# Table of Contents

1. Introduction ........................................................................................................... 4
   1. Beyond Word Processing ........................................................................... 4
   2. Why Create a Language? ........................................................................... 4
   3. Language Purpose ................................................................................. 5
   4. Language Implementation ...................................................................... 6
   5. Beginning MSL ..................................................................................... 7
   6. MSL Viewer ......................................................................................... 8
   7. MSL Engine ....................................................................................... 9

2. Language Reference .......................................................................................... 10
   8. Design Principles .............................................................................. 10
   9. Syntax Diagrams ............................................................................. 10
### 10. MSL Expressions

11. Prelude

12. Machine

13. World

14. Stream

15. View

16. Canon

17. Atom

18. Key

19. Selector

20. Value

21. Transform

22. Format

23. Hash

24. Comment

### 3. MSL in Practice

25. Getting and Setting Atom Values

26. Sample Canon and Text

27. Text Edits

28. Metadata Selectors

29. Attribution Using Metadata Selectors

30. RegEx Selectors

31. Changing Canonical Values

32. Viewing Atoms Through Alternate Canon

### 4. MSL Evaluation

33. The Atom's General Form

34. Evaluation of Expressions

35. Evaluation of Parameters

36. Value Retrieval

37. Currying

38. Two Forms of MSL

39. 1-Form

40. 2-Form

41. Reaching

42. Moving and Sharing Atoms

43. Rewinding

44. MSL Text

45. Sources of MSL
5. Data Types ..........................................................................................................................55
   46. About Data Types ........................................................................................................55
   47. Recommended Data Types ..........................................................................................56
   48. Date ................................................................................................................................57
   49. Time ................................................................................................................................58
   50. Number ........................................................................................................................58
   51. Image ..............................................................................................................................58
   52. Audio ...............................................................................................................................59
   53. Video ................................................................................................................................59
   54. Code ...................................................................................................................................59
   55. App ..................................................................................................................................60
   56. Binary ..............................................................................................................................60
   57. Custom Types ................................................................................................................60

6. Glossary ...............................................................................................................................62
Introduction

*MSL provides high level operations for creating and destroying atoms, building and applying metadata, recording and sharing streams, and accessing and editing these elements.*

-- Mimix Whitepaper

1. Beyond Word Processing

A word processor knows about words, sentences, paragraphs, pages, and documents. It does not, however, understand the factual information contained in what you read and write. Mimix represents the evolution of the word processor in that it provides a means for text processing software to interpret, display, analyze, compare, correct, and transform the factual data inside our documents.

The [Mimix whitepaper](#) describes in detail the historical precedents for a different kind of reading and writing system. It also lays out several scenarios in which such a system can radically transform how we interact with written documents. In short, this demands several new capabilities from our software:

1. Fact checking and correction of factual errors in written text.
2. Options for viewing facts in multiple formats.
3. Access to source materials for all references.
4. A "rewind" facility to return to any previous version of text.
5. A portable version that can be shared with all references intact.

Mimix is a new way of looking at research and writing. To implement it requires new tools and a new vocabulary. When this document first mentions a word that has a special meaning in Mimix, it will appear in **bold**. All of these terms any many others which are used throughout this document are defined in the Glossary at the end of the paper.

2. Why Create a Language?

The Mimix ecosystem is built around a new programming language called MSL, Mimix Stream Language. This document describes that language in detail. A new programming language was desirable for several reasons:

1. It allows modeling an editing session as a program which can be audited and replayed.
2. It is easier to read and write than the underlying system code.
3. It provides built-in protections that the underlying system cannot.
4. It serves as a common specification between all the parts of the system.
5. It enables anyone to write software that works with Mimix.
Recording a series of text documents and edits in the form of a program allows thinking about an editing session in programming terms. We can view words and paragraphs as variables and values. Text references such as a document we’re quoting can be viewed as dependencies. This lets us extract any part of the program (the editing session), find its text dependencies, and work with them using any software that understands MSL.

The MSL language is designed to be easier to write than the actual system code which is needed to perform the same functions. You can think of MSL as a shortcut method for describing reading and writing as they take place in linear time.

In most cases, Mimix users will never see MSL, but it’s designed to easy to read by human beings regardless. This makes testing, debugging, and extending the system easier.

The MSL language provides built-in protections for data that the underlying system would not. This is used to enforce protections for factually correct information, which we call canon. MSL also uses a built-in set of rules to prevent overwriting important data. Most importantly, it offers a granular rewind capability in which any piece of data can be “rewound” to any of its previous values. This same ability allows extracting any value in the system for use in a different context with all of its dependencies and sources intact. These features are unprecedented among programming languages and are defining aspects of MSL.

The MSL language is communicated between a viewer, where text editing takes place, and an MSL engine which records the expressions in a hybrid database which offers access to both past and present values. This allows the development of numerous compatible engines and viewers, all of which rely on the present MSL specification.

Anyone can write tools that work with MSL and this is encouraged by specifying a language definition that is simple, complete, and coherent. Developers can easily visualize the MSL they will need to input or output in order to achieve their desired results.

3. Language Purpose

Mimix Stream Language is a domain-specific language designed for recording and playing back text editing sessions. A list of MSL expressions is an indelible record of source materials, original writing, editing, and notes in the order in which they were introduced. MSL is the filament in the Mimix light bulb. It is the most important part of the system.

MSL is designed to be good at two kinds of jobs. First, it can be used to record the actions of a Mimix user as he or she reads, writes, edits, and annotates text. It provides good traceability in that the source of values and their changes over time can be readily ascertained. It enables a rewind facility so that previous versions of a stream can be viewed, extracted, or rolled back.
Second, it can be used as a playback language to recreate the same view or a different view of the recording. This allows recreating an entire view or stream from the MSL text at any point, or creating a new type of presentation or transformation of the materials a user has viewed, written, edited, or annotated (or any subset of these) up to a certain point. In other words, MSL can be used to play back any part of the stream with a different view than the original.

4. Language Implementation

MSL is a stream recording language which captures a user’s reading, annotations, writing, and editing into a stream of views as they take place in the viewer. User behaviors are recorded as a series of MSL expressions. These can be used individually, in parts, or in a sequence to recreate any part of the stream using the same view, a different view, or a different viewer.

MSL expressions are executed by a state machine. The state is stored in a hashtable in the MSL engine. MSL expressions passed to the engine can get or set values in the hashtable. The current value of the hashtable represents the full system state after the last MSL expression executed. The state machine serves as a convenient reference to the values contained in the stream, which are also preserved separately.

MSL is an imperative language in which every expression recalls or updates the system state in a sequential order. The state is defined as the value of the hashtable up to and including the execution of the last MSL expression. Although MSL is not procedural, it does allow the application of selectors and transforms, functions which are applied to an atom, the basic unit of data in the language. Selectors and transforms permit viewers and engines to define functions outside of MSL which are then acted upon by MSL values.

MSL is lexically scoped with atoms having values defined in the atoms themselves, their metadata, in their applied canon, or in one of these belonging to a previous containing view, stream, world, or machine.

MSL is a functional language in which every program contains all its literal values inline and all functions can be resolved through variable substitution without reliance on an independent state. MSL is also functional in the style of the Lambda Calculus in that every MSL expression is permitted only one parameter, its value. MSL supports currying values in that the result of single-argument functions are passed upwards to other functions in order to provide a final value.

The hybrid database in a Mimix system serves only as a convenience for quick resolution of an atom’s value. The actual values themselves are always recorded in MSL text and can be resolved by simply examining the MSL text backwards without reference to the hybrid database.
MSL is not Turing complete and essentially consists of only elaborate variable getters and setters. No MSL expressions perform looping, testing, branching, or recursion. MSL is an interpreted language with the system state stored in a nested series of hashtables backed up by a stream of MSL text. MSL expressions which set an atom's value change the system state by updating values in the hashtables and recording a new entry in the text. MSL expressions which only return a value do not change the system state and simply recall the current value from the hashtables or the text while also recording the expression itself in MSL text.

5. Beginning MSL

The simplest way to provide a permanent record of what you read and write is to write down everything you do in chronological order. In fact, this is how MSL works.

If you were planning a trip to Hawaii, you might read two books and make some notes:

1. (@fodor-book Fodor’s 2020 Guide to Honolulu and Oahu)
2. (@lonely-planet Lonely Planet Hawaii)
3. (@hawaii-notes International Marketplace. Pearl Harbor. Shave ice.)

These are MSL expressions. Whenever a valid MSL expression appears in this document, we'll use a code font in blue. When several expressions are given as examples of the same concept, we'll number that list. When MSL concepts are referenced without a full, valid syntax, we'll use a code font in black.

The (@) notation here refers to an atom, the basic unit of storage in MSL. Every atom includes a key which identifies it. It's customary (but not required) to write the key up against the @ sign.

(@lonely-planet Lonely Planet Hawaii)

This atom contains the title for a book. In order to be useful as a reference going forward, we would need to record the entire book. We could simply dump the book's contents into the atom. Or we could give individual paragraphs, people, places, etc. their own atoms that sit inside the book atom.

(@lonely-planet Lonely Planet (@Hawaii)
 (@intro The goddess (@Pele) …)
 (@chapter 1 (@p1 When the ancient (@Hawaii Hawaiians)... (@p2 ...))
 (@chapter 2 ...)))

Text which is read, written, or edited by a Mimix user is recorded sequentially in a series of atoms, and this will be shown through many examples.
Atoms can be grouped into **views, streams, worlds, and machines**. Atoms can also be defined in canon, giving them a type of data protection. Views, streams, worlds, machines, and canon are atoms that reside in their own **namespaces**. These basic concepts enable all five of the abilities we need to go beyond word processing.

### 6. MSL Viewer

When a user browses, creates, or edits written materials in Mimix, an MSL viewer records those actions by writing MSL expressions into a stream and passing them incrementally to the engine. As the user works, the engine records the stream as MSL text files.

The viewer creates a user interface from MSL expressions. It may offer a full word processing environment with facilities like text editing, multiple document views, importing, exporting, encryption, stream management, and verification or substitution of factual data. Alternatively, it may offer a more limited set of interfaces. A specialized viewer could be developed which only translated language text, for example. Or a different viewer might consume a specific type of documents, like legal documents, and output them as MSL for an engine.
7. MSL Engine

An MSL engine accepts MSL expressions and gets or sets the corresponding values from the internal hashtable and the existing recorded streams which together comprise the system state. It has no other externally accessible functions. The MSL engine only accepts MSL as input and only provides MSL as output. Any part of the current or previous system state, from the value of a single atom to the full set of all values from the machine, can be requested by an MSL expression and returned as an MSL expression. The “requesting” expression has keys without values and the “response” expression has the exact same form with values provided after each key name.

