (i.e., whether(ikb)is reduced tointernational-klein-blueor something else).

```
(= (ikb) international-klein-blue)
(= (color) green)
(= (color) yellow)
(= (color) (ikb))

!(assertEqual
  (match &self (= (color) $x) $x)
  (superpose ((ikb) yellow green))) ; ()
!(assertEqualToResult
   (match &self (= (color) $x) $x)
  ((ikb) yellow green)) ; Error
```

Empty■

As mentioned above, in MeTTa, functions can return empty results. This is a natural consequence on the evaluation semantics based on queries, which can find no matches. Sometimes, we may want to force a function to "return" an empty result to abort a certain evaluation branch, or to explicitly represent it to analyze this behavior on the meta-level.

(superpose ())will exactly return the empty set of results. However, stdlib provide(empty)function to do the same in a clearer and stable way. Some versions may also useEmptyas a symbol to inform the interpreter about the empty result, which may differ on some level from calling a grounded function, which really returns an empty set.(empty)is supported more widely at the moment, so we use it here.

(empty)could be useful in the construction of the asserts(assertEqual (...) (empty)), but(assertEqualToResult (...) ())can also work.

```
(Parent Bob Ann)
! (assertEqual
    (match &self (Parent Tom $x) $x)
    (empty)) ; ()
! (assertEqualToResult
    (match &self (Parent Tom $x) $x)
    ()) ; ()
```

Since expressions without suitable equalities remain unreduced in MeTTa,(empty)can be used to alter this behavior, when desirable, e.g.

```
(= (eq $x $x) True)
! (eq a b) ; (eq a b)
(= (eq $x $y) (empty))
! (eq a b) ; no result
```

(empty)can be used to turn a total function such asiforunifyinto a partial function, when we have no behavior for the else-branch, and we don't want the expression to remain unreduced.

Let us note that there is some convention in how the interpreter processes empty results. If the result ofmatchfor equality query is empty, the interpreter doesn't reduce the given expression (it transforms the empty result of such queries toNotReducible), but if a grounded function returns the empty result, it is treated as partial. When a grounded function application is not reduced, e.g.(+ 1 undefined-symbol), because the function returns not the empty result, butNotReducible. This behavior may be refined in the future, but the possibility to have both types of behavior (a partial function is not reduced and evaluation continues or it returns no result stopping further evaluation) will be supported.

From nondeterministic viewpoint,(empty)removes an evaluation branch. If we consider all the results as a collection,(empty)can be used for its filtering. In the following program,(color)and(fruit)produce nondeterministic "collections" of colors and fruits correspondingly, whilefilter-preferis a partially defined id function, which can be used to filter out these collections.

```
(= (color) red)
(= (color) green)
(= (color) blue)
(= (fruit) apple)
(= (fruit) banana)
(= (fruit) mango)
(= (filter-prefer blue) blue)
(= (filter-prefer banana) banana)
(= (filter-prefer mango) mango)
(= (filter-prefer $x) (empty))
! (filter-prefer (color)) ; [blue]
! (filter-prefer (fruit)) ; [mango, banana]
```

In case of recursion,(empty)can prune branches, which don't satisfy some conditions as shown inthis example.

Collapse■

Nondeterminism is an efficient way to map and filter sets of elements as well as to perform search. However, nondeterministic branches do not "see" each other, while we may want to get the extreme element or just to count them (or, more generally, fold over them). That is, we may need to collect the results in one evaluation branch.

Reverse operation tosuperposeiscollapse, which has the type(-> Atom Expression). It converts a nondeterministic result into a tuple.

collapseis a grounded function, which runs the interpreter on the given atom and wraps the returned results into an expression.

```
(= (color) red)
(= (color) green)
(= (color) blue)
! (color) ; three results: [blue, red, green]
! (collapse (color)) ; one result: [(blue red green)]
```

Here we've got a nondeterministic result[blue, red, green]from thecolorfunction and converted it into one tuple[(blue red green)]usingcollapse.

Thesuperposefunction reverts thecollapseresult

```
(= (color) green)
(= (color) yellow)
(= (color) red)
! (color) ; [green, yellow, red]
! (collapse (color)) ; [(green yellow red)]
! (let $x (collapse (color))
    (superpose $x))  ; [green, yellow, red]
! (superpose (1 2 3)) ; [1, 2, 3]
! (collapse (superpose (1 2 3)))
! (let $x (superpose (1 2 3)) ; [(1 2 3)]
    (collapse $x))    ; [(1), (2), (3)]
```

The color function gives the nondeterministic result[green, yellow, red](the order of colors may vary). Thecollapsefunction converts it into a tuple[(green yellow red)]. And finally thesuperposefunction inletconverts a tuple back into the nondeterministic result[red, green, yellow]. The order of colors may change again due to nondeterminism.

Note that we cannot callcollapseinsidesuperpose, becausecollapsewill not be executed before passing tosuperposeand will be considered as a part of the input tuple. In contrary, we cannot callsuperposeoutsidecollapse, because it will causecollapseto be called separately for each nondeterministic branch produced by superpose instead of collecting these branches insidecollapse.

Source:
https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/working_with_spaces.html

Working with spaces■

Space API■

Spaces can have different implementations, but should satisfy a certain API. This API includes pattern-matching (or unification) functionality.matchis an stdlib function, which calls a corresponding API function of the given space, which can be different from the program space.

Let us recap that the type ofmatchis(-> hyperon::space::DynSpace Atom Atom %Undefined%). The first argument is a space (or, more precisely, a grounded atom referring to a space) satisfying the Space API. The second argument is the input pattern to be unified with expressions in the space, and the third argument is the output pattern, which is instantiated for every found match.matchcan produce any number of results starting with zero, which are treated nondeterministically.

The basic use ofmatchwas already covered before, while its use with custom spaces will be described in other tutorials, since these spaces are not the part of stdlib. However, the Space API includes additional components, which are utilized by such stdlib functions asadd-atomandremove-atom.

Adding atoms■

The content of spaces can be not only defined statically in MeTTa scripts, but can also be modified at runtime by programs residing in the same or other spaces.

The functionadd-atomadds an atom into the Space. Its type is(-> hyperon::space::DynSpace Atom (->)). The first argument is an atom referring some Space, to which an atom provided as the second argument will be added. Since the type of the second argument isAtom, the added atom is added as is without reduction.

In the following program,add-foo-eqis a function, which adds an equality forfooto the program space whenever called. Then, it is checked that the expressions are added to the space without reduction.

```
(: add-foo-eq (-> Atom (->)))
(= (add-foo-eq $x)
   (add-atom &self (= (foo) $x)))
! (foo) ; (foo) - not reduced
! (add-foo-eq (+ 1 2)) ; () - OK
! (add-foo-eq (+ 3 4)) ; () - OK
! (foo) ; [3, 7]
! (match &self (= (foo) $x)
    (quote $x)) ; [(quote (+ 1 2)), (quote (+ 3 4))]
```

If it is desirable to add a reduced atom without additional wrappers (e.g., likeadd-foo-eqbut withoutAtomtype for the argument), thenadd-reductcan be used:

```
! (add-reduct &self (= (foo) (+ 3 4))) ; ()
! (foo) ; 7
! (match &self (= (foo) $x)
    (quote $x)) ; (quote 7)
```

Removing atoms■

The functionremove-atomremoves an atom from the AtomSpace without reducing it. Its type is(-> hyperon::space::DynSpace Atom (->)).

The first argument is a reference to the space from which the Atom needs to be removed, the second is the atom to be removed. Notice that if the given atom is not in the space,remove-atomcurrently neither raises a error nor returns the empty result.

```
(Atom to remove)
! (match &self (Atom to remove) "Atom exists") ; "Atom exists"
! (remove-atom &self (Atom to remove)) ; ()
! (match &self (Atom to remove) "Unexpected") ; nothing
! (remove-atom &self (Atom to remove)) ; ()
```

Combination ofremove-atomandadd-atomcan be used forgraph rewriting. Consider the following example.

```
(link A B)
(link B C)
(link C A)
(link C E)

! (match &self (, (link $x $y)
                  (link $y $z)
                  (link $z $x))
               (let () (remove-atom &self (link $x $y))
                       (add-atom &self (link $y $x)))
  ) ; [(), (), ()]
! (match &self (link $x $y)
    (link $x $y)) ; [(link A C), (link C B), (link B A), (link C E)]
```

Here, we find entries(link _ _), which form three-element loops, and revert the direction of links in them. Let us note thatmatchreturns three unit results, because the loop can start from any of such entries. All of them are reverted (only(link C E)remains unchanged). Also, in the current implementation,matchfirst finds all the matches, and then instantiates the output pattern with them, which is evaluated outsidematch. Ifremove-atomandadd-atomwould be executed right away for each found matching, the condition of circular links would be broken after the first rewrite. This behavior can be space-specific, and is not a part of MeTTa specification at the moment. This can be changed in the future.

New spaces■

It is possible to create other spaces with the use ofnew-spacefunction from stdlib. Its type is(-> hyperon::space::DynSpace), so it has no arguments and returns a fresh space. Creating new spaces can be useful to keep the program space cleaner, or to simplify queries.

If we just run(new-space)like this

```
! (new-space)
```

we will get something likeGroundingSpace-0x10703b398as a textual representation space atom. But how can we refer to this space in other parts of the program? Notice that the following code will not work as desired

```
(= (get-space) (new-space))
! (add-atom (get-space) (Parent Bob Ann)) ; ()
! (match (get-space) (Parent $x $y) ($x $y)) ; nothing
```

because(get-space)will create a brand new space each time.

One workaround for this issue in a functional programming style is to wrap the whole program into a function, which accepts a space as an input and passes it to subfunctions, which need it:

```
(= (main $space)
   (let () (add-atom $space (Parent Bob Ann))
     (match $space (Parent $x $y) ($x $y))
   )
)
! (main (new-space)) ; (Bob Ann)
```

This approach has its own merits. However, a more direct fix for(= (get-space) (new-space))would be just to evaluate(new-space)before adding it to the program:

```
! (add-reduct &self (= (get-space) (new-space))) ; ()
! (add-atom (get-space) (Parent Bob Ann)) ; ()
! (get-space) ; GroundingSpace-addr
```

```
! (match (get-space) (Parent $x $y) ($x $y)) ; (Bob Ann)
```

That is,(new-space)is evaluated to a grounded atom, which wraps a newly created space. Other elements of(= (get-space) (new-space))are not reduced. Instead ofadd-reduct, one could use the following more explicit code

```
! (let $space (new-space)
    (add-atom &self (= (get-space) $space)))
```

which also ensured that nothing is reduced except(new-space).

Creating tokens■

Why can't we refer to the grounded atom, which wraps the created space? Indeed, we can represent such grounded atoms as numbers or operations over them in the code. And what is about&self;?

In fact, they are turned into atoms from their textual representation by the parser, which knows a mapping from textual tokens (defined with the use of regular expressions) to constructors of corresponding grounded atom. Basically,&selfis; replaced with the grounded atom wrapping the program space by the parser before it gets inside the interpreter.

Parsing is explained in more detail inanother tutorial, while here we focus on the stdlib functionbind!.

bind!registers a new token which is replaced with an atom during the parsing of the rest of the program. Its type is(-> Symbol %Undefined% (->)).

The first argument has typeSymbol, so technically we can use any valid symbol as the token name, but conventionally the token should start with&, when it is bound to a custom grounded atom, to distinguish it from symbols. The second argument is the atom, which is associated with the token after reduction. This atom should not necessarily be a grounded atom.bind!returns the unit value()similar toprintln!oradd-atom.

