

Programming Paradigms in a Procedural Language:

By Veer Singh

Implication of different programming paradigms into a procedural language is possible. Examples in this research paper are represented in Lua.

FUNCTIONAL PROGRAMMING IN LUA

Haskell is an example of a functional reactive programming language. Unlike procedural languages, it is based off a series of functions that keep a track of and represent specific up-values throughout time. Such functional functionality can be shown in a procedural language.

To do this, we can replicate a signal. By keeping custom objects running on their own constant loop, a function can be evaluated constantly on this loop. To do this, we can easily set a break value. In this case it will be shown in the 'Event' method. When 'Bool' is ever changed, the given loop will evaluate to false and stop. To obtain the specific values that may be present in the Signal, we can create a special table to do this and obtain all keys and values. In Lua, using ':' is syntactic sugar for allowing referencing 'self', or the table from which it was used. The main focus of this is on the new method. It expects some form of input to be a key/value table. By setting a metastable in Lua, the __index metamethod will be fired upon the attempted index of a nil value in an array. Thus it refers to the correlating functions. Additionally, the point of this signal is to be secure, so for this case, the __newindex metamethod is fired whenever a new index is attempted to be made. This is especially important as nothing should be able to edit the signal from outside.

```
local Signal = {
meta = {}
}
insert = table.insert
local SignalLibrary = {
Event = function (name)
return function(callPerStep)
    local Bool;      Bool = false;
    return { ["connect"] = function() Bool = true end,
        ["Wait"] = coroutine.wrap(function()
            local orig = Bool;
            while not rawequal(not orig,Bool) do
                callPerStep()
            end
            return true;
        end()), ["disconnect"] = function() Bool = false end, ["EventType"] =
name}
end
}
```

```

end
end,
}

function SignalLibrary:get(func,...)
    --return function(...)
        if func == nil then
            local kys,vls = {},{}
            for i,v in pairs(self) do
                insert(kys,i)
                insert(vls,v)
            end
            return self,kys,vls;
        else
            func(self)
        end
    --end
end

function Signal.new	dataType)
    return function(vals)
        return function(passIn)
            local kvAr = {}
            assert(#dataType==#vals,"Error: Need Key per Value")
            for ind,val in pairs(dataType) do
                kvAr[dataType[ind]] = vals[ind]
            end
            return setmetatable(kvAr, {
                __index = function(s,k)
                    if k=="get" then
                        return passIn.get or SignalLibrary.get
                    elseif k=="connect" then
                        return SignalLibrary["Event"]
                    elseif k == "set" then
                        end
                    end,
                    __newindex = function() return "No brute-indexing" end,
                })
            end
        end
    end

return Signal

```

OBJECT ORIENTED PROGRAMMING IN LUA

Java is a popular example of an Object Oriented Language. Additionally, C++ is also one. In Lua, there is no formal way to do Object Oriented Programming. However, it is represented by the paradigm

setmetatable(class, {__index=BaseClass})

Since the __index metamethod will fire whenever a nil value is indexed, it only makes sense to always refer to the base class to see if there is such methods available there. An example can be creating a database to hold information about individuals. We will take a look at creating a database in Lua to hold information separately for *Developers, Administration, and Regular Persons*. These will all be represented by modules that will eventually be required by a "main" script.

```
local insert
do
    local tableObj = table
    insert = tableObj.insert
end

--Required Inputs: Name, Id
--This is a generic Member / data should be used for main necessities and field
should be used for loose items

--[[ DOCUMENTATION:

Methods:

add_field_item (name)
edit_field_item (field,itemName,value) -> VOID;
remove_field_item (field,itemName,value) -> VOID;
add_data_field (key,value) -> VOID;
get_data_summary -> VOID;
get_data_field -> DataType/Generic;
get_data -> Array;
get_gender -> String;
get_id -> Int;
get_name -> String;
get_timezone -> String;
get_skype -> String;
get_registration -> Timestamp [Int]
add_data_item (dataName, value) -> VOID;
set_skype (skypeUser) -> String;
edit_data_field (key,value) -> value;
print_data_items -> VOID;
```

```

get_field_item (fieldName) -> (DataType/Generic)
get_field -> Array

--]]

local Member
do
    local baseObj = {
        fields = { },
        data = { },
        add_field_item = function(self, key, value )
            if self.fields[key] then
                return error("Exists")
            else
                self.fields[key] = value
                return self.fields[key]
            end
        end,
        edit_field_item = function(self, field, itemName, value)
            if self.fields[field] and self.fields[field][itemName] then
                self.fields[field][itemName] = value
                return self.fields[field][itemName]
            else
                return error("Something went wrong! Oops!")
            end
        end,
        remove_field_item = function(self, field, itemName, value)
            if self.fields[field] and self.fields[field][itemName] then
                self.fields[field][itemName] = nil
                return self.fields[field]
            else
                return error("Data field does not exist!")
            end
        end,
        add_data_field = function(self, key, value)
            if self.data[key] then
                return error("Data field already exists!")
            else
                self.data[key] = value
                return self.data[key]
            end
        end,
        get_data_summary = function(self)
            local dataStrings
            do
                local _tbl_0 = { }
                for k, v in pairs(self.data) do
                    _tbl_0[tostring(k) .. " : " .. tostring(v)] = ""
                end
                dataStrings = _tbl_0
            end
            return dataStrings
        end,
        get_data_field = function(self, fieldName)

