Syntax Reference

Program Structure

```
# An Eve program

The following is a block of Eve code.

```
  match // the match phase of a block
    ...
  bind // the action phase of a block
    ...
```
```

Eve programs are written as you would write Markdown, except the code blocks are executable Eve code. Each block of Eve code is written in two phases: First, in the “match” phase, you collect all relevant records from the Eve database. Second, in the “action” phase, you change records in the Eve database by adding, removing, setting, or merging records.

Record

```
// addresses of people who are 30 years old
people = [#person age: 30, address]

// These two lines say age is equal to 30
person = [#person age: 30]
person.age = 30

[#person brother: [@Ryan]]
```

Data in Eve is represented as records, a set of key value pairs associated to an ID. Blocks match records and update or create records based on what was found.

Attributes can be accessed directly by using their name, or they can be accessed using dot notation.

Records can be nested to find more complex patterns.

Names

```
// These are equivalent
[@Chris]
[name: "Chris"]
```

A common attribute for records is “name”, which can be accessed using the @ operator. Names are typically used to refer to a particular unique record in the Eve database.

Tags

```
// These are equivalent
[#person]
[tag: "person"]
```

Another common attribute for records is “tag”, which can be accessed using the # operator. Tags are typically used to refer to collections of records in the Eve database.

Equivalence

```
// Pairs of people with the same age
person = [#person age]
person2 = [#person age]

// People aged 10
[#person age]
age = 10

// Something that's never true
x = 10
x = 100
```

Eve doesn't have assignment, only equivalence. Things with the same name are equivalent, and things = to each other are equivalent. The final example shows the difference between this and assignment. x is not first 10 and then 100, but instead we're saying that 10 = 100, which will never be true.

If-Then

```
guest = if p = [#person] then p
        if [#person spouse] then spouse

(pts, grade) = if score > 90 then (4, "A")
               else if score > 80 then (3, "B")
               else (2, "C")
```

If provides conditional equivalence. Here we're stating that guest is equivalent to all the people and the spouses of people.

The second example uses else to make the options exclusive (only the first matching clause will be taken) and does multiple returns.

Functions

```
// The sin function being used with degree input
x = sin[degrees: data]

// The sin function being used with radian input
x = sin[radians: data * π / 180]
```

Functions work just like any other record, but have an implicit return value. Functions operate element-wise on their input, akin to the map() function in other languages. Arguments are explicitly defined when the function is called, so they can be written in any order. This also allows for optional arguments and alternative calling patterns.

Aggregates

```
total-employees = count[given: employees]
budgets = sum[value: salary, given: salary, per: department]
```

Aggregates are functions that collapse a set to a single value, like sum, count, or max. This behavior is akin to the reduce() or fold() function in other languages.

Not

```
// people who are not employees
person = [#person]
not(person = [#employee])
```

You can check for the absence of conditions using not. In this case, we're specifying that the person is not also tagged employee.

Commit

```
// Saves Chris' age
commit
  [@Chris age: 30]
```

Commit is one of two action fences. This fence tells Eve to persist the records behind the fence, even if their supporting data are removed.

Bind

```
// Prints the current time
match
  [#time hours]
bind
  [#div text: "It is {{hours}} o'clock"]
```

Bind is one of two action fences. This fence tells Eve to update records behind the fence as their supporting data in the match phase change. This makes the actions reactive - values update as the data in Eve changes.

Action Operators

```
match
  chris = [@Chris]
commit
  chris.age := 30
  chris.favorite-food += "pizza"
  chris.favorite-color -= "blue"
  chris <- [eyes: "green", hair: "brown"]
  chris := none
```

There are four relevant operators in the action phase: add, set, remove, and merge.

- Add (`+=`) - adds value to an attribute
- Set (`:=`) - sets the value of an attribute
- Remove (`-=`) - removes the value from an attribute
- Merge (`<-`) - merges one record into another

	Using the set operator with the none keyword removes the record from the database entirely.