The hashtable is strictly a convenience. It provides a faster way to find the current value of any atom. The MSL text, however, contains everything known about the system so the hashtable isn’t technically required. The engine uses the MSL text to find previous values which might no longer be in the hashtable when those are requested by the viewer.

Decoupling the engine from the viewer allows any number of MSL viewers to be developed with unique capabilities and user interfaces. Several viewers can connect to the same engine at once and that engine can be running on the local machine, on a network, or in the cloud.
8. Design Principles

1. Readable. Code should be readable and understandable above all other goals.
2. Tolerant. Code should be tolerant. There should be few errors and many fallbacks.
3. Portable. Code should be movable between other parts of code, machines, etc.
4. Extensible. Code should be easy to extend with new functions.
5. Succinct. Code should be as small as possible.
6. Consistent. Code should use similar syntax whenever possible.
7. Implementable. Code should be easy to implement in the underlying system.

When conflicts arose, compromises were made in this order of priority.

9. Syntax Diagrams

Generalized MSL syntax is described in this document using railroad diagrams and Extended Backus-Naur Form (EBNF) notation, which is shown in the code font in red.

```
railroad ::= 'literal' required [range] ( choice | another-choice ) optional?
repeated+ optional-repeated*
```
10. MSL Expressions

```
MSL ::= ( prelude | machine | world | stream | view | canon | atom | selector |
       value | transform | comment )*  
```

An MSL expression is either a **prelude**, **machine**, **world**, **stream**, **view**, **canon**, **atom**, **selector**, **value**, **transform**, or **comment**. A valid MSL program is any series of MSL expressions in order which are themselves valid. An MSL expression whose hash value does not equate to the hash of the previous expression is also invalid and cannot be interpreted.
An MSL expression must always return a value. Any MSL expression which results in no value (a Lisp NIL) cannot be parsed and also results in all its containing expressions resulting in NIL and therefore stopping interpretation. By definition, an MSL expression cannot set or get a null value, as empty results are not meaningful in the context of parsing narrative text. No MSL expression can set a value to null and no MSL expression can result in the return of a NIL or null value.

MSL is built from **Lisp style s-expressions**. MSL differs from Lisp in the way it processes functions and arguments, so Lisp expressions cannot be directly included inside MSL nor vice-versa. MSL code can only contain other MSL. The beginning of MSL expressions inside a text file can be indicated by a prelude but this is not required to interpret a file or fragment.

1. Prelude

prelude ::= '(' 'msl' key ( selector | value | transform )* hash? comment? ')' 

An optional prelude can be used to denote the beginning of an MSL program and to provide information to the engine and viewer about the program name, its version, or environment. A prelude can also be used to load a program from a file or web address using a transform. A transform inside a prelude expression results in the transform’s value being loaded after the prelude, rather than inside it.

1. (MSL several-worlds)
2. (MSL several-worlds Work and Book Project Worlds :version 1.0)
3. (MSL hollywood-stars :author Mr. Peanut Butter :date 6/17/2015)
4. (MSL chemical-papers Chem Library [file:///d:/myfiles/chemical-papers.msl])
A Mimix **machine** (or just a **mimix**) can contain a list of canon, worlds, streams, views, and atoms as well as metadata about the machine itself. This represents the entire physical or virtual machine the user is running and all of the canon, worlds, streams, views, atoms, and metadata it knows about.

```
(m david
  (w books
    (s nuclear-book
      (c periodic-table)
      (c your-friend-the-atom)
    )
    (v draft
      (@intro (@p1) (@p2))
      (@chapter-1 (@p3) (@p4)))))
)
This MSL describes a machine (m david) with a world (w books) where two books are being authored. Each book is located inside a stream, (s nuclear-book) and (s cloud-control). Each stream begins with canon like (c periodic-table) that provides fallback values to atoms that follow it. Each book’s text sits inside a view like (v draft) that consists of several atoms such as (@intro) and (@chapter-1). A reference to an atom by its key alone like (@p1) retrieves the last value of that atom.
A world (or workspace) can contain a list of canon, streams, views, atoms, and metadata about the world itself. This represents an area of work or study separate from others. A person might have a world of work-related streams and a separate world of personal streams.

```
(w mimix-work
  (s whitepaper)
  (s blog-post-1)
  (s blog-post-2))
```
14. Stream

\[
\text{stream ::= '('(' s' key (machine | world | stream | view | canon | atom | selector | value | transform )* hash? comment? ')')}
\]

The entire session of consulting canon, creating atoms, and editing views is saved inside a stream.
This example stream contains canonical product information (c clairol-product) and a review (v haircolor-review) of one of the products in the canon.

(s beauty
  (c clairol-product
    (v miss-clairol
      (@miss-clairol-27 Lightly Blonde :no 27 :family blonde :box
       [file:///d:/box/clairol%20images/miss-clairol/27.jpg]) :name Miss Clairol)))
  (v haircolor-review
    (@p1 (@miss-clairol-27 Blonder than Bond) is what they’ll say.)
    (@p2 Does she or doesn’t she? (f italic) Only her hairdresser knows for sure. In this case, she does. With (@miss-clairol-27 :full-name (v miss-clairol :name)
      (@miss-clairol-27 :no)) (f bold) to be exact, or “(@miss-clairol-27),” as the box puts it.))
)

In the viewer, (@p2) would display like this:

*Does she or doesn’t she?* Only her hairdresser knows for sure. In this case, she does. With Miss Clairol 27 to be exact, or “Lightly Blonde,” as the box puts it.

This example illustrates several interesting features of MSL:

1. Using a stream to apply canon to views that follow it. (s beauty)
2. Using canon to provide a fallback value for an atom. (c clairol-product)
3. Using a view to group atoms together. (v haircolor-review)
4. Providing a value to an atom to use locally. (@miss-clairol-27 Blonder than Bond)
5. Using a transform to format a value or atom. (f italic)
6. Constructing a value from other atoms’ values. (@miss-clairol-27 :full-name)
7. Referencing a view’s metadata inside an atom (reaching). (v miss-clairol :name)
8. Using a selector to retrieve an atom’s metadata. (@miss-clairol-27 :no)
9. Using a key alone to retrieve an atom’s value from canon. (@miss-clairol-27)
10. Using the canon’s reset value instead of the previous set value. (@miss-clairol-27)
15. View

```
view ::= ('v' key ( machine | world | stream | view | canon | atom | selector |
value | transform )* hash? comment? ')
```

We can use a view to bundle together atoms for display. If you think of an atom as representing a word, a name, a sentence, or a paragraph, then a view is a list of paragraphs you want to see in a certain order, like a document or a note.

This is a view `(v v1)` with only one paragraph atom inside it.

```
(v v1 (@p1))
```

The viewer builds this MSL automatically and adds it to the stream along with the paragraph atoms as they are created so that they remain visible on-screen.
Writing a third paragraph without making other changes would result in another view added to the stream. The same view number, v1, is used each time, indicating that this is a revised definition of that view.

\[
(v \ v1 \ (@p1) \ (@p2))
\]

If an entire paragraph needs to go away, it can be removed with the viewer. Another view expression is recorded without that paragraph atom.

\[
(v \ v1 \ (@p1) \ (@p3))
\]

Notice that @p2 has not gone away. Neither has the previous view which included it. The engine keeps a full version of all the MSL instructions in sequence. This contains everything needed to rewind or re-produce any previous version of the system state.
16. Canon

Atoms can be wrapped in a canon expression to provide a layer of protection against future changes. Here, periodic table atoms are set inside a canon expression (c periodic-table):

```
(@H hydrogen :no 1 :symbol H :weight 1.008)
(@Li lithium :no 3 :symbol Li :weight 6.94)
(@Na sodium :no 11 :symbol Na :weight 22.990))
```

This canon can be applied to atoms which use these keys by placing the (c) expression in a view before the paragraphs that use it. Thereafter, the canonical data is recalled and used in place of the (@) references when the paragraph is displayed.
The `(@sample-sentence) atom would render like this in the viewer:

Hydrogen is the first element (atomic number 1) in the periodic table and the lightest, with an atomic weight of 1.008.

Basic sentence formatting, such as capitalizing the H in hydrogen or providing proper spacing between the period at the end of the sentence and the next sentence that might appear in the narrative can be performed by the viewer.

Values provided for atoms which are protected by canon only overwrite the specific instance in which the value is provided, not any other references to the atom going forward. Future references to the atom’s value reset to the canonical values.

1. `(@p1) `⇒ The atomic weight of Lithium is about 7.
2. `(@p2) `⇒ The exact atomic weight of Lithium is 6.94.

Here, the last invocation of `(@Li:weight) recalls the canonical value, not the value supplied in the previous paragraph.
17. Atom

atom ::= '(' '@' key (machine | world | stream | view | atom | selector | value | transform )* hash? comment? ')' 

The fundamental unit of storage is the atom, written (@). The following is an MSL expression that sets a value of “Walt Disney” for an atom with the key “WALT.” When working with atoms in the @ namespace, you can put the key right up against the @ sign, and this is the most common syntax. To set a value, you put the value after the key.

(@WALT Walt Disney) ⇒ Walt Disney

An MSL expression like this which contains a value returns the value it is given. In a typical viewer, The name “Walt Disney” would appear in paragraph text in place of this atom. It could also be used to identify this atom in other areas of the user interface.
18. Key

Every atom is identified by a key which provides a way to refer to the atom whether it is located inside a view, stream, world, or machine. The key must be unique within the entire Mimix machine. In other words, the atom (@WALT) refers to the same atom whether it appears in a view, canon, stream, or world. Two separate Mimix machines may have different values for (@WALT) but these must be coalesced into a single value by the viewer before any atom referring to (@WALT) from the first machine could be incorporated into the second.

The atom (@WALT) can be moved between machines, worlds, streams, views, or canon without changing its value. Every reference to that key whether direct (in which the full key sequence is stated) or implied (in which it is not) is a reference to the same atom, including all of its metadata and values.

Keys begin with a group of upper or lowercase letters from A-Z. Additional groups may be separated by dashes. The second and subsequent groups may also contain numbers. No other characters are permitted in keys. Keys are not case-sensitive but are set and recalled using the case provided in the expression. A space must appear between the key and the next syntax element.

Key ::= [a-zA-Z]+ ( '-'? [0-9a-zA-Z]+ )*
Valid Keys

1. (@Julie-Andrews Julie Andrews) ⇒ Julie Andrews
2. (@mary-poppins (@Julie-Andrews)) ⇒ Julie Andrews
3. (@MARY-POPPINS) ⇒ Mary Poppins
4. (@MARYPOPPINS “Mary Poppins”) ⇒ Mary Poppins
5. (@MaryPoppins) ⇒ Mary Poppins
6. (@MOVIE2 (@Mary-Poppins)) ⇒ Julie Andrews
7. (@movie2 (@MaryPoppins)) ⇒ Mary Poppins
8. (@movie2) ⇒ Mary Poppins
9. (@Julie-Andrews:birthday) ⇒ 1935-10-1
10. (@Julie-Andrews:movie2 (@movie2)) ⇒ Mary Poppins
11. (@JULIE-ANDREWS-MOVIE2-TITLE (@Julie-Andrews :movie2)) ⇒ Mary Poppins

Invalid Keys

1. (@2MOVIES) ⇒ Starts with a number.
2. (@-MOVIES) ⇒ Starts with a dash.
3. (@MOVIES-) ⇒ Ends with a dash.
4. (@MOVIES/HITS) ⇒ Invalid character: slash
5. (@MOVIES.HITS) ⇒ Invalid character: dot
6. (@MOVIES(2020)) ⇒ Invalid characters: parentheses
7. (@MOVIES[NEW]) ⇒ Invalid characters: brackets
8. (@MOVIES@HITS) ⇒ Invalid character: @
9. (@MOVIES(@HITS)) ⇒ Needs space before a new atom (@HITS).
10. (@MOVIES :HITS:NEW) ⇒ Needs a space before a metadata key :NEW.
11. (@MOVIES Top 100 Films) ⇒ Canon namespace needs a space before key.

19. Selector