Consider the following program:

```
(= (get-hello) &hello)
! (bind! &hello (Hello world)) ; ()
! (get-metatype &hello) ; Expression
! &hello ; (Hello world)
! (get-hello) ; &hello
```

We first define the function(get-hello), which returns the symbol&hello.; Then, we bind the token&helloto; the atom(Hello world). Note that the metatype of&helloisExpression;, because it is replaced by the parser and gets to the interpreter already as(Hello world).! &hellois; expectedly(Hello world). Once again,&hellois; not reduced to(Hello world)by the interpreter. It is replaced with it by the parser. It can be seen by the fact that(get-hello)returns&helloas; a symbol, because it was parsed and added to the program space beforebind!.

bind!might be tempting to use to refer to some lengthy constant expressions, e.g.

```
! (bind! &x (foo1 (foo2 3) 45 (A (v))))
! &x
```

However, this lengthy expression will be inserted to the program in place of every occurrence of&x.; However, let us note again that the second argument ofbind!is evaluated beforebind!is called, which is especially important with functions with side effects. For example, the following program will print"test"only once, while&reswill; be simply replaced with().

```
! (bind! &res (println! "test"))
! &res
! &res
```

Usingbind!for unique grounded atoms intensively used in the program can be more reasonable. Binding spaces created with(new-space)to tokens is one of possible use cases:

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

```
! (add-atom &space (Parent Bob Ann)) ; ()
! &space ; GroundingSpace-addr
! (match &space (Parent $x $y) ($x $y)) ; (Bob Ann)
! (match &self (Parent $x $y) ($x $y)) ; empty
```

However, if spaces are created dynamically depending on runtime data,bind!is not usable.

Imports■

Stdlib has operations for importing scripts and modules. One such operation isimport!. It accepts two arguments. The first argument is a symbol, which is turned into the token for accessing the imported module. The second argument is the module name. For example, the program from thetutorialcould be split into two scripts - one containing knowledge, and another one querying it.

```
; people_kb.metta
(Female Pam)
(Male Tom)
(Male Bob)
(Female Liz)
(Female Pat)
(Female Ann)
(Male Jim)
(Parent Tom Bob)
(Parent Pam Bob)
(Parent Tom Liz)
(Parent Bob Ann)
(Parent Bob Pat)
(Parent Pat Jim)
```

```
; main.metta
! (import! &people people_kb)
(= (get-sister $x)
   (match &people
     (, (Parent $y $x)
        (Parent $y $z)
        (Female $z))
     $z
   )
)
! (get-sister Bob)
```

Here,(import! &people; people_kb)looks similar to(bind! &people; (new-space)), butimport!fills in the loaded space with atoms from the script. Let us note thatimport!does more work than just loading the script into a space. It interacts with the module system, which is described in another tutorial.

&selfcan; be passed as the first argument toimport!. In this case, the script or module will still be loaded into a separate space, but the atom wrapping this space will be inserted to&self.; Pattern matching queries encountering such atoms will delegate queries to them (with the exception, when the space atom itself matches against the query, which happens, when this query is just a variable, e.g.,$x). Thus, it works similar to inserting all the atoms to&self;, but with some differences, when importing the same module happens multiple times, say, in different submodules.

One may useget-atomsmethod to see that the empty MeTTa script is not that empty and contains the stdlib space(s). Note that the resultget-atomswill be reduced. Thus, it is not recommended to use in general.

```
! (get-atoms &self)
```

Some space atoms are present in the seemingly empty program since some modules are pre-imported. Indeed, one can find, say,ifdefinition in&self;, which actually resides in the stdlib space inserted into&selfas; an atom

```
! (match &self
         (= (if $cond $then $else) $result)
   (quote (= (if $cond $then $else) $result))
  )
```

mod-space!returns the space of the module (and tries to load the module if it is not loaded into the module system). Thus, we can explore the module space explicitly.

```
! (mod-space! stdlib)
! (match (mod-space! stdlib)
         (= (if $cond $then $else) $result)
   (quote (= (if $cond $then $else) $result))
  )
```

Source: https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/control_flow.html

Control flow■

MeTTa has several specific constructs that allow a program to execute different parts of code based either on pattern matching or logical conditions.

if■

ifwas already covered inthis tutorial. But let us recap it as a part of stdlib.

Theifstatement implementation in MeTTa can be the following function

```
(: if (-> Bool Atom Atom $t))
(= (if True $then $else) $then)
(= (if False $then $else) $else)
```

Here, the first argument (condition) isBool, which is evaluated before executing the equality-query forif.

The next two arguments are not evaluated and returned for the further evaluation depending on whether the first argument is matched withTrueorFalse.

The basic use ofifin MeTTa is similar to that in other languages:

```
(= (foo $x)
   (if (>= $x 0)
       (+ $x 10)
       (* $x -1)
   )
)
! (foo 1)  ; 11
! (foo -9) ; 9
```

Here we have a functionfoothat adds10to the input value if it's grater or equal0, and multiplies the input value by-1otherwise. The expression(>= $x 0)is the first argument of theiffunction, and it is evaluated to aBoolvalue. According to that value the expression(+ $x 10)or(* $x -1)is returned for the final evaluation, and we get the result.

In contrast to other languages, one can pass a variable toifand it will be matched against equalities with bothTrueandFalse. Consider the following example

```
! (if $x (+ 6 1) (- 7 2))
(= (foo $b $x)
   (if $b
       (+ $x 10)
       (* $x -1)
   )
)
! ((foo $b 1) $b) ; [(-1 False), (11 True)]
```

fooaccepts the condition forif, and when we pass a variable, both branches are evaluated with the corresponding binding for$b.

ifcan also remain unreduced:

```
! (if (> $x 0) (+ $x 5) (- $x 5))
```

In this expression,(> $x 0)remains unreduced. Its overall type isBool, but it can't be directly matched against neitherTruenorFalse. Thus, no equality is applied.

let■

lethas been briefly describedin another tutorial. Here, we will recap it.

Theletfunction is utilized to establish temporary variable bindings within an expression. It allows introducing variables, assign values to them, and then use these values within the scope of theletblock.

Once theletblock has run, these variables cease to exist and any previous bindings are re-established. Depending on the interpreter version,letcan be either a basic grounded function, or be implemented using other primitives. Let us consider its type

```
! (get-type let)
```

The first argument ofletis a pattern ofAtomtype, which is not evaluated. The second argument is the value, which is reduced before being passed toletThe third parameter is anAtomagain. An attempt to unify the first two arguments is performed. If it succeeds, the found bindings are substituted to the third argument, which is then evaluated. Otherwise, the empty result is returned.

Consider the following example:

```
(= (test 1) 1)
(= (test 1) 0)
(= (test 2) 2)

! (let $W (test $X) (println! ("test" $X => $W)))
```

The code above will print:

```
("test" 1 => 1)
("test" 1 => 0)
("test" 2 => 2)
```

and return three unit results produced byprintln!. It can be seen that variables from both the first and the second arguments can appear in the third argument.

The following example shows the difference between the first two arguments.

```
(= (test 1) 2)
! (let 2 (test 1) YES) ; YES
! (let (test 1) 2 NO ) ; empty
```

In case of(let 2 (test 1) YES),(test 1)is evaluated to2, and it can be unified with the first argument, which is also2. In case of(let (test 1) 2 NO),(test 1)is not reduced, and it cannot be unified (as a pattern) with2, so the overall result is empty.

This example also shows that variables are not mandatory inlet. What is needed is the possibility to unify the arguments. This allows usingletfor chaining operations, and this chaining can be conditional if the first operation returns some value, e.g.

```
(= (is-frog Sam) True)
(= (print-if-frog $x)
   (let True (is-frog $x)
     (println! ($x is frog!))))
! (print-if-frog Sam) ; ()
! (print-if-frog Ben) ; empty
```

Another basic use ofletis to calculate values for passing them to functions accepting arguments ofAtomtype, for example:

```
(Sam is 34 years old)
! (match &self ($who is (+ 20 14) years old) $who) ; empty
! (let $r (+ 20 14)
   (match &self ($who is $r years old) $who)) ; Sam
```

Since the first argument can be not only a variable or a concrete value, but also an expression,letcan be used for deconstructing expressions

```
(= (fact Sam) (age 34))
(= (fact Sam) (color green))
(= (fact Tom) (age 14))
! (let (age $r) (fact $who)
   ($who is $r)) ; [(Tom is 14), (Sam is 34)]
```

The branches not corresponding to the(age $r)pattern are filtered out.

let*■

When several consecutive substitutions are required,let*can be used for convenience. The first argument oflet*isExpression, which elements are the required substitutions, while the second argument is the resulting expression. In the following example, several values are subsequently calculated, andlet*allows making it more readable (notice also how pattern matching helps to calculate minimum and maximum values together with their absolute difference in oneif).

```
(Sam is 34)
(Tom is 14)
(= (person-by-age $age)
   (match &self ($who is $age) $who))
(= (persons-of-age $a $b)
   (let* (((($age-min $age-max $diff)
           (if (< $a $b)
               ($a $b (- $b $a))
               ($b $a (- $a $b))))
          ($younger (person-by-age $age-min))
          ($older   (person-by-age $age-max))
         )
         ($younger is younger than $older by $diff years))
)
! (persons-of-age 34 14)
```

Another case, for whichlet*can be convenient, is the consequent execution of side-effect functions, e.g.

```
(Sam is 34)
(= (age++ $who)
   (let* (($age (match &self ($who is $a) $a))
          ( ()  (println! (WAS: ($who is $age))))
          ( ()  (remove-atom &self ($who is $age)))
          ( ()  (add-reduct &self ($who is (+ $age 1))))
          ($upd (match &self ($who is $a) $a))
          ( ()  (println! (NOW: ($who is $upd)))))
         $upd
   )
)
! (age++ Sam) ; 35
```

case■

Another type of multiway control flow mechanism in MeTTa is thecasefunction, which was briefly mentioned inthe tutorial. It turnsletaround and subsequently tests multiple pattern-matching conditions for the given value. This value is provided by the first argument. While the formal argument type isAtom, it will be evaluated. The second argument is a tuple, which elements are pairs mapping condition patterns to results.

```
(Sam is Frog)
(Apple is Green)
(= (test $who)
   (case (match &self ($who is $x) $x)
    (
        (42   "The answer is 42!")
        (Frog "Do not ask me about frogs")
        ($a   ($who is $a))
    )))
! (test Sam) ; "Do not ask me about frogs"
! (test Apple) ; (Apple is Green)
! (test Car) ; empty
```

Cases are processed sequentially from the first to the last. In the example above,$acondition will always be matched, so it is put at the end, and the corresponding branch is triggered, when all the previous conditions are not met. Note, however, that$ais not matched against the empty result in

the last case.

In order to handle such cases, one can useEmptysymbol as a case pattern (in some versions of the interpreter,Emptyis the dedicated symbol which(empty)is evaluated to). The following code should return"Input was really empty":

```
! (case (empty)
    ((Empty "Input was really empty")
    ($_   "Should not be the case"))
  )
```

In the current version of MeTTa (v0.1.12),Emptycan be matched against a variable. This means that the result of (empty) can be matched against $. In this situation, if we want to catch Empty case, we need to place it before $case.

