Assignment 10.1 Traits in Lua

Trait composition + is defined as a symmetric join \( \sqcup \) between two maps \( c_1, c_2 \):

\[
(c_1 + c_2)(n) = b_1 \sqcup b_2 = \begin{cases} 
  b_2 & \text{if } b_1 = \bot \lor n \notin \text{pre}(c_1) \\
  b_1 & \text{if } b_2 = \bot \lor n \notin \text{pre}(c_2) \\
  b_2 & \text{if } b_1 = b_2 \\
  \top & \text{otherwise}
\end{cases}
\]

with \( b_i = c_i(n) \)

The following Lua function dispatches lookups for key \( k \) from map \( \text{receiver} \) to the two maps \( m_1, m_2 \) in an ordered fashion with priority on \( m_1 \):

```lua
function asymmetricDispatch (receiver, k)
local v = receiver.m1[k]
if not v then return receiver.m2[k] end
return v
end
```

1. Provide a Lua implementation of the function `symmetricDispatch(receiver, k)`, which implements dispatching of key \( k \) based on the symmetric join \( \sqcup \).

2. Use this function to implement a function `composeTraits(trait1, trait2)`, which takes a pair of trait maps as input and creates an object-like map as output, that delegates its lookups to the traits in symmetric join fashion.

Assignment 10.2 Delegation & Prototypes

A Lua interpreter, implemented in Java, uses the following two Java data types to represent Lua tables and closures (anonymous functions):

```java
interface Table {
  Object get(String key); // key's value in the internal hashmap
  void put(String key, Object val); // bind key to value
  Table getMetatable(); // null if no metatable
}
interface Closure {
  Object execute(Object... params); // interprets this closure
}
```

The interpreter works on a Lua program’s syntax tree. Implement a Java method

```java
static Object eval(Table table, String key)
```

for the interpreter, which evaluates a Lua sub-expression of the form `<table>.key` as occurring e.g. in the following Lua code in line 12:
Assignment 10.3 Stream Wrapper Mixin with Prototypes

Consider the following Lua code:

```lua
Stream = {}
Stream.__index = Stream
function Stream:write(character) ...
  ...
end
function Stream:new(object)
  setmetatable(object,self)
  return object
end
Mutex = {}
Mutex.__index = Mutex
function Mutex:lock() ...
  ...
end
function Mutex:unlock() ...
end
function Mutex:new()
  object = {}
  setmetatable(object,self)
  return object
end
```

1. Create a memory diagram after execution of the code above together with:

   ```lua
   mystream = Stream:new({ mutex = Mutex:new() })
   ```

2. Extend the program by a `creator` function. This function should produce a `wrapper table` for tables, that were created with `Stream:new`. More specifically, this `wrapper table` should delegate every lookup to the wrapped table, with one exception: in case, the function `write` is called, the new table should establish a `Mutex`-locked area around a call to the wrapped table’s original `write` function.

Assignment 10.4 Prototype Based Design

Plan and implement the datastructures to represent symbolical arithmetical expressions, composed of the operators `+`, `−`, `·`, `/`, constants and variables in Lua. Don’t forget to include nice ways to specify and evaluate them!