```
selector ::= ':' metadata-key | '/' regexp '/' regexp-env | data-type
```

A selector offers access inside an atom, such as selecting its metadata or the results of a regular expression. A selector can also indicate the type of data a value represents.
20. Value

Text literals can be provided as values using a flexible syntax. A single quote can be used as in Lisp to combine multiple words into a single value, but the net effect is the same as writing them unquoted. Values are stored using the capitalization provided.

1. (@WALT Walt Disney) ⇒ Walt Disney
2. (@WALT 'Walt Disney) ⇒ Walt Disney
3. (@WALT 'Walt 'Disney) ⇒ Walt Disney
4. (@WALT 'Walt 'Disney') ⇒ Walt Disney
5. (@WALT 'Walt Disney') ⇒ Walt Disney
6. (@WALT "Walt Disney") ⇒ Walt Disney
7. (@WALT "Walt" "Disney") ⇒ Walt Disney
8. (@WALT “Walt Disney”) ⇒ Walt Disney
9. (@p1 (@WALT)’s Birthday) ⇒ Walt Disney’s Birthday
21. Transform

```plaintext
transform ::= ('[' (filespec | url) ']') | format
```

### URL & File Transforms

A transform takes a value from an atom, a selector, or literal text and either alters it (substituting a new value in its place) or modifies its appearance in the viewer. The [filespec] and [url] transforms return a result and so they change the atom’s value. Together, these are known as **bracketed transforms**. A bracketed transform which does not return a result because the file or URL reference cannot be resolved is an error, meaning the containing expression cannot be interpreted in the present context.

Transforms are rendered in the viewer rather than in the engine. They are recorded along with the rest of the MSL expression passed to the engine and returned along with results but they are only utilized by viewers.

The URL transform is a bracketed transform, written `[url]`, which returns the value found at the URL. A transform is equivalent to writing what’s found at the URL as the value of the atom.

```plaintext
(v declaration [https://archive.org/stream/unitedstatesdecl00001gut/when12.txt]) ⇒
(v declaration (@text Welcome to the Project Gutenberg version of... :url https://archive...))
```

A bracketed transform must return a value that is valid in the context where it’s written. The URL transform above is in the context of the value of a view, so it must return something that can be the value of a view. It returns the value it got from the URL and stores the transform source value internally to say where it came from.
The file transform has the same syntax.

\[
(s\ \text{school-paper}\ (c\ \text{textbooks}\ \text{[file:///user/textbooks/Biology\ II.msl]})) \Rightarrow \\
(s\ \text{school-paper}\ (c\ \text{textbooks}\ (@\text{bio-today}\ (@p1\ ...))\ (@\text{enviro-sci}\ (@p1\ ...))))
\]

Here the file transform is given in the context of a canon value, so it needs to return something that could go in a canon.

In all cases, bracketed transforms are specified in URL format with the URI filespec standard denoting files on local or network drives. Viewer applications will typically allowing browsing or bulk processing of these local drives and are responsible for converting local paths to URI filespecs before including them inside a transform. Viewers should also convert URI filespecs back to local paths in order to compare or update those sources from files, directories, or drives on the local machine. In this way, each viewer is only responsible for processing one kind of transformation: local paths on the viewer’s same platform to URI filespecs and back.

Because transforms always come after a value, it’s easy to identify all of them which exist in a stream and to resolve them or remove them as desired.

A format transform doesn’t return a value and doesn’t change the atom’s value.
22. Format

A viewer can use a format transform to alter the appearance of an expression on screen. These can be as simple as text formatting or as complex as a music player or a slideshow viewer.

A format expression can be considered an arbitrary function call that runs in the viewer. The key specifies which function should be run and can also be used to group functions together into formatting chains with each previous function’s result curried to the next function. The atom’s value and metadata act as parameters to the first function call.

1. (@LAND Edwin Land :brand Polaroid (f bold)) ⇒ Edwin Land
2. (@LAND :brand (f upper)) ⇒ POLAROID
3. (@LAND :brand (f wordmark (f upper) (f bold) (f italic)))) ⇒ POLAROID
4. (c tm (f sx-70-tm (f font Pill Gothic :size 12 pt :weight 500))) ⇒ POLAROID SX-70
5. (@LAND :brand (f workmark)) (@SX-70 SX-70 (f sx-70-tm)) ⇒ POLAROID SX-70
6. (@wonder-if-take-you-home Wonder If I Take You Home :artist (@lisa–lisa) :year (@1985) (f song-player)) ⇒ song-player interface
7. (c disney-documents (f biblio)) ⇒ automated bibliography
8. (c disney-documents (f timeline :tick-marks 5)) ⇒ timeline of canon
9. (v my-thesis (f timeline images)) ⇒ timeline of images in a view
10. (@my-program [filespec] (f js)) ⇒ run a JS file in the viewer

Format functions are atoms with one limitation: they cannot contain other atoms or transforms, only other formats, selectors, or values. Otherwise, the storage and recall of formats is identical to other atoms. This allows formats to be defined on-the-fly, stored in canon, views, streams, worlds, and machines, and recalled and used in short form like other atoms.
23. Hash

```plaintext
hash ::= '#' sha-256

sha-256 ::= [https://en.wikipedia.org/wiki/SHA-2]
```

An atom-form expression such as (@), (c), (v), (s), (w), or (m) can include an optional hash value. The value is the result of hashing the last successfully evaluated expression using the SHA-256 algorithm. It is important to note that the hash value is not the value of hashing the current expression but rather the value of hashing the expression which came before it. In this way, the hash value of the previous expression can be included in this expression’s hash to form an irrevocable chain. An engine could use an atom’s hash to locate it uniquely or to recall its contents without finding where the atom “lives” in the existing MSL stream.

If three expressions are evaluated in sequence and the second expression contains the hash of the first and the third expression contains the hash of the second, a chain of edits can be established with each expression’s precedents being established with absolute certainty, as with a blockchain.

In the following examples, pseudonyms will be used instead of actual hash values to better delineate the chaining function:

1. (@p1 It was a dark and stormy night. #hash-of-previous-expression)
2. (@p2 The wind blew fiercely through the woods. #hash-of-@p1)
3. (@p1 It was a dark and windy night. #hash-of-@p2)
4. (v ch1 (@p1) (@p2 The thick scent of jasmine loomed in the air.) #hash-of-last-value-of-@p1)

Hash values can be used in a viewer or engine to facilitate indexing of atoms. Hash values can also be exported by a viewer to an external program which tracks them independently of the content to provide a blockchain verification service.

If every atom-form expression is hashed in this way, recording those hashes externally prevents surreptitious editing of MSL text files with the intent to misrepresent the chain of edits. A bad actor attempting to assert that different text was supplied to (@p1) in step #3, for example, could be identified because the hash of that step #3 would not be the same as the one recorded in step #4. If every expression were hashed and recorded externally, it would be simple to detect where the alteration was attempted.
24. Comment

A comment can be added at the end of an MSL expression. The comment ends with the expression's closing parenthesis. Comments are not stored nor retrieved with atom values but are preserved in the MSL text in the hybrid database. Because comments are removed before processing the remainder of the MSL expression, they are not included in hash values.