Let us consider the use of patterns incaseon example of the rock-paper-scissors game. There are multiple ways of how to write a function, which will return the winner. The following function uses onecasewith five branches:

```
(= (rps-winner $x $y)
   (case ($x $y)
     (((Paper Rock) First)
      ((Scissors Paper) First)
      ((Rock Scissors) First)
      (($a $a) Draw)
      ($_ Second))
   )
)
! (rps-winner Paper Scissors) ; Second
! (rps-winner Rock  Scissors) ; First
! (rps-winner Paper Paper   ) ; Draw
```

One could also write a function, which checks if the first player wins, and use it twice (for($x $y)and($y $x)). This could be more scalable for game extensions with additional gestures, and could be more robust to unexpected inputs (although this should be better handled with types). You can try experimenting with different approaches using the sandbox above.

Source: https://metta-lang.dev/docs/learn/tutorials/stdlib_overview/atoms_ops.html

Operations over atoms■

Stdlib contains operations to construct and deconstruct atoms as instances ofExpressionmeta-type. Let us first describe these operations.

Deconstructing expressions■

car-atomandcdr-atomare fundamental operations that are used to manipulate atoms. They are named after 'car' and 'cdr' operations in Lisp and other similar programming languages.

Thecar-atomfunction extracts the first atom of an expression as a tuple.

```
! (get-type car-atom) ; (-> Expression %Undefined%)
! (car-atom (1 2 3)) ; 1
! (car-atom (Cons X Nil)) ; Cons
! (car-atom (seg (point 1 1) (point 1 4))) ; seg
```

Thecdr-atomfunction extracts the tail of an expression, that is, all the atoms of the argument except the first one.

```
! (get-type cdr-atom) ; (-> Expression %Undefined%)
! (cdr-atom (1 2 3)) ; (2 3)
! (cdr-atom (Cons X Nil)) ; (X Nil)
! (cdr-atom (seg (point 1 1) (point 1 4))) ; ((point 1 1) (point 1 4))
```

Constructing expressions■

cons-atomis a function, which constructs an expression using two arguments, the first of which serves as a head and the second serves as a tail.

```
! (get-type cons-atom) ; (-> Atom Expression Expression)
! (cons-atom 1 (2 3)) ; (1 2 3)
! (cons-atom Cons (X Nil)) ; (Cons X Nil)
! (cons-atom seg ((point 1 1) (point 1 4))) ; (seg (point 1 1) (point 1 4))
```

cons-atomreverses the results ofcar-atomandcdr-atom:

```
(= (reconstruct $xs)
   (let* (($head (car-atom $xs))
          ($tail (cdr-atom $xs)))
     (cons-atom $head $tail))
)
! (reconstruct (1 2 3)) ; (1 2 3)
! (reconstruct (Cons X Nil)) ; (Cons X Nil)
```

Note that we needletin the code above, becausecons-atomexpects "meta-typed" arguments, which are not reduced. For example,cdr-atomwill not be evaluated in the following code:

```
! (cons-atom 1 (cdr-atom (1 2 3))) ; (1 cdr-atom (1 2 3))
```

Let us consider how basic recursive processing of expressions can be implemented:

```
(: map-expr (-> (-> $t $t) Expression Expression))
(= (map-expr $f $expr)
   (if (== $expr ()) ()
       (let* (($head (car-atom $expr))
              ($tail (cdr-atom $expr))
              ($head-new ($f $head))
              ($tail-new (map-expr $f $tail))
             )
         (cons-atom $head-new $tail-new)
       )
   )
)
! (map-expr not (False True False False))
```

Comparison with custom data constructors■

A typical way to construct lists using custom data structures is to introduce a symbol, which can be used for pattern-matching. Then, extracting heads and tails of lists becomes straightforward, and special functions for this are not need. They can be easily implemented via pattern-matching:

```
(= (car (Cons $x $xs)) $x)
(= (cdr (Cons $x $xs)) $xs)
! (cdr (Cons 1 (Cons 2 (Cons 3 Nil))))
```

But one can implement recursive processing withoutcarandcons:

```
(: map (-> (-> $t $t) Expression Expression))
(= (map $f Nil) Nil)
(= (map $f (Cons $x $xs))
   (Cons ($f $x) (map $f $xs)))
! (map not (Cons False (Cons True (Cons False (Cons False Nil)))))
```

Instead ofExpression, one would typically use a polymorphicListtype (as described another tutorial).

Implementingmapwith the use of pattern matching over list constructors is much simpler. Why can't it be made withcons-atom?cons-atom,car-atom,cdr-atomwork on the very base meta-level as grounded functions. If we introduced explicit constructors for expressions, then we would just move this meta-level further, and the question would arise how expressions with these new constructors are constructed. Apparently, we need to stop somewhere and introduce the very basic operations to construct all other composite expressions. Using explicit data constructors should typically be preferred over resorting to these atom-level operations.

Typical usage■

car-atomandcdr-atomare typically used for recursive traversal of an expression. One basic example is creation of lists from tuples. In case of reducible non-nested lists, the code is simple:

```
(= (to-list $expr)
   (if (== $expr ()) Nil
     (Cons (car-atom $expr)
           (to-list (cdr-atom $expr)))
   )
)
! (to-list (False (True False) False False))
```

Parsing a tuple of arbitrary length (if the use of explicit constructors is not convenient) is a good use case for operations with expressions. For example, one may try implementinglet*by subsequently processing the tuple of variable-value pairs and applyinglet.

One more fundamental use case for analyzing expressions is implementation of custom interpretation schemes, if they go beyond the default MeTTa interpretation process and domain specific languages. A separate tutorial will be devoted to this topic. But let us note here that combiningcar-atomandcdr-atomwithget-metatypewill be a typical pattern here. Here, we provide a simple example for parsing nested tuples:

```
(= (to-tree $expr)
   (case (get-metatype $expr)
      ((Expression
         (if (== $expr ()) Nil
             (Cons (to-tree (car-atom $expr))
                   (to-tree (cdr-atom $expr)))
         ))
       ($_ $expr)
      )
   )
)
! (to-tree (False (True False) False False))
```

Note the difference of the result withto-list. The internal(True False)is also converted to the list now. It happens because the head of the current tuple is also passed toto-tree. For this to work, we need

to analyze if the argument is an expression. If it is not, the value is not transformed.

Source: https://metta-lang.dev/docs/learn/tutorials/python_use/intro.html

Using MeTTa from Python■

Table of Contents■

Source: https://metta-lang.dev/docs/learn/tutorials/python_use/metta_python_basics.html

Running MeTTa in Python■

Introduction■

As Python has a broad range of applications, including web development, scientific and numeric computing, and especially AI, ML, and data analysis, its combined use with MeTTa significantly expands the possibilities of building AI systems. Both ways can be of interest:

embedding Python objects into MeTTa for serving as sub-symbolic (and, in particular, neural) components within a symbolic system;

using MeTTa from Python for defining knowledge, rules, functions, and variables which can be referred to in Python programs to create prompt templates for LLMs, logical reasoning, or compositions of multiple AI agents.

We start with the use of MeTTa from Python via high-level API, and then we will proceed to a tighter integration.

Setup■

Firstly, you need to have MeTTa's Python API installed as a Python package. MeTTa itself can be built from source with Python support and installed in the development mode in accordance with the instructions in thegithub repository. This approach is more involved, but it will yield the latest version with a number of configuration options.

However, for a quick start, hyperon package available viapipunder Linux or MacOs (possibly except for newest processors):

```
pip install hyperon
```

MeTTa runner class■

The main interface class for MeTTa in Python isMeTTaclass, which represents a runner built on top of the interpreter to execute MeTTa programs. It can be imported fromhyperonpackage and its instance can be created and used to run MeTTa code directly:

```
from hyperon import MeTTa
metta = MeTTa()
result = metta.run('''
   (= (foo) boo)
   ! (foo)
   ! (match &self (= ($f) boo) $f)
''')
print(result) # [[boo], [foo]]
```

The result ofrunis a list of results of all evaluated expressions (following the exclamation mark!). Each of this results is also a list (each containing one element in the example above). These results are not printed to the console bymetta.run. They are just returned. Thus, we print them in Python.

Let us note thatMeTTainstance preserve their program space afterrunhas finished. Thus,runcan be executed multiple times:

```
from hyperon import MeTTa
metta = MeTTa()
metta.run('''
■(Parent Tom Bob)
■(Parent Pam Bob)
■(Parent Tom Liz)
■(Parent Bob Ann)
''')
print(metta.run('!(match &self (Parent Tom $x) $x)')) # [[Liz, Bob]]
```

```
print(metta.run('!(match &self (Parent $x Bob) $x)')) # [[Tom, Pam]]
```

Parsing MeTTa code■

The runner has methods for parsing a program code instead of executing it. Parsing produces MeTTa atoms wrapped into Python objects (so they can be manipulated from Python). Creating a simple expression atom(A B)looks like

```
atom = metta.parse_single('(A B)')
```

Theparse_single()method parses only the next single token from the text program, thus the following example will give equivalent results

```
from hyperon import MeTTa
metta = MeTTa()
atom1 = metta.parse_single('(A B)')
atom2 = metta.parse_single('(A B) (C D)')
print(atom1) # (A B)
print(atom2) # (A B)
```

Theparse_all()method can be used to parse the whole program code given in the string and get the list of atoms

```
from hyperon import MeTTa
metta = MeTTa()
program = metta.parse_all('(A B) (C D)')
print(program) # [(A B), (C D)]
```

Accessing the program Space■

Let us recall that Atomspace (or just Space) is a key component of MeTTa. It is essentially a knowledge representation database (which can be thought of as a metagraph) and the associated MeTTa functions are used for storing and manipulating information.

One can get a reference to the current program Space, which in turn may be accessed directly, wrapped in some way, or passed to the MeTTa interpreter. Having the reference, one can add new atoms into it using theadd_atom()method

```
metta.space().add_atom(atom)
```

Now let us call therun()method that runs the code from the program string containing a symbolic expression

```
from hyperon import MeTTa
metta = MeTTa()
atom = metta.parse_single('(A B)')
metta.space().add_atom(atom)
print(metta.run("!(match &self (A $x) $x)")) # [[B]]
```

The program passed toruncontains only one expression!(match &self; (A $x) $x). It calls thematchfunction for the pattern(A $x)and returns all matches for the$xvariable. The result will be[[B]], which means thatadd_atomhas added(A B)expression extracted from the string byparse_single. The code

```
atom = metta.parse_single('(A B)')
metta.space().add_atom(atom)
```

is effectively equivalent to

```
metta.run('(A B)')
```

because expressions are not preceded by!are just added to the program Space.

Please note that

```
atom = metta.parse_all('(A B)')
```

is not precisely equivalent to

```
metta.run('! (A B)')[0]
```

Although the results can be identical, the expression passed torunwill be evaluated and can get reduced:

```
from hyperon import MeTTa
metta = MeTTa()
print(metta.run('! (A B)')[0])    # [(A B)]
print(metta.run('! (+ 1 2)')[0])  # [3]
print(metta.parse_all('(A B)'))   # [(A B)]
print(metta.parse_all('(+ 1 2)')) # [(+ 1 2)]
```

parse_singleorparse_allare more useful, when we want not to add atoms to the program Space, but when we want to get these atoms without reduction and to process them further in Python.

Besidesadd_atom(andremove_atomas well), Space objects havequerymethod.

```
metta = MeTTa()
metta.run('''
■(Parent Tom Bob)
■(Parent Pam Bob)
■(Parent Tom Liz)
■(Parent Bob Ann)
''')
pattern = metta.parse_single('(Parent $x Bob)')
print(metta.space().query(pattern)) # [{ $x <- Pam }, { $x <- Tom }]
```

In contrast tomatchin MeTTa itself,querydoesn't take the output pattern, but just returns options for variable bindings, which can be useful for further custom processing in Python. It would be useful to have a possibility to define patterns directly in Python instead of parsing them from strings.