```

```

    return self.data[fieldName]
end,
get_data = function(self)
    return self.data
end,
get_gender = function(self)
    return self.data["Gender"]
end,
get_id = function(self)
    return self.data["Id"]
end,
get_name = function(self)
    return self.data["Name"]
end,
get_timezone = function(self)
    return self.data["Timezone"]
end,
get_skype = function(self)
    return self.data["Skype"]
end,
get_registration = function(self)
    return self.data["register_Date"]
end,
add_data_item = function(self, dataName, value)
    if self.data[dataName] then
        return error("Data field already exists!")
    else
        self.data[dataName] = value
    end
end,
set_skype = function(self, skypeUser)
    if self.data["Skype"] then
        local prior = self.data["Skype"]
        self.data["Skype"] = skypeUser
        return prior, self.data["Skype"]
    else
        self.data["Skype"] = skypeUser
        return self.data["Skype"]
    end
end,
edit_data_field = function(self, key, value)
    if self.data[key] then
        self.data[key] = value
        return self.data[key]
    else
        return error("Field does not exist!")
    end
end,
print_data_items = function(self)
    for str in pairs(self:get_data_summary()) do
        print(str)
    end
end,
get_class_name = function(self)

```

```

        return self.__class["__name"]
    end,
    get_field_item = function(self, fieldName)
        return self.fields[fieldName]
    end,
    get_field = function(self)
        return self.fields
    end,
        get_field_summary = function(self)
        local fieldStrings
        do
            local _tbl_0 = { }
            for k, v in pairs(self.fields) do
                _tbl_0[tostring(k) .. " : " .. tostring(v)] = ""
            end
            fieldStrings = _tbl_0
        end
        return fieldStrings
    end,
    print_field_items = function(self)
        for str in pairs(self:get_field_summary()) do
            print(str)
        end
    end,
end,
}
baseObj.__index = baseObj
local bClass = setmetatable({
    __init = function(self, Name, Id)
        self.data = {
            Name = Name,
            Id = Id,
            register_Date = os.time()
        }
    end,
    __base = baseObj,
    __name = "Member"
}, {
    __index = baseObj,
    __call = function(cls, ...)
        local selfClass = setmetatable({}, baseObj)
        cls.__init(selfClass, ...)
        return selfClass
    end
})
baseObj.__class = bClass
Member = bClass
end
return Member

```

If this looks any bit complex, take a moment to realize that the majority of this is simple methods for the sake of indexing and inserting values into the table. The main area of interest is the index and metastable declarations. First off, in the

'constructor' represented by '`__init`', values that can be used throughout the object are represented. First we keep an array, 'data', which serves as the base for some of the stated methods. This array keeps simple values such as Name, Id, and registration date. Additionally, our mentioned above paradigm can be seen with `__call`. Setmetatable also returns an array, so keeping merely {} as the first parameter will attach the stated metatable and returned first given array. Additionally, we call the initiation (our fake constructor) to handle any tuples (lists of extra inputted data) and handle specifying what the base Class is. Now, this is just an example of the class itself. Currently, there is no interaction with this class. Using this method, we can create similar classes for *Administration* and *Developers*.

Now, since *Administration* members are also *regular members* in the case that they do also have names and time zones (otherwise I would be worried), this means *Administration* can take its methods from *Member*.

```

local Administrator;
local Member = require(Member.lua)

--Required Inputs: Name, Id[int], canPromoteDemoteFromDatabase [bool]
--This is a generic HighRank which inherits from Member | data should be used for main necessities and field should be used for loose items

--[[DOCUMENTATION

Methods:

print_administrator_changes -> VOID
get_rank_bool -> boolean
set_Rank (groupID) -> VOID

--]]

do
local canRankFromDB;
local parentClass = Member
local baseClass = {
    set_Rank = function(self, iD, rank)
        self.data["currentRank"] = rank;
    end,
    get_rank_bool = function(self)
        return self.data[canRankFromDB]
    end,
    print_Member_changes = function(self)
        for plr, change in pairs(self.data) do
            print(plr .. " was " .. change[1] .. "d to " .. change[2])
        end
    end
}
--change table should look like: {"Promote" or "Demote" or "Fire", "rankName"}

```

```

    end
}
baseClass.__index = baseClass
setmetatable(baseClass, parentClass.__base)
local bClassObj = setmetatable({
    __init = function(self, Name, Id, canRankFromDB)
        self.data = {
            Name = Name,
            Id = Id,
            register_Date = os.time(),
            canRankFromDB = canRankFromDB
        }
        self.data["changes"] = { }
    end,
    __base = baseClass,
    __name = "Administrator",
    __parent = parentClass
}, {
    __index = function(cls, name)
        local val = rawget(baseClass, name)
        if val == nil then
            return parentClass[name]
        else
            return val
        end
    end,
    __call = function(cls, ...)
        local sClassObj = setmetatable({}, baseClass)
        cls.__init(sClassObj, ...)
        return sClassObj
    end
})
baseClass.__class = bClassObj
Administrator = bClassObj
return Administrator
end
return Administrator;

```

Take a look at the `__index` metamethod usage inside 'bClassObj', which refers to the parent-class whenever a nil value is set. This is in short-terms, Object Oriented Programming in Lua. Finishing the `Developer` class now:

```
Member = require(Member.lua)
```

```

local Developer
do
    local parent = Member
    local baseClass = {

```

```

fields = { },
data = { },
get_dev_type = function(self)
    return self.data["DevType"]
end
}
baseClass.__index = baseClass
setmetatable(baseClass, parent.__base)
local bClassObj = setmetatable({
    __init = function(self, Name, Id, DevType)
        self.data = {
            Name = Name,
            Id = Id,
            DevType = DevType,
            register_Date = os.time()
        }
    end,
    __base = baseClass,
    __name = "Developer",
    __parent = parent
}, {
    __index = function(cls, name)
        local val = rawget(baseClass, name)
        if val == nil then
            return parent[name]
        else
            return val
        end
    end,
    __call = function(cls, ...)
        local selfClass = setmetatable({}, baseClass)
        cls.__init(selfClass, ...)
        return selfClass
    end
})
baseClass.__class = bClassObj
Developer = class
return class
end

```

Using the same type of inheritance shown in the *Administrator* class, this is how a basic, method-less, *Developer* class is shown. This is all done with the paradigm, *setmetatable(class, {__index=baseClass})*

TRANSPOSING AND PRINTING MATRICES IN LUA

Lua's foundations come straight from tables itself. However, surprisingly, there is no built-in support for two-dimensional arrays. Therefore, one must be replicated itself.