```ml
(w harvard-world Med School
   //english :lang en-us) //school stuff)
```
MSL in Practice

25. Getting and Setting Atom Values

To retrieve a value which has been set previously in the atom, in canon, or in a view, stream, world, or machine, we only have to write the key:

\[(@WALT) \Rightarrow Walt Disney\]

To change an atom’s value going forward, we write another expression with the same key and a new value:

\[(@EPCOT EPCOT) \Rightarrow EPCOT\]
\[(@EFCOT) \Rightarrow EPCOT\]
\[(@EPCOT Epcot Center) \Rightarrow Epcot Center\]
\[(@EPCOT) \Rightarrow Epcot Center\]

The first time \((@EPCOT)\) gets a value, it is the word “EPCOT.” Recalling \((@EPCOT)\) by key alone returns this same value. The next line changes the value of \((@EPCOT)\) to “Epcot Center.” The next time the atom is recalled by key alone we get this most recent value.

Values are recorded in the @ expressions in MSL itself. The rationale behind this is that text values are not canonical and might change throughout a document. In the same paper, Disney, Walt, Walt Disney, Mr. Disney, and even words like “he” or “him” can all refer to the same atom with different text values appearing inline. The MSL records the inline value at the moment in the stream where it is used while the key holds the ongoing reference to the same atom.

26. Sample Canon and Text

In the following examples, we assume a user has typed or opened some new text which matched existing metadata. A part of that metadata might look like this:

\[(c disney-world
  (@EPCOT EPCOT :long Experimental Prototype Community of Tomorrow)
  (@WALT Walt Disney :born 1901-12-5)
  (@WDW Walt Disney World :long Walt Disney World Resort :opened 1971))\]

Dates such as Walt Disney’s birthday or the opening day of WDW are stored as text which can be interpreted as a date through the use of format expressions.
The Experimental Prototype Community of Tomorrow (EPCOT) was an unfinished concept developed by Walt Disney. Its purpose was to be a "real city that would never cease to be a blueprint of the future," designed to stimulate American industry to develop new ideas for urban living. The city was envisioned as a company town near Orlando, Florida. After his death in 1966, the property became the Walt Disney World Resort in 1972 and a theme park based on the idea opened in 1982.

MSL Recording

(@p1 The (@EPCOT:long) (@EPCOT)) was an unfinished concept developed by (@WALT). Its purpose was to be a "real city that would never cease to be a blueprint of the future," designed to stimulate American industry to develop new ideas for urban living. The city was envisioned as a company town near (@ORLANDO:city-state). After his death in (@1966), the property became the (@WDW) in (@1972) and a theme park based on the idea opened in (@1982).)

The viewer matches words and phrases in the incoming text to the canon it already knows, replacing the literal text with MSL atoms.

New text which is typed or brought into a view creates a new atom. In this case, the viewer called it (@p1). It represents this entire paragraph of typed text. Inside the paragraph, the syntax (@EPCOT:long) pulls the value of the :long key from the metadata associated with the (@EPCOT) atom in the canon, which is the project's long name, "Experimental Prototype Community of Tomorrow." The (@EPCOT) atom by itself pulls the value itself, "EPCOT," from the canon.

A year atom can be specified as (@1966). These are used in the creation of timelines.

27. Text Edits

If an atom is edited in any way, a new copy is recorded with the same key. The old copy is kept but future views refer to the new copy. References in between the old and new atom in MSL text refer to the old one. The sample text in the previous section might be edited and recorded thus:

(@p1 The (@EPCOT:long) was an unfinished concept developed by (@WALT), the last one he worked on before his death in (@1966). Its purpose was to stimulate American industry to develop new ideas for urban living. The city was planned to be a company town near (@ORLANDO:city-state). The original property became the (@WDW:name:) in (@1971) and a (@EPCOT) theme park based on the idea opened in (@1982).)
This version has moved around several parts and deleted others but the same atoms still exist. The opening year of Disney World was corrected to 1971 but the previous (@1972) atom still exists. Furthermore, the entire previous paragraph (@p1) still exists earlier in the stream.

If we type another paragraph, another paragraph atom (@p2) is created next in the stream.

(@p2 In the early (@1960:poss), entertainment industry magnate (@WALT), who by this time had many grandchildren, began to worry about the future of the world they would inhabit. He worried especially about modern cities which he believed were hectic, disorganized, dirty, and crime-ridden, a far cry from his clean and controlled (@DISNEYLAND) in (@CALIFORNIA).)

An edit to an existing paragraph is recorded as a new version of the atom with the same key, in linear sequence in the stream in the order in which it happened.

28. Metadata Selectors

The most important selector is the metadata selector. Metadata is a list of keys and values belonging to an atom. It can be set or recalled with the :metadata-key syntax. The space before the colon is optional and can be omitted when only one metadata-key is being set or retrieved.

Metadata is often used to store factual information that's associated with the atom. An atom representing a person might have metadata for his birthday, for example. Here are some sample atoms that represent elements in the periodic table with four pieces of data stored for each. The first parameter is the atom value and the others are metadata:

1. (@H hydrogen :no 1 :symbol H :weight 1.008)
2. (@Li lithium :no 3 :symbol Li :weight 6.94)
3. (@Na sodium :no 11 :symbol Na :weight 22.990)

We can recall any part of the metadata:

1. (@H) ⇒ hydrogen
2. (@H:symbol) ⇒ H
3. (@Na :weight) ⇒ 22.990
4. (@Li) @(Li:no) ⇒ Lithium #3
Setting a metadata value changes the value in that expression and going forward:

\[(\text{@Li:no }7) \Rightarrow 7\]

This value is factually incorrect since we know the atomic number of lithium is 3, not 7. Values like this which are known to be factually correct can be protected by a \textit{canon} expression.

1. \((@\text{tokyo-disneyland :rides }(@\text{pooh Pooh’s Hunny Hunt}) (@\text{nemo }…))\)
2. \((@\text{tokyo-disneyland :rides}) \Rightarrow \text{Pooh’s Hunny Hunt }…\)

\((\text{v ui-paper})\)

\((@\text{NLS NLS})\)

\((@\text{Portland Portland})\)

\((@\text{ENGELBART Douglas Engelbart :birthplace }(@\text{Portland}) :\text{software }(@\text{NLS}))\)

The definition of \((@\text{ENGELBART})\) relies on the existence of \((@\text{NLS})\) and \((@\text{Portland})\) inside the same containing expression, in this case \((\text{v ui-paper})\). It results in the atom’s value becoming Douglas Engelbart, his :\text{birthplace} becoming Portland, and his :\text{software} becoming NLS from this expression going forward.

\((@\text{DOUGLASENGELBART :software}) \Rightarrow \text{NLS}\)

The full definition of \((@\text{ENGELBART})\) contains both a value and metadata selectors. Atoms like these can be viewed in several ways by the same viewer, allowing editing of metadata in a property sheet, for example. The viewer has full access to the atom’s value and the values of all its metadata, but in narrative text a single value is returned according to the following rules:

1. \((@ \text{key}) \Rightarrow \text{return the value of the atom}\)
2. \((@ \text{key value}) \Rightarrow \text{set the atom’s value and return it}\)
3. \((@ \text{key selector}) \Rightarrow \text{return the result of the selector on the atom}\)
4. \((@ \text{key selector value}) \Rightarrow \text{set the selector to the value and return it}\)
5. \((@ \text{key value selector value}) \Rightarrow \text{set the atom’s value and the selector’s value and return the atom’s value}\)

These examples show all five varieties of rendering in reverse order because 5 is the defining type used to fully define an atom.

Example 5 contains three selectors and values for each of them. It also contains a value for the atom so that is what it returns:
Example 4 contains a selector and its value. It also contains a value for the atom so that is what it returns:

//example 4
(@miss-clairol-27 Lightly Blonde :no 27) ⇒ Lightly Blonde

Example 3 contains only a selector, so it returns the selector value, in this case, a photo:

//example 3
(@miss-clairol-27 :box) ⇒ [file:///box/clairol/miss-clairol/27.jpg]

Example 2 contains only a value, so it returns the value. This is an example of how any paragraph text can be linked to an atom:

//example 2
(@miss-clairol-27 blonder than Bond) ⇒ blonder than Bond

Example 1 contains only a key so it retrieves the last value. The atom was recently redefined to this value. If the atom were wrapped in canon, however, this would not be the case. Canon protects atoms against incidental changes, always providing a fallback value.

//example 1
(@miss-clairol-27) ⇒ blonder than Bond
29. Attribution Using Metadata Selectors

Attribution is the process of keeping track of who wrote the contents of an MSL expression, when, in what language, and what URL or file it came from. These values can be stored in metadata and will often will result in more than one value for the same metadata key within the same atom, such as the two :author values in this atom:

\[
\]

Time and date values can be used to indicate the date or dates of original publication of the content of the MSL expression and to show when the expression was added to the MSL text.

\[
\text{(@DOUBLEHELIX :pub-date 1953-4-25 :edit-date 2019-8-20))}
\]

A format expression can be used to show how a partial date should be displayed.

\[
\text{(@ANIMALFARM “Animal Farm” :author “George Orwell” :pub-date Aug 1945 (f date month year))}
\]

30. RegEx Selectors

A regular expression or regex selector can be used to extract partial values from an atom or its metadata. When followed by a value, a regex selector can perform a search and replace function on the value returned from the atom or metadata.

1. (@WALT:fullname /\w+\s/) ⇒ Walter
2. (@WALT:fullname /\w+\s\w+ Walt) ⇒ Walt Disney

Since these expressions assume that (@WALT) has been previously defined, probably in canon, we are only setting the value for this particular instance. The next invocation of the expression without the regex selector would get the canonical value.

\[
\text{(@WALT:fullname)} \quad ⇒ \text{Walter Elias Disney}
\]

If (@WALT) were not already canonical, the full expression could be recorded like this:

\[
\text{(@WALT Walt Disney :fullname Walter Elias Disney /\w+\s\w+/ Walt)} \quad ⇒ \text{Walt Disney}
\]
This is **2-form** and contains everything we drew from in forming this value. Note that the selector acts on the value in front of it, so the value of `:fullname` in this invocation is Walt Disney.

A regex selector can end with an optional `regex-env` or flag, such as `/g` which selects a global match. A selector like this followed by a value returns the full original value with the match(es) replaced by the second supplied value.

```plaintext
(p1 It is not well known that the Disneyland Monorail System and Disneyland Railroad were owned by a separate business, WED Enterprises, which Walt controlled. Later renamed Retlaw Enterprises ("Walter" in reverse), the business was sold by Disney's heirs to the Walt Disney Company in 1982.)
```

```plaintext
(p1 /business/g company) =>
```

It is not well known that the Disneyland Monorail System and Disneyland Railroad were owned by a separate company, WED Enterprises, which Walt controlled. Later renamed Retlaw Enterprises ("Walter" in reverse), the company was sold by Disney's heirs to the Walt Disney Company in 1982.

### 31. Changing Canonical Values

Canon can be changed going forward by calling another `(c)` expression with the same key. Going backwards in the stream, the old value still exists and older view expressions will show it that way when referred to in another atom like a paragraph.

To write a screenplay with three main characters, we might store some facts about them in canonical metadata:

```plaintext
(c characters
  (@matthew :name Matthew Comlin :age 34 :occupation ramp worker)
  (@brie :name Brie Summers :age 20 :occupation circle k)
  (@janitor :name Janitor :age 63 :occupation janitor))
```

We can apply this metadata to new text using a view:

```plaintext
(v working-draft
  (c characters)
  (@p1)(@p2)(@p3)(@p4))
```

If we decide to change Matthew’s last name, we can change the canonical metadata with another expression:
This generates another view expression because, going forward, we will want to see the new last name. This is exactly the same view expression as before but it refers to the last value of \((c \text{ characters})\) which was defined just before it, as opposed to the previous view which refers to the value before that.

\[
(v \text{ working-draft} \\
(c \text{ characters}) \\
(@p1)(@p2)(@p3)(@p4))
\]

The atom \((@p1)\) here having no value retrieves the last value it had inside this view, just as \((c \text{ characters})\) retrieves the last value it had inside this view. The entire construct from \((c \text{ characters})\) to \((@p4)\) becomes the new value of \((v \text{ working-draft})\).

On-screen representations, printouts, etc. of the text after this view expression would show Matthew Collins whenever \((@matthew:name)\) was accessed.

32. Viewing Atoms Through Alternate Canon

We’ve seen how multiple \((c)\) expressions can apply to the atoms in a view and this can be used to separate different data types, but multiple canonical data can also apply to the same type of information in different views. There cannot, however, be two canonical values with the same key applied to the same atom at the same time. For that, we would need create a separate view.

If we copy the view (made easy by the viewer), we can give it its own name. Then, two views of the same text can appear on screen simultaneously and refer to different canon. These can be shown in parallel windows, pop-ups, or whatever other UI device is desired or available in the viewer.

We can look at the same document using two different dictionaries or two different sets of technical specifications. Two separate views each link together the same text (a series of paragraph atoms in its own view) with a different set of reference materials (canon).
The same intro document can be viewed simultaneously using two different sets of reference materials, like the Algol 60 set and the Algol 68 set. Either of the versions could be published under the same atom.
MSL Evaluation

33. The Atom’s General Form

In Mimix, all data is stored in atoms in MSL expressions. An atom’s current value is maintained in the hashtable and its previous values are written in the MSL stream.
MSL expressions act on data stored in atoms. There are generally two parts to an atom: a key-sequence and a value. The key-sequence is comprised of the atom’s namespace, the atom’s own key, and the metadata-key, if any. This key-sequence is the operand. It gives the current value from the hybrid database. The remainder of the MSL expression is a new value, including any other atoms, values, or transforms it contains. This second portion of the expression is the operator. It tells what to do at the location.

The general rule is that if only a key-sequence is supplied, we get the value of that key-sequence from the hashtable. If a value is supplied, we set the key-sequence in the hashtable to that value.

While it would be possible to construct the entire language using only atom-based expressions, the resulting “atom soup” would be hard to be read. Instead, a hierarchy of namespaces is enforced to help users and the system find a context for values. A special namespace, the canon, provides a layer of value protection over all the other namespaces. Together, these namespaces are searched in a particular order according to the evaluation rules in order to find the current value of an atom or its metadata.

34. Evaluation of Expressions

MSL expressions in general take the form: key selector value transform.

- A key refers to an atom which is unique inside its lexical context.
- A selector can be used to access metadata or text inside the atom.
- A value replaces the previous value of a key or selector.
- A transform takes in the current value and can return a new one to replace it.

An expression is a setter if a value is provided, otherwise it’s a getter. A transform which returns a value acts like a value in the rest of the expression. A containing MSL expression cannot be resolved and processing stops if any of its inner expressions cannot be resolved by the hybrid database. When moving expressions between containing atoms, such as two views in the same machine or two streams from separate machines, expressions must be resolved in the source context and then resolved again in the destination context. This gives the destination viewer the ability to resolve these atoms against known canon and add or discard the new information as the user desires.

- key ⇒ gets the atom (getter)
- key selector ⇒ gets the selector on the atom (getter)
- key transform ⇒ transforms the atom in the viewer (getter)
- key selector transform ⇒ transforms the selector in the viewer (getter)
- key bracketed-transform ⇒ transforms the atom’s value (setter)
- key selector bracketed-transform ⇒ transforms the selector’s value (setter)
- key value ⇒ sets the atom’s value (setter)
- key value transform ⇒ sets the atom’s value and transforms it in the viewer (setter)
• key selector value ⇒ sets the selector’s value (setter)
• key selector value transform ⇒ sets the selector’s value and transforms it in the viewer (setter)

Here are some examples which demonstrate the basic relationships between keys, selectors, values, and transforms. We assume the following canon already exists in the stream:

(c disney-canon
  (@WALT Walt Disney :fullname Walter Elias Disney :wife (@LILLIANDISNEY))
  (@LILLIANDISNEY Lillian Disney))

Although the full (@WALT) atom is available to the viewer and the engine, a typical viewer will most likely display the text value of the atom. This is a getter and no values are set:

//key – gets the atom (getter)
(@WALT) ⇒ Walt Disney

The viewer and engine have access to the full (@LILLIANDISNEY) atom, but a viewer would normally display the value of that atom, Lillian Disney, in a paragraph where this syntax appeared:

//key selector – gets the selector on the atom (getter)
(@WALT :wife) ⇒ Lillian Disney

A transform acts in the viewer to change the appearance of a value or to supply a new one. In this case, the (f bold) format function causes the text to appear in bold but does not provide a return value so it is a getter:

//key transform – transforms the atom in the viewer (getter)
(@WALT (f bold)) ⇒ Walt Disney

The transform [filespec] is used to load the value from a file reference. The result of the transform, the value loaded from the file, becomes the value for (@WALT). This is a setter:

//key bracketed-transform – transforms the atom’s value (setter)
(@WALT [file:///c:/documents/disney/walt-atom.msl])
  ⇒ WALTER E. DISNEY

The formatting expression (f bold) is applied to the value returned from the :fullname metadata-key, making it bold in the viewer. This transform does not return a value:
//key selector transform – transforms the selector in the viewer (getter)
(@WALT :fullname (f bold)) ⇒ Walter Elias Disney

A book's PDF file is loaded into the view's :pdf metadata-key by this [filespec] transform:

//key selector bracketed-transform – transforms the selector’s value (setter)
(v vinyl-leaves Vinyl Leaves :pdf [file:///disney/vinyl-leaves.pdf])

In this next example, the full atom with the changed value is available, though the viewer will likely show only the changed value. The original value is still available in the canon.

//key value – sets the atom’s value (setter)
(@WALT Walter E. Disney) ⇒ Walter E. Disney

35. Evaluation of Parameters

MSL parameters are evaluated differently from other languages in order to meet the language’s goals, including readability first. Parameters in MSL expressions are evaluated or “consumed” by other parameters according to the following rules:

- A selector consumes (takes as its value) everything before the selector.
- An atom consumes nothing.
- A value consumes everything up to the next selector or comment.
- A transform consumes everything up to the next selector, transform, or comment.
- A comment consumes everything up to the end of the expression.

36. Value Retrieval

In practice, value retrieval is accomplished by simply locating the value in the hybrid database corresponding to its key-sequence. The canon namespace exists independently and separately for each namespace in the database and is always searched first.

When it is necessary to rewind the stream and obtain previous values, the MSL text can be examined in reverse chronological order to build any previous version of the system state. The search is conducted in the following order:

1. (@WALT Walter Elias Disney) ⇒ A value defined in the reference itself.
2. (@WALT) ⇒ A value for this atom defined in the containing atom, searching backwards from the current expression to the beginning of the containing atom.
3. (@draft (@chapter-1 (@p1 @WALT))) ⇒ Repeat step 2 until we’re not inside an atom.
4. (v disney-paper (c disney-people) (@draft)) ⇒ A value defined in the containing view, searching backwards from the last containing atom to the beginning of the view.
5. \((s \text{ books } (c \text{ famous-people}) (v \text{ disney-paper})) \Rightarrow \text{A value defined in the containing stream, searching backwards from the containing view to the beginning of the stream.}\)

6. \((w \text{ personal } (c \text{ famous-people}) (s \text{ books}) (s \text{ screenplays}) (s \text{ soccer})) \Rightarrow \text{A value defined in the containing world, searching backwards from the containing stream to the beginning of the world.}\)

7. \((m \text{ david } (@WALT \text{ Walt Disney}) (w \text{ personal}) (w \text{ business})) \Rightarrow \text{A value defined in the containing machine, searching backwards from the containing world to the beginning of the machine.}\)

References to an atom’s metadata in the form \((@\text{atom-key} :\text{metadata-key})\) are searched in the following order:

1. Metadata applied in this atom. \(\Rightarrow (@\text{WALT} :\text{fullname} \text{ Walter Elias Disney})\)
2. Metadata applied in this view, searching backwards from the current expression to the beginning of the view. \(\Rightarrow (v v1 (c \text{ disney-stuff}) (@p1 (@\text{WALT} :\text{fullname})))\)
3. Metadata applied in this stream, searching backwards from the containing view to the beginning of the stream. \(\Rightarrow (s \text{ disney-fiction } (c \text{ characters}) (@p1 (@\text{WALT} :\text{fullname})))\)
4. Metadata applied in this world, searching backwards from the containing stream to the beginning of the world. \(\Rightarrow (w \text{ thesis } (c \text{ famous-people}) (@p1 (@\text{WALT} :\text{fullname})))\)
5. Metadata applied in this machine, searching backwards from the containing world to the beginning of the machine. \(\Rightarrow (m \text{ david } (c \text{ heroes}) (s \text{ disney-book } (v \text{ draft } (@p1 (@\text{WALT} :\text{fullname}))))\)

In each namespace, MSL expressions have access to data defined by this search order:

1. Data defined in the expression itself.
2. Data in the same namespace as the expression, searching back from the line with the expression to the beginning of the parent namespace.
3. Data defined in the parent namespaces, searching back from the line with the parent expression to the beginning of its parent namespace.
4. Repeat step 3 until you reach the top level of the engine or viewer.

A namespace that’s earlier in the hierarchy can contain any later namespaces and, when it does, the containing atom becomes the “parent namespace” described in the search order.

\(\text{(w astronomy}\n\text{(c solar-system)}\n\text{(v jupiter-notes (@p1) (@p2) …))}\)

The entire \((v \text{ jupiter-notes})\) view has access to the \((c \text{ solar-system})\) canon because they are both defined inside the \((w \text{ astronomy})\) world.
This grouping method differs from other ordering systems because it respects *time of insertion* as well as grouping hierarchies. Not all of the data in the parent is accessible, only the data which was defined in time before the expression was written. This means that all MSL expressions have access to data from the past, but no expression has access to data written after it or in the future, unless that data is explicitly provided through a second (later) expression.

In practice, the viewer displays and works with the shortest form of MSL possible. Adding a second paragraph here only requires updating the view, not rewriting the entire reference.

```
(p2 “If I have seen further, it is by standing on the shoulders of giants.” -- Isaac Newton)
```

```
(v whitepaper-draft (p1) (p2))
```

In general, if a value is provided, we are setting the value. If no value is provided, we are getting the value.