MeTTa atoms in Python■

ClassAtomin Python (see itsimplementation) is used to wrap all atoms created in the backend of MeTTa into Python objects, so they can be manipulated in Python. An atom of any kind (metatype) can be created as an instance of this class, but classesSymbolAtom,VariableAtom,ExpressionAtomandGroundedAtomtogether with helper functions are inherited fromAtomfor convenience.

Symbol atoms are intended for representing both procedural and declarative knowledge entities for fully introspective processing. Such symbolic representations can be used and manipulated to infer new knowledge, make decisions, and learn from experience. It's a way of handling and processing abstract and arbitrary information.

The helper functionS()is a convenient tool to construct an instance ofSymbolAtomPython class. Its only specific method isget_name, since symbols are identified by their names. All instances ofAtomhasget_metatypemethod, which returns the atom metatype maintained by the backend.

```
from hyperon import S, SymbolAtom, Atom
symbol_atom = S('MyAtom')
print(symbol_atom.get_name()) # MyAtom
print(symbol_atom.get_metatype()) # AtomKind.SYMBOL
print(type(symbol_atom)) # SymbolAtom
print(isinstance(symbol_atom, SymbolAtom)) # True
print(isinstance(symbol_atom, Atom)) # True
```

Let us note thatS('MyAtom')is a direct way to construct a symbol atom without calling the parser as inmetta.parse_single('MyAtom'). It allows constructing symbols with the use of arbitrary characters, which can be not accepted by the parser.

AVariableAtomrepresents a variable (typically in an expression). It serves as a placeholder that can be matched with, or bound to other Atoms.V()is a convenient method to construct aVariableAtom:

```
from hyperon import V
var_atom = V('x')
print(var_atom) # $x
print(var_atom.get_name()) # x
```

```
print(var_atom.get_metatype()) # AtomKind.VARIABLE
print(type(var_atom)) # VariableAtom
```

VariableAtomalso hasget_namemethod. Please note that variable names don't include$prefix in internal representation. It is used in the program code for the parser to distinguish variables and symbols.

AnExpressionAtomis a list of Atoms of any kind, including expressions. It has theget_children()method that returns a list of all children Atoms of an expression.E()is a convenient method to construct expressions, it takes a list of atoms as an input. The example below shows that queries can be constructed in Python and the resulting expressions can be processed in Python as well.

```
from hyperon import E, S, V, MeTTa

metta = MeTTa()
expr_atom = E(S('Parent'), V('x'), S('Bob'))
print(expr_atom) # (Parent $x Bob)
print(expr_atom.get_metatype()) # AtomKind.EXPR
print(expr_atom.get_children()) # [Parent, $x, Bob]
# Let us use expr_atom in the query
metta = MeTTa()
metta.run('''
■(Parent Tom Bob)
■(Parent Pam Bob)
■(Parent Tom Liz)
■(Parent Bob Ann)
''')
print(metta.space().query(expr_atom)) # [{ $x <- Pam }, { $x <- Tom }]
result = metta.run('! (match &self (Parent $x Bob) (Retrieved $x))')[0]
print(result) # [(Retrieved Tom) (Retrieved Pam)]
# Ignore 'Retrieved' in expressions and print Pam, Tom
for r in result:
■print(r.get_children()[1])
```

GroundedAtom■

GroundedAtomis a special subtype ofAtomthat makes a connection between the abstract, symbolically represented knowledge within AtomSpace and the external environment or the behaviors/actions in the outside world. Grounded Atoms often have an associated piece of program code that can be executed to produce specific output or trigger an action.

For example, this could be used to pull in data from external sources into the AtomSpace, to run a PyTorch model, to control an LLM agent, or to perform any other action that the system needs to interact with the external world, or just to perform intensive computations.

Besides the content, which aGroundedAtomwraps, there are three other aspects which can be customized:

the type of GroundedAtom (kept within the Atom itself);

the matching algorithm used by the Atom;

aGroundedAtomcan be made executable, and used to apply sub-symbolic operations to other Atoms as arguments.

Let us start with basic usage.G()is a convenient method to construct aGroundedAtom. It can accept any Python object, which hascopymethod. In the program below, we construct an expression with a custom grounded atom and add it to the program Space. Then, we perform querying to retrieve this atom.GroundedAtomhasget_object()method to extract the data wrapped into the atom.

```
from hyperon import *
metta = MeTTa()
entry = E(S('my-key'), G({'a': 1, 'b': 2}))
metta.space().add_atom(entry)
result = metta.run('! (match &self (my-key $x) $x)')[0][0]
```

```
print(type(result)) # GroundedAtom
print(result.get_object()) # {'a': 1, 'b': 2}
```

As the example shows, we can add a custom grounded object to the space, query and get it in MeTTa, and retrieve back to Python.

However, wrapping Python object directly toG()is typically not recommended. Python API for MeTTa implements a generic classGroundedObjectwith the fieldcontentstoring a Python object of interest and thecopymethod. There are two inherited classes,ValueObjectandOperationObjectwith some additional functionality. MethodsValueAtomandOperationAtomis a sugared way to constructG(ValueObject(...))andG(OperationObject(...))correspondingly. Thus, it would be preferable to useValueAtom({'a': 1, 'b': 2})in the code above, although one would need to writeresult.get_object().contentto access the corresponding Python object (ValueObjecthas a gettervalueforcontentas well, whileOperationObjectusesopfor this).

TheGroundedObjectconstructor takes acontentargument (a Python object) to wrap into a grounded atom. It also optionally accepts anidargument to represent the atom and to compare atoms if using thecontentfor this purpose isn't ideal. TheValueObjectclass provides a getter methodvalueto return the content of the grounded atom.

Arguments of theOperationObjectconstructor includename,op, andunwrap.nameserves as theidfor the grounded atom,op(a function) defining the operation is used as thecontentof the grounded atom, andunwrap(a boolean, optional) indicates whether to unwrap theGroundedAtomcontent when applying the operation (see more onuwraponthe next pageof this tutorial).

While there is a choice whether to useValueAtomandOperationAtomclasses for custom objects or to directly wrap them intoG, grounded objects constructed in the MeTTa code are returned as such sugared atoms:

```
from hyperon import *
metta = MeTTa()
calc = metta.run('! (+ 1 2)')[0][0]
print(type(calc.get_object())) # ValueObject
print(calc.get_object().value) # 3

metta.run('(my-secret-symbol 42)') # add the expression to the space
pattern = E(V('x'), ValueAtom(42))
print(metta.space().query(pattern)) # { $x <- my-secret-symbol }
```

As can be seen from the example,ValueAtom(42)can be matched against42appeared in the MeTTa program (although it is not recommended to use grounded atoms as keys for querying).

It should be noted, however, that stdlib operations in MeTTa are not Python operations. While atoms wrapping objects of such privitive types asNumberare automatically converted into Python objects (so one can get Python3from calculations!(+ 1 2)in MeTTa), the following code will cause a error, because the grounded operation wrapped by+atom is not a Python operation.

```
plus = metta.parse_single('+')
plus.get_object()
```

There is a module calledpy_ops, which replaces some basic operations with Python operations, so the following code works:

```
from hyperon import *
metta = MeTTa()
metta.run("! (import! &self py_ops)")
plus = metta.parse_single('+')
print(type(plus.get_object())) # OperationObject
print(plus.get_object().op) # some lambda
print(plus.get_object()) # + as a representation of this operation
print(metta.run('!(* "A" 4)')) # [["AAAA"]]
```

Apparently, there is a textual representation of grounded atoms, from which atoms themselves are built by the parser. But is it possible to introduce such textual representations for custom grounded atoms, so we could refer to them in the textual program code? The answer is yes. The Python and MeTTa API for this is described on the next page.

Source: https://metta-lang.dev/docs/learn/tutorials/python_use/tokenizer.html

Parsing grounded atoms■

Tokenizer■

The MeTTa interpreter operates with the internal representation of programs in the form of atoms. Atoms can be constructed in the course of parsing or directly using the corresponding API. Let us examine what atoms are constructed by the parser. In the following program, we parse the expression(+ 1 S).

```
from hyperon import *
metta = MeTTa()
expr1 = metta.parse_single('(+ 1 S)')
expr2 = E(S('+'), S('1'), S('S'))
print('Expr1: ', expr1)
print('Expr2: ', expr2)
print('Equal: ', expr1 == expr2)
for atom in expr1.get_children():
    print(f'type({atom})={type(atom)}')
```

The result of parsing differs from the expression(+ 1 S)composed of symbolic atoms. Indeed, the atoms constructed from+and1by the parser are grounded atoms - not symbols. At the same time,S('+')is already a symbol atom.

Transformation of the textual representation to grounded atoms is not hard-coded. It is done by the tokenizer on the base of a mapping from tokens in the form of regular expressions to constructors of corresponding grounded atoms.

The initial mapping is provided by thestdlibmodule, but it can be modified later. In the simple case, tokens are just strings. For example, the tokenizer is informed that if+is encountered in the course of parsing, the following atom should be constructed

```
OperationAtom('+', lambda a, b: a + b,
              ['Number', 'Number', 'Number'])
```

Here,['Number', 'Number', 'Number']is a sugared way to defined the type(-> Number Number Numer), which should also be represented as an atom.

Regular expressions are needed for such cases as parsing numbers. For example, integers are constructed on the base of the tokenr"[-+]?\d+", and the constructor needs to get the token itself, so the atom is created by the following function once the token is encountered

```
lambda token: ValueAtom(int(token), 'Number')
```

evaluate_atom■

Once atoms are created, the interpreter doesn't rely on the tokenizer. Instances ofMeTTaclass have methodevaluate_atom, which is the function accepting the atom to interpret.

```
from hyperon import *
metta = MeTTa()
expr1 = metta.parse_single('(+ 1 2)')
print(metta.evaluate_atom(expr1))
expr2 = E(OperationAtom('+', lambda a, b: a + b),
          ValueAtom(1), ValueAtom(2))
print(metta.evaluate_atom(expr2))
```

The example above shows that the parsed expression is interpreted in the same ways as the expression atom constructed directly.MeTTa.runsimply parses the program code expression-by-expression and puts the resulting atoms in the program space or immediately interprets them when!precedes the expression. Note that we could get the operation atom for+(which would be correctly typed) viametta.parse_single('+')

Creating new tokens■

Access to the tokenizer is provided by thetokenizer()method of theMeTTaclass. However, it may not be used directly.MeTTaclass has theregister_tokenmethod, which is intended for registering a new token. It accepts a regular expression and a function, which will be called to construct an atom each time the token is encountered. The constructed atom should not necessarily be a grounded atom, although it is the most typical case.

If the token is a mere string, and creation of different atoms depending on a regular expression is not supposed,register_atomcan be used. It accepts a regular expression and an atom, and callsregister_tokenwith the given token and with the lambda simply returning the given atom.

The following example illustrates creation of an Atomspace and wrapping it into aGroundedAtom

```
from hyperon import *

metta = MeTTa()

# Getting a reference to a native GroundingSpace,
# implemented by the MeTTa core library.
grounding_space = GroundingSpaceRef()
grounding_space.add_atom(E(S("A"), S("B")))
space_atom = G(grounding_space)

# Registering a new custom token based on a regular expression.
# The new token can be used in a MeTTa program.
metta.register_atom("&space", space_atom)
print(metta.run("! (match &space (A $x) $x)"))
```

Parsing and interpretation■

Although the interpreter works with the representation of programs in the form of atoms (as was mentioned above), and expressions should be parsed before being interpreted, the tokenizer can be changed in the course of MeTTa script execution. It is essential for the MeTTa module system (described in more detail in another tutorial).

import!is not only loads a module code into a space. It can also modify the tokenizer with tokens declared in the module. This is the reason why a MeTTa is not first entirely converted to atoms and then interpreted, but parsing and interpretation are intervened. Another approach would be to load all the atoms as symbols and resolve them at runtime, so the interpreter would verify if some symbols are grounded in subsymbolic data. This approach would have its benefits, and it might be chosen in the future versions of MeTTa. However, it would imply that introduction of new groundings to symbols has retrospective effect on the previous code.