A custom 2D array can be created by simply using nested arrays such as: `{ {}, {} }` which represents a two-dimensional array. The important equation in this is $n*(row-1) + col$; This represents the proper equation for obtaining the correct index in the 1D array *contents*. In the transpose method, a new array is created by switching the key-values in the tables (as that is what transposing is)

```

local Matrix = {}

function Matrix.new(m, n, contents)
    assert(m*n == #contents)
    local ourMatrix = { }
    function ourMatrix:transpose()
        local twoDTransform = {}
        for row = 1,m do
            local temp = {}
            for col = 1,n do
                local currentIndex = n*(row-1) + col;
                table.insert(temp,contents[currentIndex])
            end
            table.insert(twoDTransform,temp)
            temp = {}
        end
        local function transpose(inputMatrix)
            local newMatrix = {}
            for i = 1, #inputMatrix[1] do
                newMatrix[i] = {}
                for v = 1, #inputMatrix do
                    newMatrix[i][v] = inputMatrix[v][i]
                end
            end
            return newMatrix
        end
        local transposed = transpose(twoDTransform)
        local final = {}
        for _,v in pairs(transposed) do
            for i,k in pairs(v) do
                table.insert(final,k)
            end
        end
        contents = final;
        local temp = m;
        m = n;
        n = temp;
    end

    function ourMatrix:print()
        local str = "";
        for row = 1,m do
            str = str.."["
            for col = 1, n do
                local currentIndex = n*(row-1) + col
                str = str .. (col ~= n and
contents[currentIndex] .. " " or contents[currentIndex].."")
            end
            str = str .. "]"
        end
        print(str)
    end
end

```

```

        end
        str = str.."]"
        str=str.."\n"
    end
    print(str)
end
return ourMatrix
end

```

TOSTRING FOR TABLES IN LUA

In Lua, calling `toString` or `Print` on a table will result in the tables *pointer* or memory address. By iterating over values in the table and checking the tables metatable, it is easy to output these tables into a more readable format:

```

function tableToString(t)
local function handleDataTypes(v)
    local datatype = type(v);
    if datatype=="table" then
        return "{" .. tableToString(v) .. "}"
    elseif datatype=="userdata" or datatype=="function" then
        return datatype;
    elseif datatype=="string" then
        return ("").format("%q",v)
    else
        return tostring(v);
    end
end
local str = "{"
local sep = ", "
for i, v in pairs(t) do
    if not tonumber(i) then
        str = str .. sep .. tostring(i) .. " = " .. handleDataTypes(v);
    end
end
for i, v in ipairs(t) do
    str = str .. sep .. handleDataTypes(v);
end
local final = "...str:sub(#sep + 2) .. "}";
if getmetatable(t) ~= nil then
    final = final .. " <- " .. tableToString(getmetatable(t))
end
return final
end

```

REAL-WORLD APPLICATION: CLEAN SKYPE LOGS

Skype logs, when copied, can be very messy with all the extremely sloppy data-stamps put. Sometimes the sender and message are all that really matters. Although Lua does not use *POSIX Regex*, it has its own set of string patterns that can still be used.

```
local dates = [[.%d+:%d+:%d+ .M. ]]
local nameCap = "(.+):"
local msgCap = [[:(.*)]]
local singleLineComplement = "[^\n]+"

function readLineAndFormat(txt)
local str = txt
local subbedMessage = txt:gsub(dates,"")
local finalStr = "";
for line in subbedMessage:gmatch(singleLine) do
    local name = line:match(nameCap)
    local message = line:match(msgCap)
    finalStr = finalStr .. (name~=nil and name .. " | " or "") ..
(message~=nil and tostring(message) .. string.char(10) or "")
end
return finalStr;
end
```

. = any character, %d = matches a digit, [^\n] = complement, in Lua, a complement is a *match anything but this*. Since \n is the escape for a new line, Lua matches anything that isn't a new line, thus matching everything in a specific line. Using the '+' matches *1 or more* of the given capture group while '*' matches *0 or more*.