```
(s whitepaper
   (p1 This paper introduces (Mimix), a future computer system for research, writing, collaborating, and teaching. (Mimix) is inspired by the ideas of its forebears such as the (Memex), (NLS), and (Xanadu). I hope to achieve with (Mimix) what those platforms could not: a modern, deliverable system which can be used by anyone to augment his or her own intellectual work.)
   (p2 “If I have seen further, it is by standing on the shoulders of giants.”
    -- Isaac Newton)
   (v whitepaper-draft (p1) (p2))
   (v whitepaper-draft (p1))
   (p3 (Mimix) (pronounced “mimics”) is named after (Vannevar Bush :poss)
    (1945) proposal for the (Memex), a machine with similar goals.)
   (v whitepaper-draft (p1) (p3)))
```

37. Currying

Values from selectors and transforms are curried or passed to the subsequent selector or transform in the expression. This allows the expression to read left-to-right and describes changes to the value in that order. Currying allows writing selectors (which invoke engine functions) and transforms (which invoke viewer functions) in a functional programming style.

Currying removes ambiguities in the source of values since they are explicitly stated before the selector or transform. It simplifies understanding of data flow through external functions because only one value can be passed between them and the order of passing is explicit in the expression.
1. (@EPCOT Experimental Prototype Community of Tomorrow /\b[A-Za-z]/g (f upper)) ⇒ EPCOT
2. (@WED (@WALT Walt Disney /\w+s\w+/ Walter Elias /\b[A-Z]/g)) ⇒ WED
3. (@ATIS Adventure Thru Inner Space (f translate :fr en :to es) (f bold)) ⇒ Aventura a Través de Espacio Interior
4. (@arrow Arrow Dynamics, Inc. /(^[\w\s]+)[,]\1/g (f bold) /\w+/ (f italic)) ⇒ Arrow Dynamics

In the first example, the result of the regex selector is curried to the format transform. In example two, the regex match is replaced by the succeeding value, then the entire previous value with the replacement made is curried to the second selector. In the third example, The result of a format transform which returns a value is curried to the succeeding format transform to alter its appearance. In the last example, the regex result is transformed and then further sub-selected by a second regex before being formatted by a transform.

38. Two Forms of MSL

In order to maximize readability, tolerance, and portability (the top goals of the language), MSL can exist in two forms: a short form (called the 1-form) which depends on its containing expression to resolve values and a full form (called the 2-form) which does not. Using this methodology, an entire Mimix machine can be resolved into a single expression which is entirely portable. Likewise, any part down to an atom can also be fully resolved and exported or imported as a self-contained unit.

39. 1-Form

The 1-form of the language is useful for recording editing sessions, for writing MSL scripts for others, and for entering into a REPL-type command line environment. The 1-form includes many shortcuts, optional features, and defaults in order to minimize the amount of MSL that needs to be written and to make the source more readable. However, every 1-form expression must be able to be recast immediately into 2-form in order to be accepted by the engine.

The simplest 1-form expression recalls the value of an atom, which must be previously defined (in a 2-form expression). The 1-form atom expression is written (@ key).

( @ charlie-brown ) ⇒ Charlie Brown

A shortcut is to write the key up against the @ sign and, in fact, this is the more typical syntax.

( @charlie-brown ) ⇒ Charlie Brown

In the 1-form, an expression has only a key-sequence which must be unique within its namespace (stream, view, canon, atom, or metadata). The key-sequence is the one and only parameter to the
expression, hence 1-form. When new expressions are generated (by copying, for example) the viewer assigns a new key immediately. The viewer can use any strategy to assign keys, such as sequential numbers, UUIDs, or lookup of existing atom values for possible matches. The same strategy must be applied anytime an invalid key is referenced in typed or imported code (for example, a key with spaces or another invalid name). Any expression which contains a 1-form expression is, itself, a 1-form expression.

In the 1-form, an expression relies on a previous 2-form expression with the same key to derive its value. 1-form expressions are not portable outside the element in which they’re contained. The viewer may insert the 1-form inside a containing expression where it is valid, but nowhere else. The system always checks its internal representation of the atom’s values to make sure that a 2-form can be derived.

40. 2-Form

In the 2-form, every expression is a setter. Setter expressions have 2 (or more) parameters which essentially reduce to 2 parameters: a key-sequence and a value, thus 2-form. There are no getters and no reliance on previous values. The 2-form of MSL can be directly displayed in a viewer or exported to any other context.

1. (@CharlieBrown Charlie Brown)

In both the examples, the atoms fully set their values so both are examples of 2-form.

Any 1-form expression can be converted to 2-form by retrieving the current value of the atom and adding it as a value parameter.

2. (@CharlieBrown) => (@CharlieBrown Charlie Brown)

If these three atoms appeared in this order in a view, the first one provides a value for the atom itself, “Charlie Brown” and some metadata. The second expression (1-form) can be converted to 2-form by adding the value of the atom from the previous expression with the same key. The third expression references metadata (1-form) and can be converted to 2-form by resolving the atom’s value and metadata references.

When constructing the 2-form, the viewer takes only the data needed to resolve the atom’s value.

Any getter can be rewritten in 2-form as a setter simply by running all the getters within it and adding...
the values they return as a second parameter in the expression. This turns the getters into setters and makes the single 2-form expression fully portable.

```
//set the keys to their original values
(@GEORGE George Washington)
(@USA United States of America)

//@p1 as a getter for @GEORGE and @USA
(@p1 (@GEORGE) was the first President of the (@USA).)

//@p1 recast as 2-form with no dependencies
(@p1 (@GEORGE George Washington) was the first President of the (@USA United States of America).)
```

These examples show the two simplest kinds of getter-setters but these have an underlying form with further options. A getter-setter can contain any combination of selectors, values, and transforms.

A selector function tells how to find the value you want inside the atom. A metadata selector function can retrieve values from the atom’s metadata, for example. A regex selector can subselect from an atom’s value by using a regular expression. These can be combined, so that you can take the regular expression results from a metadata value, for example.

A value itself will set the atom’s value (or whatever is pointed to the by selector). A value can contain any number of atoms or literal text values strung together (and these can, of course, have values and metadata references).

Before being applied to the selector, a value can be run through a transform. This can be used to apply changes to data before it is set as the atom (or selector’s) value. An URL transform can retrieve something from an URL and use it to set the value. A file transform can load something from a file into a value.

A transform function is also used to apply formatting, such as that used by the viewer. A transform function which does not return any data does not change the data in the atom on which its called; it is simply used for its side effects.
41. Reaching

An atom cannot contain atoms in a higher namespace, but it can reach them and access their values. Reaching takes place when atom explicitly refers to a namespace before it in the universal order:

1. machine
2. world
3. stream
4. view
5. atom

```
(m my-machine
 (w some-world
   (s this-stream
    (v draft-view
     (@p1 para one text)
     (@p2 para two text))
    (v second-draft
     (@p1 different p1)
     (@p2 (v draft-view (@p2)))))
```

This definition of `(v second-draft)` has an atom `(@p2)` is that is reaching for a namespace above it `(v draft-view)`. The namespace above that, the stream `(s this-stream)`, is searched for `(v draft-view)` to retrieve the value of `(@p2)` there.

Here is `(v second-draft)` in 2-form:

```
(v second-draft
 (@p1 different p1)
 (@p2 (v draft-view (@p2 para two text))))
```

A reaching expression cannot change values in the namespaces it reaches into, but it can be used to recreate the expression’s environment from scratch. The 2-form version of `(v second-draft)` cannot alter the value of an existing `(v draft-review)` or its `(@p2)`, but an expression like this can be used to recreate the dependency environment if it does not exist already. If `(v second-draft)` were imported into a new stream, for example, it could be used to build a `(v draft-view (@p2 para two text))` in that stream.
Here’s another example of reaching for data in a different world:

```
(m my-machine
  (w work-projects
    (v phone-list (@larry Larry Smith :cell 305-555-1212))
    (w personal-projects
      (v contacts (@larry (w work-projects (v phone-list (@larry)))))
    ))
)
```

And `(w personal-projects)` in 2-form:

```
(w personal-projects
  (v contacts (@larry (w work-projects (v phone-list (@larry Larry Smith :cell 305-555-1212))))))
```

Reaching for data in another machine:

```
(m david
  (v our-friend-the-atom (m nuclear-machine (s books (v our-friend-the-atom))))
)
```

`(v our-friend-the-atom)` in 2-form:

```
(v our-friend-the-atom (m nuclear-machine (s books (v our-friend-the-atom @chapter1 ... @chapter2 ...))))
```

The entire `(m david)` machine in 2-form:

```
(m david
  (v our-friend-the-atom
    (m nuclear-machine
      (s books
        (v our-friend-the-atom @chapter 1...) (@chapter 2...))))
)
```

This expression could not be used to overwrite the `(m nuclear-machine)` if it already exists, but it could be used to create one if it does not.

Every MSL expression must be complete within its containing expression. In other words, every expression in 1-form must resolve immediately to 2-form in which no getters are required. It is an error to write or hold in memory an MSL expression which doesn’t resolve fully to 2-form. This is enforced by the viewing environment, which assists the user in resolving expressions as MSL code is moved between contexts.
42. Moving and Sharing Atoms

We can use a view to bundle together atoms for display. If you think of an atom as representing a word, a name, a sentence, or a paragraph, then a view is a list of paragraphs you want to see in a certain order, like a document or a note.

\[
\begin{align*}
\text{(v \text{ui-paper The History of User Interfaces}} & \\
\text{\quad (@p1 (@DOUGLASENGELBART Douglas Engelbart), famous for his}} & \\
\text{\quad (@DOUGLASENGELBART :software (@NLS NLS)) software, was born in (@DOUGLASENGELBART :birthplace Portland, OR).}} & \\
\text{\quad (@p2 (@DOUGLASENGELBART Doug) eventually left (@SRI SRI) after the (@NLS) IP}} & \\
\text{\quad was sold to (@Tymshare Tymshare).))} & \\
\end{align*}
\]

Typically, we would extract everything known in this view and store those atoms in a canon.

\[
\begin{align*}
\text{(c engelbart}} & \\
\text{\quad (@DOUGLASENGELBART Douglas Engelbart :software (@NLS) :birthplace Portland, OR)} & \\
\text{\quad (@NLS NLS :fullname oNLine System :inventor (@DOUGLASENGELBART))} & \\
\text{\quad (@SRI SRI :fullname Stanford Research Institute)} & \\
\text{\quad (@Tymshare Tymshare :fullname Tymshare, Inc.))} & \\
\end{align*}
\]

Defining atoms in canon gives them a layer of backup protection in that their values always revert to the canonical values, even after you override them in a particular instance.

We can bundle together canon and views inside a stream, giving the atoms inside the views access to the canonical values.
This is the shortest possible form of the stream because it contains everything necessary to resolve all references inside the stream itself. It is a 1-form expression, as the shortest version often is, because it contains references like (@NLS) that depend on previous expressions inside. From any expression inside this stream, however, the viewer can create a full form of that single expression which has no dependencies.

If the user would like to move (@p1) to a different context, the viewer can copy this 2-form version:

```
(@p1 (@DOUGLASENGELBART Douglas Engelbart), famous for his (@DOUGLASENGELBART :software (@NLS NLS) software, was born in (@DOUGLASENGELBART :birthplace Portland, OR).)
```

This allows the expression to be moved into any other context in MSL where it would be legal. It brings along with it everything it “knew” from the previous context, and all of it plain text. This allows the second viewer or engine to integrate the incoming information in any way that the user sees fit.

If this paragraph were pasted into a stream which had a different canonical definition for (@DOUGLASENGELBART), then these invocations would only change the values where they appeared. Any other writing in that stream would still have access to the canonical values. A viewer could detect this condition and offer to use some or all of the canonical values of the receiving context instead.

The viewer might also choose to “lift” these new values to canon and re-write the MSL as such in the new context. This allows the user to turn the previous values on and off in the new context and to reuse them later without re-pasting. The viewer could even copy the entire canon from which the definition of (@DOUGLASENGELBART) was taken. It’s entirely up to the user, but the viewer should offer both simple and complex options and re-write the MSL code as it is pasted to effect the desired results.
43. Rewinding

In practice, rewinding is achieved by maintaining both the current value of every atom (in the hashtable) and the MSL which was used to create it (in MSL text files). Granularity is achieved by searching through MSL backwards, beginning with the expression from which the rewind is started, through the expression where the rewind is ended, then backwards from the ending point up through the containing expression until all values can be rendered in 2-form with no dependencies. Note that it is not necessary to re-evaluate all the MSL “from the beginning” to determine an atom’s previous value. It’s only necessary to evaluate the last expression which set the atom’s value, working backwards.

44. MSL Text

In its most basic form, an MSL program is a plain text file with MSL expressions executed in the order in which they appear. Any MSL text file which is processed by the engine has the effect of executing the expressions within in it in the context of the present state. In other words, any new text file of MSL introduced into the engine will create or modify the system state as necessary to reflect the information contained in the file’s expressions.

A Mimix machine can start with an empty state only if no MSL text file is provided. Typically, a machine would be started with a start.msl file with MSL expressions that load the canonical data and views that are desired when starting the machine for the first time. A start.msl file could load all the streams required by a particular corporation, department, or university class.

In practice, an MSL text file can be encrypted when it is being stored and decrypted before use.

45. Sources of MSL

Any value in MSL code can draw from another source. There are two kinds of sources for MSL values. The first is other MSL elements. As long as these are known to the system, they can be used as values in an expression.

    (@WALT Walt Disney :wife (@LILLIANDISNEY))
    (@LILLIANDISNEY Lillian Disney :husband (@WALT))

Circular references are prevented because the selectors only apply to the atom they’re directly inside. Anything after a selector is a value or a transform. In other words, there is no way to get from Walt to Lillian’s husband or to change values in (@WALT) while accessing (@LILLIANDISNEY).

You can refer to an atom (or a view, stream, etc) far away as long as the expression returns a value that is valid for the containing expression. Immediate recasting to full form ensures this value is valid at the time the expression is run.
The resulting value, the value of (@walter-elia-disney) inside the named canon inside the named stream, world, and machine, is applied to the new atom (@WALT) in the current context. As long as all these other references resolve immediately, this expression is valid. It has already been recast as a setter which can be used to reconstruct the entire context in which it was called.

This short example assumes only a simple value for (@walter-elia-disney).

('@WALT (m david (w personal (s disney-book (c disney-people (@walter-elia-disney) Walt Elias Disney)))))

This expression alone can be used to construct the entire machine, or at least enough of it to process itself. If (@WALT) does not exist in the current context, it can be created. The next parameter sets its value.

If a machine named (m david) does not exist, it can be created. The next parameter sets its value.

If a world named (w personal) does not exist, it can be created... and so on, down to the canonical atom (@walter-elia-disney) which is created if it does not exist.

Another source for MSL is a file reference. This is provided through a transform function which supplies a value to the containing expression.

(s term-paper (c syllabus [file:///Users/.../term-paper/syllabus]))

This would retrieve the named file and supply it as the value to the (c syllabus) canon. It would be an error if this file did not return a valid canon. The viewer is responsible for processing all transforms and must supply the engine with valid MSL to replace this URL reference. Typically, a viewer would translate the disk file to plain text or MSL if it is not already in one of those formats, then return the MSL text as the value in the expression. A :url metadata-key could be appended with the original URL before the MSL is submitted to an engine.

A third source for MSL is a URL. This is provided through the same transform function as files.

(s shakespeare-research (v romeo-and-juliet [http://archive.org/romeo-juliet]))

This would retrieve the file from the URL and supply it as the value of the named view. It would be an error if this URL did not return a view. Here again, this would be managed by the viewer. A plain text file is a valid view.
46. About Data Types

Data type selectors inform an engine and viewer how to interpret values so that they can be optimized for storage and display. Types are not required and do not need to be declared in advance. Any data types which are not understood by the viewer are simply ignored in rendering the screen. Any data types which are not understood by the engine are stored and ignored.

Data types can flexibly represent real information that might be found in narrative text without inducing breakage from unusual values, such as the very large and small numbers found in scientific notation. Data types can be used to denote inline code which is useful in creating viewers for code editing or programming language development. Data types can also indicate binary results, such as a link to a YouTube video in narrative text that could be played directly in the viewer.

Data types serve as a way to select the desired meaning from an atom or its metadata and so they are the third type of selector. Like other selectors, data types are preserved when an atom is used again later, so they only need to be set once. Data types can also be set in canon. Data types are different from formats because they denote how a value should be interpreted rather than how it should appear in the viewer.

1. (@10M 10000000 (d number)) ⇒ 10000000
2. (@10M 10,000,000 (d number)) ⇒ 10000000
3. (@10M 10.000.000 (d number)) ⇒ 10000000
4. (@10M 10M (d number)) ⇒ 10000000
5. (@10M 10 million (d number)) ⇒ 10000000
6. (@10M 1.0E7 (d number)) ⇒ 10000000
7. (@ONE-THIRD 10/3) ⇒ 10/3
8. (@ONE-THIRD (d number integer)) ⇒ 3
9. (@ONE-THIRD (d number real 3)) ⇒ 0.333

Data types can be combined with formats to change how values are shown are the screen without changing their representation in narrative text. This allows factual values such as "$5M" to be captured in narrative text and used as numbers in comparisons, conversions, or other operations.
These data types might be used in a narrative expression as such:

(@p1 By (@1995), the resort developer had invested over (@cost PHP 253284500 (d number) (f currency php)) or about (@cost (d number integer) (f currency convert usd) US dollars in the project.)

By 1995, the resort developer had invested over ₱ 253,284,500 or about $50 million US dollars in the project.

Later writing which referred to (@cost) could use a different format to view the original number in a new way, such as adjusted for inflation.

(@p2 This represented a significant commitment to local economy of the time, equivalent to more than (@cost (d number integer 2) (f currency convert usd words) (f currency inflate 1995 2019)) today.

This represented a significant commitment to the local economy of the time, equivalent to more than $83 million today.

47. Recommended Data Types

data-type ::= date | time | number | text | image | audio | video | code | app | binary | custom

The following sections discuss suggested data types that can be used when building engines and viewers. None of these types is required to implement a viewer or engine, but a basic viewer or engine should be able to cope with these recommended types.

It is anticipated that many more data types will be developed for specialized text and applications, such as 3D models, chemical and mathematical formulas, complex series, and so forth. Data type expressions use the same key selector value syntax as other MSL expressions which allows unlimited types and parameters for types.
48. Date

All of the values in the following examples represent narrative text that can be interpreted by a person or program as dates:

1. (@WALT :birthday 1901 (d date))
2. (@WALT :birthday December 1901 (d date))
3. (@WALT :birthday 1901-12-5 (d date))
4. (@WALT :birthday 12/5/1901 (d date))
5. (@WALT :birthday Tuesday (d date))
6. (@WALT :anniversary next year (d date))
7. (@US-DATE 11/1/68 (d date month day year))
8. (@EURO-DATE 1/11/68 (d date day month year))
9. (@BIGBANG :start -1.38e10 (d date year))
10. (@LASTWEEK last week (d date range))
11. (@NEXT-QTR next quarter (d date range))
12. (@TENANCY since December (d date range))

(@ROME The Roman Empire thrived between ~(@PERIOD 31 BCE and 476 AD (d date range)) :start (@PERIOD \w+ and\) (d date year) :end (@PERIOD \and \w+) (d date year)) ⇒ The Roman Empire thrived between ~31 BCE and 476 AD.

Examples 7-9 show how data types like (d date) can include additional parameters to further explain their values. Examples 7 and 8 show how different narrative text can represent the same date. Example 9 shows a nontraditional value for a date, the start of the Big Bang 13.8 billion years ago. Neither the number nor its scientific notation would fit within the date fields of most storage representations and could not be sorted or filtered in those systems.

Data types like these can also help a viewer conform documents to a single internal representation or visible format, even if those documents use different formatting themselves. A reader consulting documents in both European and US date formats could sort, filter, or conform those documents using the viewer.

Custom date types can be used to represent non-Gregorian calendars (Mayan, Chinese, Japanese, Islamic, etc.) or to specify complex date expressions such as date ranges or offsets.
49. Time

Time types can be used to represent simple time values as well as complex custom times for special applications. As with other data types, times can be used to express ranges, offsets, or alternative time measurement systems which might be encountered in research and reading. Time types can be combined with formats to preserve the original text yet change its appearance in the viewer.

1. (@meeting 8am (d time 12))  ⇒ 08:00:00
2. (@meeting 0800 (d time 24))  ⇒ 08:00:00
3. (@meeting (f time convert-city Houston Manila))  ⇒ 22:00:00
4. (@movie-frame :time 00:30:19.24 (d time smpte))  ⇒ 00:30:19.24
5. (@movie (@clip (@movie-frame :time) +00:01:00 (d time range smpte)) (d video))  ⇒ 00:30:19.24- 00:31:19.24
6. (@experiment :length 342 (d time millisecond))  ⇒ 342
7. (@sunrise :start 5:15-5:18 am (d time range))  ⇒ 5:15 am – 5:18 am

50. Number

Number types are designed to represent various types of numerical values as they might appear in narrative text in order to normalize them for storage, for presentation in the viewer as a fallback to format functions, and as parameters for advanced format functions like auditing, aggregation, or sorting.

Numbers are composed of integer and real subparts. Every number can be expressed using any of these types. Integer types are those which don’t include decimal places, taking all of their values as integers. Real types are those which include a decimal portion. Scientific types are expressed in scientific notation using an upper or lowercase E character to indicate the exponent portion.

1. (@TWENTY-4-HUNDRED 2400.33 (d number integer))  ⇒ 2400
2. (@TWENTY4-HUNDRED (d number real))  ⇒ 2400.33
3. (@TWENTY4-HUNDRED (d number real 3))  ⇒ 2400.330
4. (@DOLLAR-RANGE $2-3 million (d number range))  ⇒ 2000000-3000000
5. (@TWENTY4-HUNDRED 2.4e3 (d number sci))  ⇒ 2400

51. Image

Image types can be used to facilitate the storage of binary files that should be rendered as images in the viewer. The engine will likely store these binary images outside MSL text while still allowing their retrieval and use from within atoms. Image type values are usually supplied through [filespec] and [url] transforms. As with all transforms, file or URL content (in the case of images, the image file) is stored by the engine itself and the transform source is recorded in the atom’s metadata.
52. Audio

Audio, like other binary types can be used by the engine to optimize storage of the original files. Audio types can also be used by viewers to assign file type information to a single audio file or a group of them, regardless of the file names or URLs. Finally, audio types can help viewers generate a default player interface without calling a specific format function.

1. (@song [http://youtube.com…] (d audio mp3)) ⇒ audio stream from video file
2. (@song [file:///music/edm/gravity] (d audio flac)) ⇒ specifies file type
3. (c edm (d audio mp3)) ⇒ type of all atoms in canon