We have also encountered creation of new tokens inside MeTTa programs with the use ofbind!showing that token bindings don't have backward effect. The same is definitely true, when we create tokens using Python API:

```
from hyperon import *

# A function to be registered
def dup_str(s, n):
    r = ""
    for i in range(n):
        r += s
    return r

metta = MeTTa()
# Create an atom. "dup-str" is its internal name
dup_str_atom = OperationAtom("dup-str", dup_str)

# Interpreter will call this operation atom provided directly
print(metta.evaluate_atom(E(dup_str_atom, ValueAtom("-hello-"), ValueAtom(3))))
# Let us add a function calling `dup-str`
```

```
metta.run('''
  (= (test-dup-str) (dup-str "a" 2))
''')

# The parser doesn't know it, so dup-str will not be reduced
print(metta.run('''
 ! (dup-str "-hello-" 3)
 ! (test-dup-str)
'''))

# Now the token is registered. New expression will be reduced.
# However, `(= (test-dup-str) (dup-str "a" 2))` was added
# before `dup-str` token was introduced. Thus, it will still
# remain not reduced.
metta.register_atom("dup-str", dup_str_atom)
print(metta.run('''
! (dup-str "-hello-" 3)
! (test-dup-str)
'''))
```

Kwargs for OperationAtom■

Python supports variable number of arguments in functions. Such functions can be wrapped into grounded atoms as well.

```
from hyperon import *
def print_all(*args):
    for a in args:
        print(a)
    return [Atoms.UNIT]
metta = MeTTa()
metta.register_atom("print-all", OperationAtom("print-all", print_all))
metta.run('(print-all "Hello" (+ 40 2) "World")')
```

In cases when the function representing the operation has optional arguments with default values, theKwargskeyword can be used to pass the keyword parameters. For example, let us define a grounded functionfind-poswhich receives two strings and searches for the position of the second string in the first one. Let the default value for the second string be"a". Additionally, this function has the third parameter which specifies whether the search should start from the left or the right, with the default value beingleft=True.

```
from hyperon import *
def find_pos(x:str, y="a", left=True):
    if left:
        return x.find(y)
    pos = x[-1:].find(y)
    return len(x) - 1 - pos if pos >= 0 else pos
metta = MeTTa()
metta.register_atom("find-pos", OperationAtom("find-pos", find_pos))
print(metta.run('''
 ! (find-pos "alpha") ; 0
 ! (find-pos (Kwargs (x "alpha") (left False))) ; 4
 ! (find-pos (Kwargs (x "alpha") (y "c") (left False))) ; -1
'''))
```

Hence, to set argument values using Kwargs, one needs to pass pairs of argument names and values.

Unwrapping Python objects from atoms■

Above, we have introduced a summation operation asOperationAtom('+', lambda a, b: a + b),whereaandbare Python numbers instead of atoms.a + bis also not an atom. Creating of operation atoms getting Python objects is convenient, because it eliminates the necessity to retrieve values from grounded atoms and wrap the result of the operation back to the grounded atom. However, sometimes it is needed to write functions that operate with atoms themselves, and these atoms may

not be grounded atoms wrapping Python objects.

Unwrapping Python values from input atoms and wrapping the result back into a grounded atom is the default behavior ofOperationAtom, which is controlled by the parameterunwrap. Let us consider an example of implementing+while setting this parameter toFalse.

```
def plus(atom1, atom2):
    from hyperon import ValueAtom
    sum = atom1.get_object().value + atom2.get_object().value
    return [ValueAtom(sum, 'Number')]

from hyperon import OperationAtom, MeTTa
plus_atom = OperationAtom("plus", plus,
    ['Number', 'Number', 'Number'], unwrap=False)
metta = MeTTa()
metta.register_atom("plus", plus_atom)
print(metta.run('! (plus 3 5)'))
```

WhenunwrapisFalse, a function should be aware of thehyperonmodule, which can be inconvenient for purely Python functions. Thus, this setting is desirable for functions processing or creating atoms themselves. For example,bind!takes an atom to be bound to a token.parsetakes a string and return an atom of any metatype constructed by parsing this string. One can imagine different custom operations, which accept and return atoms. Say, if a crossover operation in genetic algorithms would be implemented as a grounded operation, it would accept two atoms (typically, expressions), traverse them to find crossover points, and construct a child expression.

Source: https://metta-lang.dev/docs/learn/tutorials/python_use/py_atom.html

Embedding Python objects into MeTTa■

py-atom■

Introducing tokens for grounded atoms allows for both convenient syntax and direct representation of expressions with corresponding grounded atoms in a Space. However, wrapping all functions of rich Python libraries can be not always desirable. There is a way to invoke Python objects such as functions, classes, methods or other statements from MeTTa without additional Python code wrapping these objects into atoms.

py-atomallows obtaining a grounded atom for a Python object imported from a given module or submodule. Let us consider usage ofnumpyas an example, which should be installed. For instance, the absolute value of a number in MeTTa can be calculated by employing theabsolutefunction from thenumpylibrary:

```
! ((py-atom numpy.absolute) -5) ; 5
```

Here,py-atomimportsnumpylibrary and returns an atom associated with thenumpy.absolutefunction.

It is possible to designate types for the grounded atom inpy-atom. For convenience, one can associate the result ofpy-atomwith a token usingbind!:

```
! (import! &self py_ops)
! (bind! abs (py-atom numpy.absolute))
! (+ (abs -5) 10) ; np.int64(15)
```

We specify here that the constructed grounded operation can accept an argument of typeNumberand its result will be ofNumbertype.

When(abs -5)is executed, it triggers a call toabsolute(-5). It can be seen that the results of executing Python objects imported viapy-atomcan then be directly utilized in other MeTTa expressions.

py-atomcan actually execute some Python code, which shouldn't be a statement likex = 42, but should be an expression, which evaluation produces a Python object. In the following example,(py-atom "[1, 2, 3]")produces a Python list, which then passed tonumpy.array.

```
! (bind! np-array (py-atom numpy.array))
! (np-array (py-atom "[1, 2, 3]")) ; array([1, 2, 3])
```

py-atomcan be applied to functions accepting keyword arguments. Constructed grounded atoms will also supportKwargs(mentioned earlier), which allows for passing only the required arguments to the function while skipping arguments with default values. For example, there isnumpy.arangein NumPy, which returns evenly spaced values within a given interval.numpy.arangecan be called with a varying number of positional arguments:

```
! (bind! np-arange (py-atom numpy.arange)) ; ()
! (np-arange 4) ; array([0, 1, 2, 3])
! (np-arange (Kwargs (step 2) (stop 8))) ; array([0, 2, 4, 6])
! (np-arange (Kwargs (start 2) (stop 10) (step 3))) ; array([2, 5, 8])
```

py-dot■

What if we wish to call functions from a submodule, saynumpy.random? Accessing these functions via something like(py-atom numpy.random.randint)will work. However, it would be more efficient to getnumpy.randomitself as a Python object and access other objects in it.py-dotis introduced to carry out this operation.

```
! (bind! np-rnd (py-atom numpy.random))
! ((py-dot np-rnd randint) 25)
```

In this casepy-dotoperates with two arguments: it takes the first argument, which is the grounded atom wrapping a Python object, and then searches for the value of an attribute within that object based on the name provided in the second argument.

This second argument can also contain objects in submodules. In the following example, we wrapnumpyin the grounded atom:

```
! (bind! np (py-atom numpy))
! ((py-dot np abs) -5)
! ((py-dot np random.randint) -25 0)
! ((py-dot np abs) ((py-dot np random.randint) -25 0))
```

Here, when(py-dot np random.randint)is executed, it takesnumpyobject and searches forrandomin it and then forrandintinrandom. The overall result is the grounded operation wrappingnumpy.random.randint, which is then applied to some argument. Similar topy-atom,py-dotalso permits the designation of types for the function, and supportsKwargsfor arguments specification.

Bindingnpto(py-atom numpy)and accessing functions in it via(py-dot np abs)looks not more convenient than just using(py-atom numpy.abs), but is slightly more efficient ifnumpy.absis accessed multiple times.

py-dotworks for any Python object - not only modules:

```
! ((py-dot "Hello World" swapcase)) ; "hELLO wORLD"
```

Notice the additional brackets to callswapcase. The equivalent Python code is"Hello World".swapcase(), which also contains(). One more pair of brackets in MeTTa is needed, becausepy-dotis also a function.

Let us consider another example.

```
! ((py-dot (py-atom "{5: \'f\', 6: \'b\'}") get) 5)
```

Here, a dictionary{5: 'f', 6: 'b'}is created bypy-atom, and then the value corresponding to the key5is retrieved from this dictionary usinggetaccessed viapy-dot.

py-list,py-tuple,py-dict■

While it is possible to create Python lists and dictionaries using code evaluation bypy-atom, it can be desirable to construct these data structures by combining atoms in MeTTa.

In this context, since passing dictionaries, lists or tuples as arguments to functions in Python is very common, such dedicated functions aspy-dict,py-listandpy-tuplewere introduced.

```
! ((py-atom max) (py-list (-5 5 -3 10 8))) ; 10
! ((py-atom numpy.inner)
     (py-list (1 2)) (py-list (3 4))) ; 1 * 3 + 2 * 4 = 11
```

In this example,py-listgenerates three Python lists:[-5, 5, -3, 10, 8],[1,2]and[3,4], which are passed tomaxandnumpy.inner.

Of course, one can usepy-dict,py-list, andpy-tupleindependently - not just as function arguments:

```
! (py-dict (("a" "b") ("b" "c"))) ; creates a dict {"a":"b", "b":"c"}
! (py-tuple (1 5)) ; creates a tuple (1, 5)
! (py-list (1 (2 (3 "3")))) ; creates a nested list [1, [2, [3, '3']]]
```

Source: https://metta-lang.dev/docs/learn/tutorials/metta-motto/metta-motto.html

MeTTa-Motto■

MeTTa-Mottois a library that allows combining the capabilities of LLMs (Large Language Models) and MeTTa. MeTTa-Motto allows calling LLMs from MeTTa scripts, which enables prompt composition and chaining of calls to LLMs in MeTTa based on symbolic knowledge and reasoning.

Simple queries to LLMs■

To make simple queries to an LLM using the MeTTa-Motto library, the following commands can be used:

```
!((anthropic-agent) (user "What is a black hole?"))

!((chat-gpt-agent)
      (user "What is a black hole?"))
```

chat-gpt-agentandanthropic-agentare the agents used to make requests to the ChatGPT and Claude model respectively. Currently, MeTTa-Motto supports the following LLMs:

ChatGPT (by OpenAI)

Claude (by Anthropic)

but more LLMs can be added if needed, and one can also use other LLMs via LangChain integration (seebelow).