USING LUA IN GAME DEVELOPMENT: THE ROBLOX GAME ENGINE CUSTOM CAMERA

Lua is often found in many game engines. One of the more popular ones is the Roblox game engine. Roblox uses primarily Lua 5.1, and also comes with its own set of basic built-in methods. To replicate the famous custom camera seen in many video games, it requires a bit of knowledge about Euclidian Space and some of the built-in methods:

```
local player = game.Players.LocalPlayer;
local character = player.Character or player.CharacterAdded:wait()

local currentArm = character:FindFirstChild("Right Arm");
workspace.CurrentCamera:Destroy()
wait()
local camera = workspace.CurrentCamera;
```

```

camera.CameraType = "Scriptable"
camera.FieldOfView = 80;
local mouse = player:GetMouse();

local camPart = Instance.new("Part", camera);
camPart.FormFactor = "Custom"
camPart.Size = Vector3.new(.2,.2,.2)
camPart.Anchored = true;
camPart.CanCollide = false;
camPart.Name = "CameraPart";
--camPart.Shape = "Ball";
camPart.TopSurface = "Smooth";
camPart.BottomSurface = "Smooth";
camPart.BrickColor = BrickColor.new("White");
camPart.Transparency = .8;

local centerPart = character.HumanoidRootPart;

local motivator = Instance.new("BodyPosition", camPart);
motivator.D = motivator.D / 2;
local angle = 0;
local anglex = math.deg({centerPart.CFrame:toEulerAnglesXYZ()}[1]);

local lastpos = Vector2.new(mouse.ViewSizeX/2, mouse.ViewSizeY/2);

local position = CFrame.new(centerPart.Position, centerPart.Position +
centerPart.CFrame.lookVector) * CFrame.new((centerPart.Size.X/2) +
currentArm.Size.X, centerPart.Size.Y/2, 0)
local pos = CFrame.new();

position = centerPart.CFrame:toObjectSpace(position);
print(position);

character.Humanoid.AutoRotate = false;
local characterMover = Instance.new("BodyGyro", character.HumanoidRootPart);
characterMover.maxTorque = Vector3.new(0, 400000, 0);

function updatePart()
    local anglexPadder = anglex;
    motivator.position = ((centerPart.CFrame * position) + (
        (centerPart.CFrame * position * CFrame.Angles(math.rad(angle),0,
0)).lookVector * 10)).p
    characterMover.cframe = CFrame.new(centerPart.Position) * CFrame.Angles(0,
math.rad(-anglexPadder), 0);
end
camPart.Anchored = false;
camPart:BreakJoints();

local deltax = 0;
local deltay = 0;

function mouseMoveDeltaHandler(deltax, deltay)
    if deltax~=0 then
        local deltaxSign = deltax / math.abs(deltax);

```

```

        local deltaxMove = (math.abs(deltax)>=5 and 5 or 1);
        anglex = ((anglex + ((deltaxMove * deltaxSign)))) % 360;
    else
    end

    if deltay~=0 then
        local deltaySign = deltay / math.abs(deltay);
        local deltayMove = (math.abs(deltay)>=5 and -5 or -1);
        angle = math.min(math.max(angle + (deltayMove * deltaySign)), -20),
30);
    else
    end;

    updatePart();
end

game:GetService("UserInputService").MouseIconEnabled = false
game:GetService("UserInputService").MouseBehavior = Enum.MouseBehavior.LockCenter;
game:GetService("UserInputService").InputChanged:connect(function(inputObject)
    if inputObject.UserInputType == Enum.UserInputType.MouseMovement then
        mouseMoveDeltaHandler(inputObject.Delta.x, inputObject.Delta.y);
        print("delta is (" .. tostring(inputObject.Delta.x) .. ", " ..
tostring(inputObject.Delta.y) .. ")")
    end
end)

game:GetService("RunService").RenderStepped:connect(function()
    updatePart();
    pos = centerPart.CFrame * position;
    local lookCFrame = CFrame.new(motivator.position, pos.p)
--    pos = pos + (lookCFrame.LookVector * 7);
--    pos = pos + Vector3.new(3, 4, 4);
    camera.CoordinateFrame = CFrame.new(pos.p, camPart.Position);
end)

local b2Down = false;

mouse.Button2Down:connect(function()
    b2Down = true;
    for i = camera.FieldOfView, 50, -10 do
        if not b2Down then break end;
        camera.FieldOfView = i
        wait()
    end
end)

mouse.Button2Up:connect(function()
    b2Down = false;
    for i = camera.FieldOfView, 80, 10 do
        if b2Down then break end;
        camera.FieldOfView = i;
        wait();
    end

```

```
end)
```

By handling the mouseMovement along with taking the current Torso position of the Player in the Roblox Game Engine, a custom camera is very easy to replicate.

DETERMINE SHORTEST ANGULAR OFFSET IN LUA

Suppose you want to determine if its quicker to get to an angle by clockwise or counter-clockwise rotation:

```
function angularOffset(startAngle, goalAngle)
local rot1 = math.abs((math.floor(math.deg(startAngle)) -
(math.floor(math.floor(math.deg(startAngle))/360) * 360)) -
(math.floor(math.deg(goalAngle)) -
(math.floor(math.floor(math.deg(goalAngle))/360) * 360)))
local rot2 = 360 - rot1
return ((rot1 < rot2 and 1) or (rot2 < rot1 and -1) or (rot2==rot1 and 0))
end
```

This will simply return '1' if going clockwise is shorter, otherwise -1 if going counterclockwise is shorter. 0 is returned if both distances are equal. This is a sloppy way to do this without using modulus. To use modulus would be much easier:

```
local pi = math.pi
local tau = 2*pi

function angularOffsetSign(startAngle, goalAngle)
    startAngle = startAngle%tau
    goalAngle = goalAngle%tau
    local same = startAngle == goalAngle
    local fsame = (startAngle + pi)%tau == goalAngle
    local result = (startAngle - goalAngle)%tau
    return result < pi and -1 or ((fsame or same) and 0) or 1
end
```

The following examples are unique cases to go about handling different types of problems in a procedural language such as Lua. Lua is a very light-weight language derived from ANSI C and has a very clean syntax. Lua is processed into what is called "Lua Bytecode". Understanding such a topic can allow one to create their very own language in Lua.

