Switch Statement

The problem

Lua lacks a C-style switch[1] statement. This issue has come up a number of times on the mailing list. There are ways to emulate the same effect as discussed here.

The first question to ask is why we might want a switch statement rather than a comparison chain as such:

```
local is_canadian = false
function sayit(letters)
  for _,v in ipairs(letters) do
    if     v == "a" then print("aah")
    elseif v == "b" then print("bee")
    elseif v == "c" then print("see")
    elseif v == "d" then print("dee")
    elseif v == "e" then print("eee")
    elseif v == "f" then print("eff")
    elseif v == "g" then print("gee")
    elseif v == "h" then print("aych")
    elseif v == "i" then print("eye")
    elseif v == "j" then print("jay")
    elseif v == "k" then print("kay")
    elseif v == "l" then print("el")
    elseif v == "m" then print("em")
    elseif v == "n" then print("en")
    elseif v == "o" then print("ooh")
    elseif v == "p" then print("pee")
    elseif v == "q" then print("queue")
    elseif v == "r" then print("arr")
    elseif v == "s" then print("ess")
    elseif v == "t" then print("tee")
    elseif v == "u" then print("you")
    elseif v == "v" then print("vee")
    elseif v == "w" then print("doubleyou")
    elseif v == "x" then print("ex")
    elseif v == "y" then print("why")
    elseif v == "z" then print(is_canadian and "zed" or "zee")
    elseif v == "?" then print(is_canadian and "eh" or "")
    else                 print("blah")
    end
  end
end
sayit{'h','e','l','l','o','?'}
```

When there are many tests as such, the comparison chain is not always the most efficient. If the number of elements in letters is M and the number of tests is N, then the complexity is O(M*N), or potentially quadratic. A more minor concern is the syntax redundancy of having "v ==" for each test. These concerns (minor as they may be) have been noted elsewhere as well ([Python PEP 3103]).

If we rewrite this as a lookup table, the code can run in linear-time, O(M), and without the redundancy so that the logic is easier to modify at whim:

```
do
  local t = {
    a = "aah",
    b = "bee",
    c = "see",
    d = "dee",
    e = "eee",
    f = "eff",
    g = "gee",
    h = "aych",
    i = "eye",
    j = "jay",
    k = "kay",
    l = "el",
    m = "em",
    n = "en",
    o = "ooh",
    p = "pee",
    q = "queue",
    r = "arr",
    s = "ess",
    t = "tee",
    u = "you",
    v = "vee",
        w = "doubleyou",
        x = "ex",
        y = "why",
        z = function() return is_canadian and "zed" or "zee" end,
        ['?'] = function() return is_canadian and "eh" or "" end
    }
    function sayit(letters)
        for _,v in ipairs(letters) do
            local s = type(t[v]) == "function" and t[v]() or t[v] or "blah"
            print(s)
        end
    end
end
sayit{'h','e','l','l','o','?'}
```

C compilers can optimize the switch statement in a roughly similar way via what is called a jump table, at least under suitable conditions.[2]

Note how the table construction was placed outside the block to avoid recreating the table for each use (table constructions cause heap allocations). This improves performance but has the side-effect of moving the lookup table further from its usage. We might address that with this minor change:

```
do
    local t
    function sayit(letters)
        t = t or {a = "ahh", .....}
        for _,v in ipairs(letters) do
            local s = type(t[v]) == "function" and t[v]() or t[v] or "blah"
            print(s)
        end
    end
end
sayit{'h','e','l','l','o','?'}
```

The above is a practical solution that is the basis for the more elaborate approaches given below. Some are the below solutions are more for syntactic sugar or proof-of-concept rather than recommended practices.

Simple Table of functions

A simple version of a switch statement can be implemented using a table to map the case value to an action. This is very efficient in Lua since tables are hashed by key value which avoids repetitive if <case> then ... elseif ... end statements.

```
action = {
    [1] = function (x) print(1) end,
    [2] = function (x) z = 5 end,
    ["nop"] = function (x) print(math.random()) end,
    ["my name"] = function (x) print("fred") end,
}
```

Usage (Note, that in the following example you can also pass parameters to the function called) :-

```
action[case](params)
```

This is pseudocode for the above:

```
switch (caseVariable)
    case 1: print(1)
    case 2: z=5
    case "nop": print(math.random())
    case "my name": print("fred")
end
```

Table elements called with loadstring()

Here's a neat and compact version using the loadstring() function and calling each element in a table of cases. This method is close to Python's eval() method and it's nice looking. It allows for arguments to be put in formatted.

```
switch = function(cases,arg)
    return assert (loadstring ('return ' .. cases[arg]))()
end

local case = 3

local result = switch({
    [0] = "0",
    [1] = "2^1+" .. case,
    [2] = "2^2+" .. case,
    [3] = "2^3+" .. case
},
case
)
print(result)
```

Case method

This version uses the function switch(table) to add a method case(table,caseVariable) to a table passed to it.

```
function switch(t)
    t.case = function (self,x)
        local f=self[x] or self.default
        if f then
            if type(f)=="function" then
                f(x,self)
            else
                error("case "..tostring(x).." not a function")
            end
        end
    end
    return t
end
```

Usage:

```
a = switch {
    [1] = function (x) print(x,10) end,
    [2] = function (x) print(x,20) end,
    default = function (x) print(x,0) end,
}

a:case(2)  -- ie. call case 2
a:case(9)
```