Additionally, it is possible to specify the version of ChatGPT or Claude, such as

```
(chat-gpt-agent "gpt-3.5-turbo")
(anthropic-agent "claude-3-opus-20240229")
```

In the example above, the agent may not be specified:!(llm (user "What is a black hole?")). In this case, the default agent (currently,chat-gpt) will be used. The messages which we send to agents as parameters have the form(ROLE "Text of the Message"). There are 3 roles for messages:user,assistantandsystem.

llmis a method defined in MeTTa-Motto, which passes messages to the specified agent and returns their results to MeTTa. For convenience, the keywordllmhas been omitted in scripts starting from MeTTa-Motto version 0.0.7. The previous syntax:

```
!(llm (Agent (chat-gpt))
      (user "What is a black hole?")) ;
```

is now automatically included in the code for the given agent.

We've also included theopen-router-agentto use the OpenRouter API for obtaining responses from LLMs:

```
!(import! &self motto)
!((open-router-agent) (user "Who was the 22nd President of France?"))
!((open-router-agent "openai/gpt-3.5-turbo" True) (user "Who was the 22nd President of France?"))
```

this agent allows to specify the model type, the second call has one more additional parameterTruewhich means that the response should be a stream (Streaming)

As a demonstration, instead of calling LLM agents, we will use theecho-agent. This agent returns the message sent to it, including the role on whose behalf the message was sent

```
!(import! &self motto) ; ()
!((echo-agent)
      (user "The agent will return this text along with a role: user"))
      ; "user The agent will return this text along with a role: user"
```

MeTTa agents■

Also, as an Agent, we can specify the path to a file with a MeTTa script, which typically has a.msa(MeTTa Script Agent) extension. This script can contain any commands (expressions) in MeTTa, and may not necessarily include queries to LLMs in it, but it is supposed to run in a certain context.

For example, let us assume, there is a file namedsome_agent.msacontaining the following code:

```
( = (response)
   (if (== (messages) (user "Hello world."))
       "Hi there"
       "Bye"))
```

(response)is used to indicate that this is the output of the agent. Thesome_agent.msacan be used in another script in the following manner:

```
!(import! &self motto)
!((metta-script-agent "some_agent.msa")
      (user "Hello world.")) ; Hi there
```

or in the following manner:

```
!(import! &self motto)
!((metta-agent "some_agent.msa")
      (user "Hello world.")) ; Hi there
```

For a MeTTa agent, the new atom(= (messages) (user "Hello world."))will be added to the MeTTa space, where the agent will loadsome_agent.msa. This allows(messages)to be used withinsome_agent.msa. Typically,.msaagents are more complex and utilize LLM responses during processing. Ametta-agentexecutes the code insome_agent.msaupon creation, and only runs the(response)function when the__call__method is executed. In contrast, ametta-script-agentexecutes the code insome_agent.msawhenever the__call__method is invoked. This is not the only difference between the two agents. In ametta-agent, there is a field that stores an instance of the MeTTa class, which is used to execute MeTTa code. As a result, all states calculated during the execution of the.msascript will remain in memory (MeTTa space) while the script in which the MeTTa agent was created is running. Formetta-script-agentan instance of the MeTTa class is created each time when the__call__method is executed.

Functional calls■

Suppose we have a function that returns the current weather for a location passed as a parameter to this function. We want to ask about the weather in natural language, e.g. "What is the weather in New York today?", and receive information about the weather in conversational format. For such cases one can describe functions and have the LLM model intelligently select and output a JSON object containing the arguments needed to call one or more functions.

The latest OpenAI and Anthropic models have been trained to both detect when a function should to be called (depending on the input) and to respond with JSON that adheres to the function signature more closely. We can describe such functions in MeTTa-Motto too. For example, for theget_current_weatherfunction, we should first describe it withindocsection and define the function behavior:

```
!(import! &self motto)
(= (doc get_current_weather)
   (Doc
     (description "Get the current weather for the city")
     (parameters
       (location "the city: " ("Tokyo" "New York" "London"))
     )
   )
)
(= (get_current_weather ($arg) $msgs)
   (if (contains-str $arg "Tokyo")
       "The temperature in Tokyo is 75 Fahrenheit"
       (if (contains-str $arg "New York")
           "The temperature in New York is 80 Fahrenheit"
```

```
           (concat-str (concat-str "The temperature in " $arg)
                       " is 70 Fahrenheit")
    )
  )
)
!((echo-agent)
   (user "Get the current weather for the city: London")
   (Function get_current_weather) ; The temperature in London is 70 Fahrenheit.
)
```

The parameters section describes the arguments of the function that should be retrieved from the user's message. The parameters can have the following properties:name,type,descriptionand an enum with possible values.

Thetypeproperty has a specific purpose. It can be provided in the form((: parameter Atom) "Parameter description")indicating that thisparametershould be converted from the Python string to a MeTTa expression before passing to the function.

In our example,concat-str(concatenates two strings) andcontains-str(which checks if the first string contains the second string) are grounded functions defined in MeTTa-Motto.echo-agentis used for the demo purpose, but in real applications it will be any agent that supports functional calls. When a functional call is used withecho-agent, arguments can be extracted from the user's message only if the message includes the function description and the parameter description concatenated with a possible value of the parameter (for example: "the city: " + London). This example is useful only for testing and demonstration purposes.

Streaming■

Some LLMs has API which allows streaming responses back to a client, enabling partial results for specific requests.metta-mottosupports streaming, but currently only for thechat-gpt-agent. By setting the second parameter of thechat-gpt-agenttoTrue, the response ofchat-gpt-agentwill be an atom that contains a Stream object:

```
!((chat-gpt-agent "gpt-3.5-turbo" True) (user "How many planets are in the solar system?"))
```

Scripts■

It is convenient to store lengthy prompts and their templates for LLMs in separate files. For this reason, one can specify the path to such a file as a parameter along with agent. While these files are also MeTTa files and can contain arbitrary computations, they are evaluated in a different context and are recommended to have.mps(MeTTa Prompt Script) extension. Basically, each such file is loaded as a MeTTa script to a space, which should contain expressions reduced to the parameters of thellmmethod. For example,some_template.mpsfile containing:

```
(system ("Answer the user's questions if he asks about art, music, books, for other cases say: I can't answe
```

can be utilized from another.mettafile:

```
!(import! &self motto)
!((chat-gpt-agent) (Script some_template.mps)
  (user "What is the name of Claude Monet's most famous painting?"))
!((chat-gpt-agent) (Script some_template.mps)
  (user "Which city is the capital of the USA?"))
```

The following result will be obtained:

```
"Claude Monet's most famous painting is called "Impression, Sunrise."”
"I can't answer your question."
```

Notice that the parameters specified in the mps-file are combined with the parameters specified directly. This allows separating reusable parts of prompts and utilizing them in different contexts in a composable way. In particular, if one supposes to try different LLMs with the same prompts,Agentshould not be mentioned in the.mpsfile.

Since prompt templates are just spaces treated as parameters tollm, they can be created and filled in directly, but this is rarely needed.

```
!(import! &self motto)
!(bind! &space (new-space))
!(add-atom &space (user "Table"))
!(add-atom &space (user "Window"))
!((echo-agent) (Script &space)) ; "user Table user Window"
```

We are usingecho-agenthere. The result will be a concatenation of all the provided messages.

dialog-agent■

To store dialogue history during interaction with LLMs, we include a special dialogue agent. Let us consider an example. We will use thedialog-agentagent for this purpose. Since the agent will be used multiple times, let us create a binding for it:

```
!(bind! &chat (dialog-agent dialog.msa)))
```

Thedialog-agentagent stores the dialogue history in a special array namedhistory. With each new message, thehistoryis updated. Thehistoryarray can be accessed from the MeTTa script (in our example, fromdialog.msa) via the(history)function.

The filedialog.msacontains the following lines.

```
(= (context)
   (system "You are an AI assistant.
           Please, respond correspondingly."))
(= (response)
   (llm (Messages (context) (history) (messages))))
```

And the dialogue can be executed:

```
!(&chat (user "Hello! My name is John."))
!(&chat (user "What do you know about the Big Bang Theory?"))
!(&chat (user "Do you know my name?")) ; "Yes, you mentioned earlier that your name is John. Is there anythi
```

After the execution of the following line:

```
!((dialog-agent dialog.msa) (user "Hello "))
```

the new atom

```
= (history) (Messages (user "Hello!") (assistant "Greetings, Frodo Baggins! It is a pleasure to see you. How
```

will be added to the MeTTa space, created to executedialog.msa.

dialog-agentinherits frommetta-agent, this means that all states calculated during the execution of the.msascript will remain in memory (MeTTa space) while the script in which the MeTTa agent was created is running.

Retrieval Agent■

Sometimes we may require passing information from various documents as parts of prompts for LLMs. However, these documents may also contain irrelevant data not pertinent to our objectives. In such cases, we can use a retrieval agent. When defining this agent, it is necessary to specify the document location or a path to one document, the chunk length for embedding creation, the desired number of chunks for the agent to return, and a designated path for storing the dataset:

```
!(bind! &retrieval (retrieval-agent "text_for_retrieval.txt" 200 1 "data"))
```

Here,text_for_retrieval.txtis the document that will be searched according to the user's request, the chunks size is equal to 200, and number of the closest chunks to return is 2. This agent computes embeddings for the provided texts and stores them in a dedicated database. In our case, we use ChromaDB, an open-source vector database. When the retrieval agent is invoked for a particular sentence, it first generates embeddings for the sentence and subsequently returns the closest chunks based on cosine distance metrics. For example, the text contains information about a

scientist named John, then we can ask:

```
(&retrieval (user "What is John working on?"))
```

Here is the usage example of a retrieval agent with ChatGPT:

```
!(let $question "What is John working on?"
    ((chat-gpt-agent "gpt-3.5-turbo")
         (Messages
           (system
             ("Taking this information into account, answer the user question"
             (&retrieval (user $question)))
           )
           (user $question)
         )
     )
)
```

The retrieval agent can search not only within a single document but also across an entire folder if specified. In the following example,./data/texts_for_retrievalis the path to the folder where additional information can be searched.

```
!(bind! &retrieval (retrieval-agent "./data/texts_for_retrieval" 200 1 "data"))
!(&retrieval_agent
      (user "Who made significant advancements in the fields of electromagnetism?"))
```

In this case, you can also specify the required document during the agent call using the specialKwargskeyword:

```
!((&retrieval (Kwargs (doc_name "story1.txt")))
      (user  "Who made significant advancements in the fields of electromagnetism?"))
```

LangChain Agents■

As mentioned in thistutorial, by usingpy-atomandpy-dot, you can invoke from MeTTa such Python objects as functions, classes, methods, or other statements. Taking this possibility into account, we have created agents in MeTTa-Motto that allow using LangChain components.LangChainis a Python framework for developing applications powered by large language models (LLMs). LangChain supports many different language models. For example, the following code uses GPT to translate text from English to French:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
messages = [
    ("system", "You are a helpful assistant that translates English to French."),
    ("human", "Translate this sentence from English to French: I love programming."),
]
llm.invoke(messages)
```

The implementation of the code, provided above, in MeTTa-Motto looks like this:

```
!(import! &self motto)
(bind! &chat
  (langchain-agent
    ((py-atom langchain_openai.ChatOpenAI)
     (Kwargs (model "gpt-3.5-turbo") (temperature 0)))
    motto/langchain_agents/langchain_agent.msa))
!(&chat
      (system "You are a helpful assistant that translates English to French.")
      (user "Translate this sentence from English to French: I love programming."))
```

The grounded functionlangchain-agenthas two parameters. The first is a chat model (in this case,langchain_openai.ChatOpenAI), which should be an instance of LangChain "Runnables" with aninvokemethod. The second parameter is the path to the file used to call theinvokemethod for the given chat model.

Filemotto/langchain_agents/langchain_agent.msais a part of MeTTa-Motto library and contains the following lines:

```
(py-dot ((py-dot (langchain-model) invoke) &list) content)
```

The&listis; used to store the entire message history. The grounded atomlangchain-modelis automatically initialized with the chat model passed tolangchain-agent:

```
(= (langchain-model)
   ((py-atom langchain_openai.ChatOpenAI)
      (Kwargs (model "gpt-3.5-turbo") (temperature 0)))
),
```

Thelangchain-agentcan be used in the same situations as thechat-gpt-agentordialog-agentagents.

These examples add not too much for what can be done without LangChain agents. However, if one wants to use LLMs not directly supported by MeTTa-Motto or to use some other components of LangChain together with knowledge representation and symbolic processing capabilities provided by MeTTa, then calling LangChain functions from MeTTa can be very useful. LangChain offers a variety of useful tools. These tools serve as interfaces that an agent, chain, or LLM can use to interact with the world. We can use these tools directly from MeTTa. For example, the following script demonstrates the use of a tool designed to queryarXiv, an open-access archive with 2 million scholarly articles across various scientific fields:

```
!(bind! &arxiv_tool ((py-atom langchain_community.tools.arxiv.tool.ArxivQueryRun)))
!((py-dot &arxiv_tool  invoke) "What's the paper 1605.08386 about?") ;Published: 2011-02-18 Title: Quantum A
```

This example demonstrates how to use the tool individually. The tool can also be used as part of an agent. For this purpose, there islangchain_openai_tools_agent.msain MeTTa-Motto, which utilizeslangchain.agents.AgentExecutorto execute LLM agents with the use of LangChain tools:

```
!(import! &self motto)
!(import! &self motto:langchain_agents:langchain_states)
!(bind! &lst (py-list ()))
!((py-dot &lst append) ((py-atom langchain_google_community.GoogleSearchRun)
    (Kwargs (api_wrapper ((py-atom   langchain_google_community.GoogleSearchAPIWrapper))))))
!(set-langchain-agent-executor  &lst)
!((metta-script-agent "motto/langchain_agents/langchain_openai_tools_agent.msa")(user  "What is the name of
```

The Google search tool is used here to get the answer to the user's question. The script includes the import of thelangchain_states.mettafile, which contains helper functions to create and store prompts, constructlangchain.agents.create_tool_calling_agent, and set parameters forlangchain.agents.AgentExecutor.

Advantages of MeTTa-Motto■

Using MeTTa-Motto, we can process user messages with LLMs to create new knowledge bases or extend existing ones. These knowledge bases can be further processed using MeTTa expressions and then utilized in MeTTa-Motto to solve various tasks. For example, let's consider an agent defined in the file namedsome_agent.msa:

```
(= (response)
    (_eval
        ((chat-gpt-agent)
           (system "Represent natural language statements as expressions in Scheme.
               We should get triples from statements, describing some relations between items.
               Relation of location  should be  represented with 'location' property.
               Relation of  graduated from (or studies)  should be presented as 'educated_at' property.
               For example, the sentence 'New York City is  located at the southern tip of New York State' s
               (\"New York City\" location \"New York State\").
               'Lisbon is in Portugal' should be transformed to (\"Lisbon\" location \"Portugal\").
               Return cities, countries, states and universities in quotes.
               The sentence  'Ann graduated from the University of Oxford' should be transformed to (Ann edu
               The sentence 'John is studying mathematics at MIT' should be transformed to (John educated_at
               For questions about place of study we use function study_location,  for example:
               The sentence 'Is John studying in the USA?' should be transformed to (study_location John \"U
               The sentence 'Did Alan graduate from the University of USA?' should be transformed to (study_
```

```
                The sentence 'Did Mary study in the USA?' should be transformed to (= (study_location Mary \"
                Return result without quotes."
            )
            (messages)
        )
    )
)
```

This agent converts sentences containing location or education-related information into triples, such as(Ann educated_at "Oxford")or("New York City" location "New York State"). If someone asks about the city or country where the education was received, it converts the question into a MeTTa function. For example: Did Mary study in the USA? will be converted to(= (study_location Mary "USA"). Let's define two functions:is-located, which checks if$xis located in$y, andstudy_location, which checks if$xstudied at a place that is located in$y.

```
(= (is-located $x $y)
   (case (match &self ($x location $z) $z)
       (
           (Empty False)
           ($z (if  (== $z  $y) True (is-located $z $y) ))
       )
   )
)

(= (study_location $x $y)
   (case (match &self ($x educated_at $z) $z)
       (
           (Empty False)
           ($z (if  (== $z  $y) True (is-located $z $y) ))
       )
   )
)
```

Then, using thesome_agent.msa, we can add certain relations to the meta space based on the provided facts in natural language, and verify certain facts about the place of study.

```
!(import! &self motto)
(Fact "Harvard is located in Massachusetts state")
(Fact "Massachusetts state is located in United States")
(Fact "Ann graduated from Harvard.")

!(match &self (Fact $fact)
  (let $expr ((metta-script-agent "some_agent.msa") (user $fact))
       (add-atom &self $expr))
 )

!(get-atoms &self)
!((metta-script-agent "some_agent.msa") (user "Did Ann study in the United States?")) ;True
```

This is a straightforward example demonstrating the potential of Metta-Motto to integrate MeTTa functionality with the capabilities of LLMs.

Running Metta-motto in Python■

It is possible to call MeTTa-Motto agents directly from python code:

```
from motto.agents import  MettaAgent
agent = MettaAgent(code = '''
    (= (response) ((echo-agent) (messages)))
    ''')
print(agent('(user "Hello")').content)
```

In this example, an instance ofMettaAgentis created using thecodeargument, which contains a script that defines the agent's behavior. Alternatively, the agent can be initialized with thepathargument instead ofcode. Thepathshould point to a .mps file containing the script to be executed to get a response from the agent. Similarly, we can useDialogAgent, which inherits all methods ofMettaAgentand additionally stores dialogue history. This is thedialog-agentdescribed

above. It is possible to pass additional information, and the method call is executed for eitherMettaAgentorDialogAgent.

For example, let's say the filesome_agent.msacontains a script:

```
(= (respond)
   ((chat-gpt-agent (model_name) (is_stream) True) (Messages (history) (system (system_msg)) (media (media_m
)
(= (response) (respond))
```

here(system_msg)and(media_msg)should be passed to agent during execution.

```
from motto.agents import  MettaAgent
agent = MettaAgent()
v = agent('(user "What is a black hole?")',
          additional_info=[("system_msg", long, 'String'),
                           ("model_name", "gpt-4o", 'String'),
                           ("is_stream", True, 'Bool'), ""]).content
stream = v[0].get_object().content
for chunk in stream:
    print(chunk.choices[0].delta.content)
```

This script sends a request to the ChatGPT modelgpt-4o. Sinceis_streamis set toTrue, the response is returned incrementally in chunks through an event stream. In Python, you can iterate over these events using a for loop.

Source: https://metta-lang.dev/docs/learn/tutorials/backends.html

Backends■

MeTTa has several implementations providing different backends with different computational models under the hood and different features. The original MeTTa implementation is available in therepository. Its interpreter is built on top ofMinimal MeTTa core.

Source: https://metta-lang.dev/docs/learn/tutorials/minimal-metta/minimal-metta.html

Minimal MeTTa■

Minimal MeTTa is a basic set of instructions, which are necessary and sufficient to implement theMeTTa interpreter. The interpreter chains theequality queriesand calls to grounded functions controlling this process depending on the current evaluation context and results including exceptions (errors), empty or non-reducible results. Thus, the main instruction in Minimal MeTTa iseval, which performs one step of evaluation, and which should be applied and chained explicitly. Any atom, which is not a Minimal MeTTa instruction (and not wrapped into such the instruction) will be evaluated to itself.

Minimal MeTTa instructions are available from MeTTa. However, their results can be different, because they will be processed by the MeTTa interpreter further. In order to work with pure Minimal MeTTa, one can use specialpragma. If it is activated, the interpreter will not be used at all, and all expressions in the code will be evaluated without automatic chaining or additional processing. Consider the following example:

```
!(pragma! interpreter bare-minimal)
(= (foo) (bar))
(= (bar) a)
!(foo) ; (foo)
!(eval (foo)) ; (bar)
```

(foo)is not reduced, becausefoois not a Minimal MeTTa instruction, while(eval (foo))is reduced, but only to(bar), which is not automatically reduced further. Let us consider the basic Minimal MeTTa instructions.

Basic operations:chain,eval,unify■

eval■

(eval ARG)searches for the pattern(= ARG $body)in the current space and returns$bodyunified with the corresponding atom in the space if this equality pattern was matched orNotReduciblesymbol otherwise. Consider the following example

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo) (bar))
(= (bar) a)
!(eval (foo)) ; (bar)
!(eval (bar)) ; a
!(baz) ; (baz)
!(eval (baz)) ; NotReducible
```

(eval (foo))is reduced to(bar)(but not toa), because(= (foo) $body)matches against(= (foo) (bar))with$bodybound to(bar), while(eval (bar))is reduced toa. In turn,(eval (baz))returnsNotReduciblesymbol.

It should be noted that in MeTTa the type ofevalis(-> Atom Atom), so its result will not be evaluated further, but such special results asNotReduciblewill still be processed. Withoutbare-minimal, we'll get

```
(= (foo) (bar))
(= (bar) a)
!(baz) ; (baz)
!(eval (foo)) ; (bar)
!(eval (baz)) ; (eval (baz))
```

(eval (baz))in MeTTa remains itself, because it isevalthat returnsNotReducible, so the interpreter keeps the whole expression unreduced. Let us also emphasize that types are processed by the MeTTa interpreter (and could be processed in a different way), while the Minimal MeTTa instructions do not consider them.

It can be confusing at first that(eval (baz))produces different results in MeTTa and Minimal MeTTa, while(baz)remains itself in both cases. The latter has more complex reasons.(baz)is simply not evaluated in Minimal MeTTa, but the MeTTa interpreter tries to evaluate it (secretly callingeval), receivesNotReducible, and returns(baz), because it is(baz)that had no reduction rules (not(eval (baz))as in the previous example).

There are also other conditions and processing steps, which MeTTa interpreter is doing. Consider the following code:

```
(= A AA)
! A ; A
!(eval A) ; AA
```

The MeTTa interpreter appliesevalonly to expressions, which might be function applications. However, if one explicitly executesevalon a pure symbol, the equality query will also be constructed. There could be other choices, when to executeevaland how to deal with its results. Minimal MeTTa allows implementing different versions of the interpreter. Using Minimal MeTTa instructions from MeTTa can also be used to tweak the interpretation process. However, the fact should be taken into account that they will still be called not directly, but from the interpreter code.

We have covered only a part ofevalfunctionality so far. If the argument is an expression, which first element is a grounded function, thenevalcalls this function and returns its result. If the result is a grounded error, it is converted as the following:

ExecError::Runtime->(Error ...)

ExecError::NoReduce->NotReducible

ExecError::IncorrectArgument->NotReducible

Let us note that even thoughbare-minimalturns the MeTTa interpreter off, Minimal MeTTa instructions are still called within its environment, so grounded functions fromstdlibare available. Consider the following examples.