CREATING A PROGRAMMING LANGUAGE FROM WITHIN LUA

With an understanding of how memory and bytes work, one can create his or her own language. I will be calling mine, SIL, for "Scripted Interpreted Language"

Making an assembler is fairly straight-forward. There will be methods for parsing, removing identifier (signed integers), and handling hexadecimal along with decimal integers. A lot goes into it such as checking for endianness, verifying headers, assembling;interpreting; and disassembling. The current language will only handle a certain amount of operations. Integers, Hexadecimal, Characters, and Comments will be handled. The following is a research proposal on how to create a language using Lua.

```
local header = "SIL\1\0";
local lexerState, parserState;
local param = {off = 0, cst = 1, offcst = 2};
local instructionSet = {
["move"] = {0, 2, param.off, param.offcst},      -- MOVE <OFF> <OFFCST>
["in"] = {1, 1, param.off},
["out"] = {2, 1, param.offcst},
["add"] = {3, 3, param.off, param.offcst, param.offcst}, -- ADD
["sub"] = {4, 3, param.off, param.offcst, param.offcst}, -- SUB
["mul"] = {5, 3, param.off, param.offcst, param.offcst}, -- MUL
["div"] = {6, 3, param.off, param.offcst, param.offcst}, -- DIV
["mod"] = {7, 3, param.off, param.offcst, param.offcst}, -- MOD
["pow"] = {8, 3, param.off, param.offcst, param.offcst} -- POW
};

--- LEXER LIBRARY ---
local tokenType = {whitespace = 0, integer = 1, identifier = 2, comment = 3,
section = 4};
local sections = {header = 0, consts = 1, code = 2};

do
local lexerLibrary = {};
local lexerStateMetatable = {__index = lexerLibrary};
lexerLibrary.getChar = function(state)
    state.idx = state.idx + 1;
if state.idx <= #state.program then
    state.peek = state.program:sub(state.idx + 1, state.idx + 1);
        return state.program:sub(state.idx, state.idx);
else
end
```

```

        return false, "EOF unexpected";
    end
end;

lexerLibrary.consumeWhitespace = function(state)
while state.peek == " " or state.peek == "\t" or state.peek == "\n" or state.peek
== "\r" do
    state:getChar();
end
if state.peek == "" then return true; end
end;
lexerLibrary.getToken = function(state)
    local token = "";
    local tokType = 0;
    if state:consumeWhitespace() then return true; end
if state.peek:match(";") then -- COMMENT
    tokType = tokenType.comment;
    while state.peek ~= "" and state.peek ~= "\n" do
        state:getChar();
    end
elseif state.peek:match("%d") then -- INTEGER
    tokType = tokenType.integer;
    while state.peek:match("%d") do
        token = token .. state:getChar();
    end
if state.peek == "h" then -- HEXADECIMAL
    token = tonumber(token, 16);
    state:getChar();
elseif state.peek == "b" then
    token = tonumber(token, 2);
    state:getChar();
else
    token = math.floor(tonumber(token));
end
elseif state.peek:match("[%a_]") then -- IDENTIFIER
    tokType = tokenType.identifier;
    while state.peek:match("[%w_]") do
        token = token .. state:getChar();
    end
elseif state.peek == "." then -- SECTION
    tokType = tokenType.section;
    state:getChar();
    if state.peek:match("[%a_}") then
        while state.peek:match("[%w_}") do
            token = token .. state:getChar();
        end
    else
        token = "";
    end
if #token == 0 then return false, "Identifier expected"; end
elseif state.peek == "\n" or state.peek == "\r" then
state:getChar();
end
return token, tokType;

```

```

end;
lexerLibrary.tokenify = function(state)
    local tokens = {};
    while state.idx < #state.program do
        local token, tokenType = state:getToken();
        if token and token ~= true then
            tokens[#tokens + 1] = {token, tokenType};
        elseif token ~= true then
            return false, tokenType;
        end
    end
    return tokens;
end;
lexerState = function(program)
    return setmetatable({
        program = program;
        idx = 0;
        peek = program:sub(1, 1);
    }, lexerStateMetatable);
end;
end

--- PARSER LIBRARY ---
do
local parserLibrary = {};
local parserStateMetatable = {__index = parserLibrary};
parserLibrary.getToken = function(state)
    state.idx = state.idx + 1;
    if state.idx <= #state.program then
        state.peek = state.program[state.idx + 1];
        return state.program[state.idx];
    else
        return false, "End of token list";
    end
end;
parserLibrary.emitIdentifier = function(state, identifier)
for strIdx = 1, #identifier do
    state.bytecode[#state.bytecode + 1] = identifier:sub(strIdx, strIdx):byte();
end
end;
parserLibrary.emitByte = function(state, byte)
    state.bytecode[#state.bytecode + 1] = byte % 256;
end;
parserLibrary.emitInteger = function(state, littleEndian, integer, signed)
    if integer < 0 and signed then
        integer = 4294967295 - math.abs(integer);
    end
local byte1, byte2, byte3, byte4 = math.floor(integer / (2 ^ 24)),
math.floor(integer / (2 ^ 16)) % 256, math.floor(integer / (2 ^ 8)) % 256,
integer % 256;
state:emitByte(littleEndian and byte4 or byte1);
state:emitByte(littleEndian and byte3 or byte2);
state:emitByte(littleEndian and byte2 or byte3);
state:emitByte(littleEndian and byte1 or byte4);