4. (@music [.../file1.mp3] [.../file2.mp3] (d audio mp3)) ⇒ audio playlist
5. (@music (d audio fantasound)) ⇒ audio special format

53. Video

Video types are used by the engine to optimize storage of the original files. Video types can also be used by viewers to assign type information to a file or a group. Video types can help viewers generate a default player interface without calling a specific format function.

1. (@song [http://youtube.com…] (d video mp4)) ⇒ video stream
2. (@song (d video webm) (f video new-player)) ⇒ video alternative file type
3. (v disney-videos (d video mov) (@vid1 [filespec])…) ⇒ set type for video library

54. Code

Code types can be used to designate inline code which should not be parsed like narrative text. Like other types, code types can also provide a fallback format in the viewer in lieu of specifying a particular format function. Code types should not be used to include code which is intended to be run in the viewer, as formats are provided for this function (the format being defined in the viewer itself.)

Code types are primarily useful in reading and writing software documentation.
55. App

An app type is designed to indicate a reference to an external application which runs outside of the viewer. An app type can be used by the viewer to launch the external application and pass the atom's values or metadata as parameters. App types are useful when narrative text refers to information that requires an external program to access or manipulate, as opposed to information that can be readily resolved or displayed inside the viewer.

1. (@train-ticket Versailles Special (d app eurotrain-booking :from Paris :to Versailles :type winter-discount))
2. (@parts-request Find Replacement Filter for 223C (d app parts-master :model 223C :type replacement-filter))
3. (@dna-sample Tomato DNA Sample [http://dna.org/] (d app MendelView))

56. Binary

Binary types are designed to represent a binary blob which might be referenced in narrative text, such as a file in an atypical format. Binary type information is not expected to have a text representation and data designated as binary type is not parsed further.

1. (@hello-world 72 69 76 76 79 32 87 79 82 76 68 (d binary ascii))
2. (@posgresql-database [file:///files/sql/my-db.sql] (d binary postgres))
3. (@wp-backup Thursday Backup [http://myco.wpengine.com/] (d binary zip))
4. (@h Hydrogen :hologram [file:///d:/holo/periodic/h.hol] (d binary holo))

57. Custom Types

Custom types can be used to designate any other type of data which is not represented in these recommendations, or to provide alternative syntax for a type. Custom types should be used only where they provide beneficial information to the engine in storing the data or to the viewer in interpreting it for display. Data-about-data which can be represented as text should be stored as metadata rather than using custom types.
1. (@script Ibuprofen :amt 500 mg :freq tid (d rx))
2. (@rocket-sensor-values [wss://stream-source] (d bitstream))
3. (@arithmetic-math πr^2 (d math))
4. (@msds-library [filespec] (d msds multiple))
Glossary

1-form
The simplest form of an expression which specifies a key alone without a value. Any expression which contains a 1-form expression is, itself, 1-form. Every 1-form expression must be able to be immediately resolved into 2-form in order to be accepted by the engine. So named because it has only one element, a key.

2-form
The full form of an expression which specifies a key and a value. Every expression inside a 2-form expression must, itself, be 2-form. A 2-form expression is fully portable and has no dependencies. So named because it has two elements, a key and a value.

3-form
A compact, portable form of an expression rewritten so that the first invocation of each atom and every subsequent change to it are in 2-form while later or intervening recall-only invocations are in 1-form. So named because it combines 2-form and 1-form (2+1).

archive
A snapshot which is saved to disk, usually indicated by a data type. An archive can be optimized by rewriting it in 3-form before saving.

@ The (@) or atom namespace. Atoms in this namespace cannot contain atoms from any other namespace, but they can have subatomic values.

atom
The fundamental unit of data storage. Atoms are identified by a unique key within their namespace. They are assigned values via expressions and recalled according to evaluation rules. Atoms can have their values affected by selectors and transforms.

attribution
The use of metadata to store information about the source of an expression’s value, such as its authors, dates, or languages. Attribution can result in multiple values for the same metadata key.
bracketed transform
A transform written in square brackets which contains a filespec or URL. The transform returns the file or value found at the location. Often used with data types to store non-text data.

burroughs-intro
An example MSL stream cited in the Specifications which shows how alternate canon can be applied to the same atoms using two different views.

canon
The specialized (c) namespace which is evaluated first inside the other namespaces. It provides a layer of data protection through reset values. Canon can only contain other canon or atoms.

canonize
A form of lifting in which the all of an atom’s metadata from every invocation in a stream is rewritten as a single 2-form expression. The resulting canon expression can be lifted to a higher namespace and used to generate a compact 3-form of the stream.

comment
Text after // which appears at the end of an expression but is ignored. Comments are not retrieved by evaluation and are not hashed but are stored in MSL text.

compositing
The process of building an atom’s value or metadata value from (the same or different) atom or metadata values, alone or in combination with literal inline values. Used to apply data types or formatting to a sequence of values or to preserve a complex value for later use. Formats can also be composited.

currying
The method of chaining the value from a selector or transform to the next selector or transform in the expression, modeled after the lambda calculus.

data type
A selector which triggers special functions in the engine. Used to assist the engine in storing and the viewer in displaying non-text data.
engine
Provides an interpreter for MSL expressions along with storage via the hybrid database. Accepts and returns only MSL expressions. Typically accessed by a viewer via a websocket.

evaluation
The process of resolving a 1-form expression into 2-form according to the namespace hierarchy and the rules defined in the Specifications and taking into account the past and present values of the referenced atoms in the hybrid database.

expression
A syntactic element of the MSL language modeled after Lisp s-expressions. Expressions are enclosed in parentheses and follow the general form: (key-sequence value). Keys, bracketed transforms, metadata selectors, regex selectors, hashes, and comments are not MSL expressions.

filespec
The literal value of a bracketed transform stored in the atom’s bracketed transform namespace and used to return an atom or metadata value from a local or network drive.

format
A transform which can run arbitrary code in the viewer, identified by its key. The invoked viewer function receives the preceding value in the expression as a parameter, along with any metadata supplied inside the format expression. Used to format or alter data beyond its plain text representation. A format transform can optionally return a value.

functional
The programming style described by Backus in which every function takes only one parameter and returns only one parameter, both of which are fully defined. No function may accept, pass, or return a null value.

hash
The result of running an expression through a hashing algorithm. The hash value obtained is recorded in the subsequent instruction, forming a chain.
hashtable
The internal storage used by the engine to quickly evaluate expressions. Contains only the most recent value of each atom.

hybrid database
The combination of a hashtable and MSL text which can be used to recall present expression values or rewind values from the past.

invocation
A single appearance of an atom in narrative text. The same atom can appear with different values in different invocations. A 2-form invocation supplies its own value. A 1-form invocation’s value depends on the evaluation of previous expressions in the MSL text.

key
An atom’s unique identifier within its namespace. Every invocation of the key within the namespace refers to the same atom.

key sequence
The complete path to where a value resides in the hashtable, including all its containing namespaces in sequence, its atom key, and its subatomic keys, if any.

lifting
A special form of rewriting in which one or more atoms is elevated to a higher namespace, changing its evaluation rules going forward.

machine
The (m) namespace which represents an entire physical or virtual machine and all the worlds, streams, views, canon, and atoms it knows about.

merging
The process of integrating atoms from one stream into another while resolving any key collisions. Atoms which should retain the same keys can be combined as in canonization. Atoms which are different can have their keys changed before merging.
metadata
A subatomic namespace which belongs to each atom. Within the namespace, any number of unique keys can exist.

metadata selector
The syntactic part of an expression beginning with a colon (:) which selects a key in the atom’s metadata namespace.

Mimix
The entire ecosystem of MSL, a viewer, and an engine. Can also refer to the machine (m) namespace.

miss-clairol
An example MSL view cited in the Specifications which demonstrates many of the core concepts of the language.

MSL
Mimix Stream Language, a functional programming language for representing text review and editing sessions in a viewer and engine.

MSL text
A series of MSL expressions executed in order. Typically stored in a file or memory buffer.

namespace
A method of grouping atoms according to MSL evaluation rules. An atom cannot contain an atom which is higher up in the namespace hierarchy, but it can retrieve its values by reaching.

namespace hierarchy
The machine (m), world (w), stream (s), view (v), canon (c), and atom (@) namespaces, in that order.

nebula
A system for deploying MSL applications which combines a viewer and local engine and allows access to remote engines.
reaching

The process of retrieving a value from an atom higher up in the namespace hierarchy. Values cannot be set by reaching.
recasting
The process of converting an existing document which is not MSL, such as plain text, a word processing file, or a PDF, into MSL text.

recording
The viewer’s process of generating MSL expressions in response to user interactions, such as reading, writing, annotating, or editing text.

regex
A regular expression provided as the literal value in a regex selector. Stored separately by the engine in a subatomic namespace.

regex selector
A selector which applies a regular expression to the preceding value and returns its result. If followed by a value, the new value replaces the result in the original expression.

reset value
In a 1-form expression, the value of an atom or its metadata retrieved from canon which overrides any previous 2-form invocation.

rewinding
The process of retrieving the chain of previous values or metadata from an atom when they differ from the values presently in the hashtable. Achieved by examining the MSL text backwards to the last 2-form expression which set the value of that key. Rewound values can be reintroduced into the hashtable by rewriting.

rewriting
The process of composing a new MSL expression from an existing expression or stream while preserving the underlying values. Used in lifting, canonization, merging, rewinding, and viewer operations such as copy-paste or bulk transformations.

selector
A syntactic element inside an expression which selects the atom’s metadata, applies a regex, or denotes a value’s data type. The literal values supplied to selectors are stored in subatomic namespaces.
snapshot
The MSL text generated by the viewer in response to a user’s selection within a stream, such as a paragraph (atom), document (view), or body of work (world). Used in copy-paste, in moving streams between engines, and in creating archives.

stream
The (s) namespace which represents a series of views over time. In Nebula, a collection of MSL texts from a specific machine. More generally, any sequence of MSL expressions.

subatomic
The group of namespaces which sit below the atom. Includes metadata selectors, regex selectors, datatype selectors, bracketed transforms, and format transforms. Within these namespaces, each metadata selector, datatype selector, and format transform has its own key. Subatomic namespaces can store multiple values for the same key, a feature used in attribution and currying.

transform
The syntactic element of an expression which triggers special functions in the viewer. Used to load file specs and URLs and to enable viewer formatting of the atom’s data.

URL
The literal value of a bracketed transform. Stored in the atom’s bracketed transform namespace and used to return an atom or metadata value from a web address.

value
In a 1-form expression, the actual contents of the hashtable located at the key-sequence determined by evaluating the expression. In a 2-form expression, the newly supplied data to be stored at a key-sequence indicated by the expression. Every expression can be resolved to a single value according to the evaluation rules, resulting in default data for the viewer to display.

view
The (v) namespace used to group atoms together for display.

viewer
An application for working with MSL expressions through a customized interface, such as a word processor.
Walt
The (@WALT) atom used throughout the Specifications and in Mimix testing to demonstrate MSL evaluation rules. Named in honor of Walt Disney.

world
The world or workspace (w) namespace used to group streams together to organize areas of work or study. In Nebula, the user interface provided to manage streams from a particular engine, such as a Local World.