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo) (bar))
(= (bar) a)
!(eval (match &self (= (foo) $x) $x)) ; (bar)
!(eval (match &self (not-exist $x) $x)) ; Empty
!(eval (+ 1 2)) ; 3
!(eval (+ 1 "a")) ; NotReducible
!(eval (pow-math 2 200000000000000)) ; (Error (pow-math 2 200000000000000) power argument is too big, try us
```

matchis a grounded function, which can be executed viaeval. If matching fails, the result isEmptyin Minimal MeTTa.+or any other available grounded function can also be executed viaeval. If the argument of a grounded function is not suitable, it will cause an error, which will be converted toNotReduciblelike in case of(+ 1 "a"). Other errors will be converted to anError-expression.

It should be noted thatevaldoesn't process expressions recursively. If the top-level function is applicable to its arguments in their non-reduced form, then, it will be applied before the arguments reduction. Otherwise, the result will beNotReducible.

Consider the following example.

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo $x) ($x $x))
(= (bar $x $y) ($y $x))
!(eval (bar (foo A) (foo B))) ; ((foo B) (foo A))
```

The result ofevalis((foo B) (foo A))meaning thatbarwas successfully applied. Now, consider the following example

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo OK) OK!!!)
(= (bar) OK)
!(eval (foo (bar))) ; NotReducible
!(eval ((bar))) ; NotReducible
!(eval (+ (+ 1 2) 3)) ; NotReducible
```

In all cases, there is no immediate way to evaluate the top-level function, even in the case of((bar))expression, which also requires peeking one level deeper into it. Thus,evalis a very basic instruction, and a lot of work is performed by the interpreter. Resorting toevalenables low-level control of the evaluation process, but a lot of functionality should be implemented on top of it manually.

unify■

(unify ARG1 ARG2 UNIFIED NOT-UNIFIED)tries to unifyARG1andARG2atoms (any of which can contain variables) and returnsUNIFIED(which typically but not necesserily contains variables fromARG1and/orARG2) if unification is successful and returnsNOT-UNIFIEDotherwise.

The basic usage is as following:

```
!(pragma! interpreter bare-minimal) ; ()
!(unify (a $b) ($a b) ($a $b) fail) ; (a b)
!(unify (a $b) (b a) ok fail) ; fail
```

Wrappingunifyintoevalis not necessary even ifbare-minimalis on, because it is a basic Minimal MeTTa instruction itself.

Since spaces implement custom matching,unifycan accept them as one of its arguments. Thus,unifycan be used instead ofmatch, but with branching depending on the success of unification:

```
(friend Bob Alice)
!(unify &self (friend $who Alice) $who no-friends) ; Bob
!(unify &self (friend Pol $who) $who no-friends) ; no-friends
```

The latter can be achieved withmatchandcasecatchingEmpty, but in a less efficient way.

The code above works identically with and withoutbare-minimal. However,unifyhas%Undefined%return type in MeTTa, so its result will be evaluated further by the interpreter.

chain■

evalperforms only one step of evaluation in Minimal MeTTa, and it is insufficient to perform chained evaluation of custom functions. For example, the following code returns(bar).

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo) (bar))
(= (bar) a)
!(eval (eval (foo))) ; (bar)
```

Why(bar)instead of(eval (bar))ora? Minimal MeTTa executes the outermost instruction. This iseval, which evaluates its argument. Since this argument is a grounded function call (innereval), outerevalcalls innerevaland it returns(bar).

It should be underlined that the result ofevalwill not be evaluated by Minimal MeTTa even if it is a Minimal MeTTa instruction. There is no evaluation loop - just a set of instructions. The topmost instruction is executed in each expression in the program, and itsresultis not evaluated any further. Thus,returnedevalwill be the final result:

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo) (eval (bar)))
(= (bar) a)
!(eval (foo)) ; (eval (bar))
```

A special instruction(chain ARG $VAR RESULT)in Minimal MeTTa is used to execute one instruction (ARG), bind$VAR(which should be a variable) to its result, and substitute$VARinsideRESULTwhich is returned after substitution:

```
!(pragma! interpreter bare-minimal)
!(eval (result is (+ 1 2))) ; NotReducible
```

```
!(chain (eval (+ 1 2)) $x (result is $x)) ; (result is 3)
!(eval (+ 1 (+ 2 3))) ; NotReducible
!(chain (eval (+ 2 3)) $x (eval (+ 1 $x))) ; 6
```

chaincan be used to continue evaluation of a function, which returnseval:

```
!(pragma! interpreter bare-minimal) ; ()
(= (foo) (eval (bar)))
(= (bar) a)
!(chain (eval (foo)) $x $x) ; a
```

This example explicitly shows thatchainnot just returns its last argument (after substitution), but executes it in contrast toeval.

In Minimal MeTTa, arguments are always passed byevalto functions without changes, and subexpressions will never be evaluated. If one wants to construct a (possibly unreducible) expression by evaluating its subexpression, or to first evaluate a subexpression and then the whole expression, then it is necessary to usechainwith this subexpression asARGas shown in the example above. The MeTTa interpreter does this work - it traverses subexpressions and chains their evaluation keeping non-reducible (sub)expressions as is:

```
!(result is (+ 1 (+ 2 3))) ; (result is 6)
```

Let us consider a few more examples in Minimal MeTTa:

```
!(pragma! interpreter bare-minimal)
(= (foo) (bar))
(= (bar) a)
(= (buz $x) ($x $x))
!(chain (eval (foo)) $x (result is $x)) ; (result is (bar))
!(chain (eval (foo)) $x (eval $x)) ; a
!(chain (eval (foo)) $x (eval (buz $x))) ; ((bar) (bar))
!(chain (foo) $x (buz $x)) ; (buz (foo))
```

Apparently, if we perform only one step of evaluation of the function argument, this argument can remain not fully reduced. The only functionality provided bychainis to substitute the result of execution of one atom to another atom, and try to execute the result of substitution (which will be executed if it is a Minimal MeTTa instruction).

chaincan be used in MeTTa instead ofletif a Minimal MeTTa instruction (such asevalorunify) should be executed first. For example,letreturnsEmpty, when the result of unification of its two arguments is empty, whilechainstill returns its last argument with substitutions, which may or may not be further reduced toEmpty:

```
(= (id-eq $x $x) $x)
(= (id-eq $x $y) Empty)
!(let $r (id-eq 1 2) (quote $r)) ; []
!(let $r (eval (id-eq 1 2)) (quote $r)) ; []
!(chain (eval (id-eq 1 2)) $r (quote $r)) ; [(quote Empty)]
!(chain (eval (id-eq 1 2)) $r (it is $r)) ; []
```

quotehasAtomas its argument type, so this argument is not evaluated. That is why(quote Empty)is a possible expression. However, in case ofletthis expression is simply not constructed. In case ofchain,(it is Empty)is constructed but reduced to no result by the interpreter, because the whole tuple doesn't exist if one of its elements doesn't exist.

Calculations in a loop: function/return■

In principle,eval,unifyandchainare enough to implement recursion with arbitrary depth, but it may require morechains than expected:

```
!(pragma! interpreter bare-minimal)
(= (div $x $y $accum)
   (chain (eval (- $x $y)) $r1
     (chain (eval (< $r1 0)) $r2
       (chain (unify $r2 True
         $accum
```

```
         (chain (eval (+ 1 $accum)) $inc
           (chain (eval (div $r1 $y $inc)) $r4 $r4)
         )) $r3 $r3
       )
     )
   )
)
!(chain (eval (div 35 5 0)) $rr $rr) ; 7
```

We will not analyze this program in detail - just notice the number ofchains in it. Minimal MeTTa introduces a pair of instructions -functionandreturn.(function CODE)evaluatesCODE(and its result) until encounters(return RESULT), after which it terminates and returnsRESULT:

```
!(pragma! interpreter bare-minimal)
(= (foo) (eval (bar)))
(= (bar) (eval (baz)))
(= (baz) (return a))
(= (fooo) (function (eval (foo))))

!(function (eval (foo))) ; a
!(eval (fooo)) ; a
```

Let us consider thedivexample. Withfunctionandreturn, there will be twochains less:

```
!(pragma! interpreter bare-minimal)
(= (div $x $y $accum)
   (chain (eval (- $x $y)) $r1
     (chain (eval (< $r1 0)) $r2
       (unify $r2 True
         (return $accum)
         (chain (eval (+ 1 $accum)) $inc
           (eval (div $r1 $y $inc))
         )
       )
     )
   )
)
!(function (eval (div 35 5 0))) ; 7
```

We still need to chain arithmetic calculations, so the profit seems not that big. However, the code looks more natural - we don't need to explicitly put the results ofunifyandevalinsidechainand guess, how many additional steps of evaluation should be done.

functionexpects(return RESULT)expression, and if it encounters something else, which is not an executable Minimal MeTTa instruction, it return an error (;can be removed to see other error messages):

```
!(pragma! interpreter bare-minimal)
(= (foo) (eval (boo)))
(= (boo) (eval (goo)))
(= (goo) G)

; !(function (A)); (Error (A) NoReturn)
; !(function (eval A)) ; (Error NotReducible NoReturn)
!(function (eval (foo))) ; (Error G NoReturn)
```

In the last expression,(eval (foo))is evaluated byfunctionuntil(eval goo)returnsG, which is wrapped intoError.

An interesting feature offunctionis that it is treated by Minimal MeTTa in a special way. Unlike other instructions, it is executed even if it is a result of another instruction (not an argument ofevalorchain). In the example below,evalandchainremain unreduced after being returned by(eval (foo))and(eval (boo))correspondingly, whilefunctionreturned by(eval (goo))is evaluated:

```
!(pragma! interpreter bare-minimal)
(= (foo) (eval (+ 1 2)))
(= (boo) (chain (eval (+ 1 2)) $r $r))
(= (goo) (function (chain (eval (+ 1 2)) $r (return $r))))
! (eval (foo)) ; (eval (+ 1 2))
```

```
! (eval (boo)) ; (chain (eval (+ 1 2)) $r#nn $r#nn)
! (eval (goo)) ; 3
```

collapse-bindandsuperpose-bind■

collapse-bindandsuperpose-binddiffer from collapse and superpose functions fromstdlibin that they store not only atoms, but also bindings (values) of variables.

For example,collapse-bindcan collect results of branches, in which the value of variable$ais different.superpose-bindapplied to the result ofcollapse-bindwill restore the value of this variable in each context. That's whycollapse-bindreturns andsuperpose-bindaccepts not the list of atoms, but the list of pairs(ATOM BINDING), whereBINDINGis a special grounded atom. In the following example, we have a nondeterministic functionxoo.collapse-bindconverts its results to a list of results with bindings.

```
!(pragma! interpreter bare-minimal)
(= (xoo a) xa)
(= (xoo b) xb)

!(chain
  (collapse-bind (eval (xoo $a))) $c
  ($c $a)) ; (((xb { $a <- b }) (xa { $a <- a })) $a)

!(chain
   (collapse-bind (eval (xoo $a))) $c
   (chain
     (superpose-bind $c) $x
     ($x $a))) ; [(xa a), (xb b)]
```

chainis used here to extract bindings of$ainto the external context (i.e., to construct($x $a)).

collapse-bindis similar tochainin that it executes Minimal MeTTa instructions to collect the result.

cons-atomanddecons-atom■

cons-atomconstructs an expression given its head and tail, whiledecons-atomseparates an expression into its head and tail, and returns them as a pair.decons-atomexpects a non-empty expression.

```
!(cons-atom a (b c)) ; (a b c)
!(decons-atom (a b c)) ; (a (b c))
!(decons-atom (a)) ; (a ())
!(decons-atom ()) ; (Error (decons-atom ()) expected: (decons-atom (: <expr> Expression)), found: (decons-at
```

Source: https://metta-lang.dev/docs/learn/tutorials/func_prog/intro.html

Basics of Functional Pragramming in MeTTa■

Coming Soon■

Source: https://metta-lang.dev/docs/learn/tutorials/types_adv/intro.html

Coming Soon■