```

```

end;
parserLibrary.emitInstruction = function(state, littleEndian, opcode, paramA,
paramB, paramC)
paramA = paramA or 0;
paramB = paramB or 0;
paramC = paramC or 0;
state:emitByte(littleEndian and paramC or opcode);
state:emitByte(littleEndian and paramB or paramA);
state:emitByte(littleEndian and paramA or paramB);
state:emitByte(littleEndian and opcode or paramC);
end;
parserLibrary.parse = function(state)
while state.idx < #state.program do
local token, reason = state:getToken();
if not token then return false, reason end
if token[2] == tokenType.section then -- section : '.' sectionType
state.section = sections[token[1]];
if state.section == sections.header then-- header : signature version endianness
local signature = state:getToken();
if signature and signature[2] == tokenType.identifier then-- signature
    state:emitIdentifier(signature[1]);
    local version = state:getToken();
    if version and version[2] == tokenType.integer then -- version
        state:emitByte(version[1]);
        local endianness = state:getToken();
        if endianness and endianness[2] == tokenType.integer then-- endianness
            state:emitByte(endianness[1]);
            state.le = endianness[1] == 1;
        else
            return false, "Bad header, expected endianness (byte)";
        end
    else
        return false, "Bad header, expected version (byte)";
    end
else
    return false, "Bad header, expected signature (identifier)";
end
elseif state.section == sections.consts then -- consts : constsBodyList;
local constCount = 0;
local consts = {};
while state.peek[2] == tokenType.integer do
local constant, reason = state:getToken();
if constant and constant[2] == tokenType.integer then
constCount = constCount + 1;
consts[constCount] = constant[1];
else
return false, "Expected integer";
end
end
if constCount > 128 then return false, "Too many constants"; end
state:emitByte(constCount);
for constIdx = 1, #consts do
    state:emitInteger(state.le, consts[constIdx], true);
end

```

```

elseif state.section == sections.code then      -- code : instructionsBodyList;
local instructionCount = 0;
local instructions = {};
while true do
local mnemonic, reason = state:getToken();
if mnemonic == false then break; end
if mnemonic and mnemonic[2] == tokenType.identifier then
local data = instructionSet[mnemonic[1]:lower()];
    if data then
        local parameters = {};
        paramIdx = 1, data[2] do
            local param, reason = state:getToken();
            if param and param[2] == tokenType.integer then
                parameters[#parameters + 1] = param[1];
            else
                return false, "Expected byte";
            end
        end
        instructionCount = instructionCount + 1;
        instructions[instructionCount] = {data[1], unpack(parameters)};
    else
        return false, "Invalid instruction";
    end
else
return false, "Expected instruction";
end
end
state:emitByte(instructionCount);
for instructionIdx = 1, instructionCount do
state:emitInstruction(state.le, unpack(instructions[instructionIdx]));
end
else
return false, "Invalid section";
end
else
return false, "Expected section";
end
end
return state.bytecode;
end;

parserState = function(program)
return setmetatable({
    program = program;
    idx = 0;
    section = -1;
    le = false;
    bytecode = {};
}, parserStateMetatable);
end;
end

-- TESTING CLASS --
local program = [

```

```

.header
SIL
1
0

.consts
666
3

.code
MOVE 0 128
MUL 0 0 129
OUT 0
[]]

-- Assembles into: \83\73\76\1\0\2\0\0\2\154\0\0\0\3\3\0\0\128\0\5\0\0\129\2\0\0\0
--

local lexed, reason = lexerState(program):tokenify();
if not lexed then return print("ERROR LEXING", reason);end
print("LEXED");
for key = 1, #lexed do
    print(lexed[key][1], lexed[key][2]);
end

local parsed, reason = parserState(lexed):parse();
if not parsed then return print("ERROR PARSING", reason); end
print("Parsing Successful");
for key = 1, #parsed do io.write(("\\%d"):format(parsed[key]));
end

```

CREATING A DISASSEMBLER FOR SIL IN LUA:

Starting at this part, a disassembler will now be created. The interpreter and the disassembler are similar to a very loose extent. However, both are completely different from the assembler.

```

local parseState;
local bytecodeVersion = 1;
--- PARSER LIBRARY START ---
do
local parseLibrary = {};
local parseStateMetatable = {__index = parseLibrary};
parseLibrary.getByte = function(state)
    state.idx = state.idx + 1;
    if state.idx <= #state.program then
        return state.program:sub(state.idx, state.idx):byte();
    else
        return false, "EOF unexpected";
    end
end;

```

```

parseLibrary.getBytes = function(state, count)
    local bytes = {};
    for n = 1, count do
        local nextByte, reason = state:getByte();
        if nextByte then
            bytes[n] = nextByte;
        else
            return false, reason;
        end
    end
    return bytes;
end;

parseLibrary.readInteger = function(state, littleEndian, signed)
    local bytes, reason = state:getBytes(4);
    if bytes then
        if littleEndian then
            bytes[1], bytes[2], bytes[3], bytes[4] =
                bytes[4], bytes[3], bytes[2], bytes[1];
        end

        local offset = (signed and bytes[1] > 127) and (-2 ^ 32) or 0;
        return offset +
            (bytes[1] * (2 ^ 24)) +
            (bytes[2] * (2 ^ 16)) +
            (bytes[3] * (2 ^ 8)) +
            (bytes[4]);
    else
        return false, reason;
    end
end;

parseLibrary.parseInstruction = function(state, littleEndian, instruction, reason)
    if instruction and #instruction == 4 then
        if littleEndian then
            instruction[1],instruction[2],instruction[3],instruction[4]=
                instruction[4], instruction[3], instruction[2], instruction[1];
        end
        local instruction = {
            opcode = instruction[1],
            paramA = instruction[2],
            paramB = instruction[3],
            paramC = instruction[4]
        };
        if instruction.opcode <= 8 then
            return instruction;
        else
            return false, "Invalid instruction at instruction " .. state.pc;
        end
    else
        return false, reason;
    end
end;

```

```

parseLibrary.parseInt = function(state, double)
    local flr = math.floor(double);
    if flr > (2 ^ 31) - 1 then
        flr = (2 ^ 31) - 1;
    elseif flr < -(2 ^ 31) then
        flr = -(2 ^ 31);
    end
    return flr;
end;

parseState = function(program)
    return setmetatable({
        program = program,
        idx = 0,
    }, parseStateMetatable);
end;
end

local disassemble = function(program)
local state = parseState(program);
state.pc = 0; -- program counter
state.le = false; -- little endian
local disassembled = "";
--- VALIDATE AND DISASSEMBLE HEADER ---
do
    local signature, reason = state:getBytes(3);
    local version = state:getByte();
    local endianness = state:getByte();
if not (signature and version and endianness) then return false, reason;
elseif string.char(unpack(signature)) ~= "SIL" then return false, "Invalid signature";
elseif version ~= bytecodeVersion then return false, "Unsupported version";
elseif not (endianness == 0 or endianness == 1) then return false, "Unsupported endianness";
end
state.le = endianness == 1;
disassembled = disassembled .. string.format(".header ; 5 bytes\nSIL ;
signature\n0x%X ; version\n0x%X ; endianness (0 - big, 1 - little)\n\n", version,
endianness);
end
local st = state.st;

--- CONSTANTS ---
local constants = {};
local constCount = state:getByte();
if not constCount then return false, reason;
elseif constCount > 128 then return false, "Too many constants";
end
for constIdx = 0, constCount - 1 do
    local integer, reason = state:readInteger(state.le, true);
    if integer then
        constants[2 * constIdx + 1] = integer;
        constants[2 * constIdx + 2] = constIdx;
    end
end

```

```

        else
            return false, reason;
        end
    end
disassembled = disassembled .. string.format(".consts ; %d constants\n" .. ("%d ;
constant %d\n"):rep(constCount) .. "\n", constCount, unpack(constants));
--- INSTRUCTIONS ---
local instructions = {};
local instructionCount = state:getByte();
if not instructionCount then return false, reason; end
while state.pc < instructionCount do
local instruction, err = state:parseInstruction(state.le, state:getBytes(4));
    if instruction then
        if instruction.opcode == 0 then                                -- MOVE <off> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "MOVE " .. instruction.paramA .. " " ..
instruction.paramB;
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        elseif instruction.opcode == 1 then                            -- IN <off>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "IN " .. instruction.paramA;
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        elseif instruction.opcode == 2 then                            -- OUT <offcst>
            instructions[state.pc + 1] = "OUT " .. instruction.paramA;
        elseif instruction.opcode == 3 then                            -- ADD <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "ADD " .. instruction.paramA .. " " ..
instruction.paramB .. " " .. instruction.paramC;
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        elseif instruction.opcode == 4 then                            -- SUB <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "SUB " .. instruction.paramA .. " " ..
.. instruction.paramB .. " " .. instruction.paramC;
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        elseif instruction.opcode == 5 then                            -- MUL <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "MUL " .. instruction.paramA .. " " ..
instruction.paramB .. " " .. instruction.paramC;
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        end
    end
end

```

```

        elseif instruction.opcode == 6 then      -- DIV <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "DIV " .. instruction.paramA .. "
" .. instruction.paramB .. " " .. instruction.paramC;
            else
        return false, "Register expected as parameter A at instruction " .. state.pc;
        end
        elseif instruction.opcode == 7 then      -- MOD <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
instructions[state.pc + 1] = "MOD " .. instruction.paramA .. " " ..
instruction.paramB .. " " .. instruction.paramC;
            else
        return false, "Register expected as parameter A at instruction " .. state.pc;
        end
        elseif instruction.opcode == 8 then      -- POW <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                instructions[state.pc + 1] = "POW " .. instruction.paramA .. "
" .. instruction.paramB .. " " .. instruction.paramC;
            else
                return false, "Register expected as parameter A at instruction
" .. state.pc;
            end
        end
        else
            return false, err;
        end
        state.pc = state.pc + 1;
    end
disassembled = disassembled .. ("").format(".code ; %d instructions\n" ..
("%s\n"):rep(state.pc), state.pc, unpack(instructions));
    return disassembled;
end;

print(disassemble("\x83\x73\x76\x1\x0\x4\x0\x0\x0\x5\x0\x0\x0\x10\x0\x0\x0\x15\x0\x0\x0\x20\x4\x2\x128\x0\x0\x2\x129\x0\x0\x3\x0\x128\x129\x2\x0\x0\x0"));

return disassemble;

```

CREATING AN INTERPRETER FOR SIL IN LUA

This is the final interpreter for SIL. It is similar to the disassembler to a very loose extent. As with all the prior code-examples, sample code is given at the bottom to show an example of how this can be ran. This, like the other assembler and disassembler, are modules.

```

local parseState;
local bytecodeVersion = 1;
--- PARSER LIBRARY START ---

```

```

do
local parseLibrary = {};
local parseStateMetatable = {__index = parseLibrary};

parseLibrary.getByte = function(state)
    state.idx = state.idx + 1;
    if state.idx <= #state.program then
        return state.program:sub(state.idx, state.idx):byte();
    else
        return false, "EOF unexpected";
    end
end;
parseLibrary.getBytes = function(state, count)
    local bytes = {};
    for n = 1, count do
        local nextByte, reason = state:getByte();
        if nextByte then
            bytes[n] = nextByte;
        else
            return false, reason;
        end
    end
    return bytes;
end;

parseLibrary.readInteger = function(state, littleEndian, signed)
local bytes, reason = state:getBytes(4);
if bytes then
    if littleEndian then
        bytes[1], bytes[2], bytes[3], bytes[4] =
        bytes[4], bytes[3], bytes[2], bytes[1];
    end

    local offset = (signed and bytes[1] > 127) and (-2 ^ 32) or 0;
    return offset + (bytes[1] * (2 ^ 24)) +
           (bytes[2] * (2 ^ 16)) +
           (bytes[3] * (2 ^ 8)) +
           (bytes[4]);
    else
        return false, reason;
    end
end;
parseLibrary.parseInstruction = function(state, littleEndian, instruction, reason)
if instruction and #instruction == 4 then
if littleEndian then
instruction[1], instruction[2], instruction[3], instruction[4] =
    instruction[4], instruction[3], instruction[2], instruction[1];
end

local instruction = {
    opcode = instruction[1],
    paramA = instruction[2],
    paramB = instruction[3],
    paramC = instruction[4]
};

```

```

if instruction.opcode <= 8 then
    return instruction;
else
    return false, "Invalid instruction at instruction " .. state.pc;
end
else
    return false, reason;
end
end;
parseLibrary.parseInt = function(state, double)
local flr = math.floor(double);
if flr > (2 ^ 31) - 1 then
    flr = (2 ^ 31) - 1;
elseif flr < -(2 ^ 31) then
    flr = -(2 ^ 31);
end
return flr;
end;
parseState = function(program)
    return setmetatable({
        program = program,
        idx = 0;
    }, parseStateMetatable);
end;
end

local interpret = function(program)
local state = parseState(program);
state.pc = 0;           -- program counter
state.le = false;       -- little endian
state.st = {};          -- program stack
do
    local signature, reason = state:getBytes(3);
    local version = state:getByte();
local endianness = state:getByte();
if not (signature and version and endianness) then return false, reason;elseif
string.char(unpack(signature)) ~= "SIL" then return false, "Invalid
signature";elseif version ~= bytecodeVersion then return false, "Unsupported
version";
elseif not (endianness == 0 or endianness == 1) then return false, "Unsupported
endianness";
    end
    state.le = endianness == 1;
end

local st = state.st;

--- INITIALIZE Stack ---
for reg = 0, 255 do
    st[reg] = 0;
end

--- STORE CONSTANTS ---

```

```

local constCount, reason = state:getByte();
if not constCount then return false, reason;
elseif constCount > 128 then return false, "Too many constants";
end
for constIdx = 0, constCount - 1 do
    local integer, reason = state:readInteger(state.le, true);
    if integer then
        st[constIdx + 128] = integer;
    else
        return false, reason;
    end
end

--- INTERPRET INSTRUCTIONS ---
local instructionCount, reason = state:getByte();
if not instructionCount then return false, reason; end
while state.pc < instructionCount do
local instruction, err = state:parseInstruction(state.le, state:getBytes(4));
if instruction then
    if instruction.opcode == 0 then          -- MOVE <off> <offcst>
        if instruction.paramA <= 127 then
            st[instruction.paramA] = st[instruction.paramB];
        else
            return false, "Register expected as parameter A at instruction
" .. state.pc;
        end
    elseif instruction.opcode == 1 then        -- IN <off>
        if instruction.paramA <= 127 then
            while true do
                local input = io.read("*n");
                if input then
                    st[instruction.paramA] = state:parseInt(input);
                end
            end
        else
            return false, "Register expected as parameter A at instruction " .. state.pc;
        end
    elseif instruction.opcode == 2 then      -- OUT <offcst>
        print(st[instruction.paramA]);
    elseif instruction.opcode == 3 then      -- ADD <off> <offcst> <offcst>
        if instruction.paramA <= 127 then
            st[instruction.paramA] = state:parseInt(st[instruction.paramB] +
st[instruction.paramC]);
        else
            return false, "Register expected as parameter A at instruction
" .. state.pc;
        end
    elseif instruction.opcode == 4 then-- SUB <off> <offcst> <offcst>
        if instruction.paramA <= 127 then
            st[instruction.paramA] = state:parseInt(st[instruction.paramB] -
st[instruction.paramC]);
        else
            return false, "Register expected as parameter A at instruction
" .. state.pc;
    end
end

```

```

        end
        elseif instruction.opcode == 5 then-- MUL <off> <offcst> <offcst>
            if instruction.paramA <= 127 then st[instruction.paramA] =
state:parseInt(st[instruction.paramB] * st[instruction.paramC]);
            else
                return false, "Register expected as parameter A at instruction " ..
state.pc;
            end
        elseif instruction.opcode == 6 then      -- DIV <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                st[instruction.paramA] = state:parseInt(st[instruction.paramB] /
st[instruction.paramC]);
            else
                return false, "Register expected as parameter A at instruction
" .. state.pc;
            end
        elseif instruction.opcode == 7 then-- MOD <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                st[instruction.paramA] = state:parseInt(st[instruction.paramB] %
st[instruction.paramC]);
            else
                return false, "Register expected as parameter A at instruction " .. state.pc;
            end
        elseif instruction.opcode == 8 then      -- POW <off> <offcst> <offcst>
            if instruction.paramA <= 127 then
                st[instruction.paramA] = state:parseInt(st[instruction.paramB] ^
st[instruction.paramC]);
            else
                return false, "Register expected as parameter A at
instruction " .. state.pc;
            end
        else
            return false, err;
        end
        state.pc = state.pc + 1;
    end
    return state;
end;

interpret("\x83\x73\x76\x1\x0\x4\x0\x0\x0\x5\x0\x0\x10\x0\x0\x0\x15\x0\x0\x20\x4\x2\x128\x0\x0\x2\x129\x0\x
0\x3\x0\x128\x129\x2\x0\x0\x0");
return interpret;

```

All the following are unique ways to implement different techniques into a procedural language such as Lua. The more advanced topics covered are the programming paradigms (OOP, and Functional) along with comprehending and understanding how to make a language in Lua. To understand functional and object oriented languages, taking a shot at a procedural language isn't a bad option. Additionally, no matter where you work as an engineer, you will somehow, somewhere, end up stumbling across a procedural language and it is important to

understand how to grasp the concepts behind it. This is my research take upon different paradigms in a procedural language.

-Veer Singh