Caseof method table

Here's yet another implementation of a "switch" statement. This one is based on Luiz Henrique de Figueiredo's switch statement presented in a list message dated Dec 8 1998, but the object/method relationship has been flipped around to achieve a more traditional syntax in actual use. Nil case variables are also handled - there's an optional clause specifically for them (something I wanted), or they can fallback to the default clause. (easily changed) Return values from the case statement functions are also supported.

```
function switch(c)
    local swtbl = {
        casevar = c,
        caseof = function (self, code)
            local f
            if (self.casevar) then
                f = code[self.casevar] or code.default
            else
                f = code.missing or code.default
            end
            if f then
                if type(f)=="function" then
                    return f(self.casevar,self)
                else
                    error("case "..tostring(self.casevar).." not a function")
                end
            end
        end
    }
    return swtbl
end
```

Here's sample usage:

```
c = 1
switch(c) : caseof {
        [1]   = function (x) print(x,"one") end,
        [2]   = function (x) print(x,"two") end,
        [3]   = 12345, -- this is an invalid case stmt
    default = function (x) print(x,"default") end,
    missing = function (x) print(x,"missing") end,
}
```

```
-- also test the return value
-- sort of like the way C's ternary "?" is often used
-- but perhaps more like LISP's "cond"
--
print("expect to see 468:  ".. 123 +
    switch(2):caseof{
        [1] = function(x) return 234 end,
        [2] = function(x) return 345 end
    })
```

Switch returns function instead of table

Yet another implementation of an even more "C-like" switch statement. Based on Dave code above. Return values from the case statement functions are also supported.

```
function switch(case)
    return function(codetable)
        local f
        f = codetable[case] or codetable.default
        if f then
            if type(f)=="function" then
                return f(case)
            else
                error("case "..tostring(case).." not a function")
            end
        end
    end
end
```

Example usage:

```
for case = 1,4 do
    switch(case) {
        [1] = function() print("one") end,
        [2] = print,
        default = function(x) print("default",x) end,
    }
end
```

Note that this works, but trashes the gc with a function closure each time the switch is used (as do most of the examples on this page). Still, i like the way it works. Just don't use it in real life ;-) --PeterPrade

Switch returns callable table instead of function

This one has the exact same syntax as the one above, but is written much more succinctly, as well as differentiates between the default case and a string containing the word 'default'.

```
Default, Nil = {}, function () end -- for uniqueness
function switch (i)
    return setmetatable({ i }, {
        __call = function (t, cases)
            local item = #t == 0 and Nil or t[1]
            return (cases[item] or cases[Default] or Nil)(item)
        end
    })
end
```

`Nil` here is an empty function because it will generate a unique value, and satisfy the [nilpotence] requirement in the `return` statement call, while still being having a value of `true` to allow its use in the `and or` ternary. In Lua 5.2, however, a function might not create a new value if one is present, which will raise problems if you somehow end up using `switch` to compare functions. Should it ever come to this, a solution would be to define `Nil` with yet another table: `setmetatable({}, { __call = function () end })`.

A case-insensitive variant can be made by adding if type(item) == "string" then item = string.lower(item) end, provided all the keys of the table are done the same way. Ranges could potentially be represented by an __index function metatable on the cases table, but that would break the illusion: switch (case) (setmetatable({}, { __index = rangefunc })).

Example usage:

```
switch(case) {
    [1] = function () print"number 1!" end,
    [2] = math.sin,
    [false] = function (a) return function (b) return (a or b) and not (a and b) end end,
    Default = function (x) print"Look, Mom, I can differentiate types!" end, -- ["Default"] ;)
    [Default] = print,
    [Nil] = function () print"I must've left it in my other jeans." end,
}
```

I can't say anything for its resource usage, however, especially compared to other examples here.

Using vararg function to build case list

In the interest of more 'stupid Lua tricks', here's yet another implementation: (Edit: It is necessary that the default functionality is put last in the ... parameter)

```
function switch(n, ...)
    for _,v in ipairs {...} do
        if v[1] == n or v[1] == nil then
            return v[2]()
        end
    end
end

function case(n,f)
    return {n,f}
end

function default(f)
    return {nil,f}
end
```

Example usage:

```
switch( action,
    case( 1, function() print("one")     end),
    case( 2, function() print("two")     end),
    case( 3, function() print("three")   end),
    default( function() print("default") end)
    )
```

Case expression types other than just matching a value

Here's one from TheGreyKnight?, which can handle cases representing ranges, lists and default actions. It also supports mismatch cases and fall-through (ie, continue with the next statement). The part that checks for the "-fall" suffix could probably be made more efficient, but I think this version is easier to read. The functions which are used as the bodies of the cases are passed a single parameter, which is the final form of the switch expression (a feature I've longed for in switches for ages) The supporting functions contain(x, valueList) and range(x, numberPair) merely test whether or not x is a value in the table valueList or a number in the closed range specified by the two elements of numberPair.

```
function switch(term, cases)
    assert(type(cases) == "table")
    local casetype, caseparm, casebody
    for i,case in ipairs(cases) do
        assert(type(case) == "table" and count(case) == 3)
        casetype,caseparm,casebody = case[1],case[2],case[3]
        assert(type(casetype) == "string" and type(casebody) == "function")
        if
                (casetype == "default")
            or  ((casetype == "eq" or casetype=="") and caseparm == term)
            or  ((casetype == "!eq" or casetype=="!") and not caseparm == term)
            or  (casetype == "in" and contain(term, caseparm))
            or  (casetype == "!in" and not contain(term, caseparm))
            or  (casetype == "range" and range(term, caseparm))
            or  (casetype == "!range" and not range(term, caseparm))
        then
            return casebody(term)
        else if
                (casetype == "default-fall")
            or  ((casetype == "eq-fall" or casetype == "fall") and caseparm == term)
            or  ((casetype == "!eq-fall" or casetype == "!-fall") and not caseparm == term)
            or  (casetype == "in-fall" and contain(term, caseparm))
            or  (casetype == "!in-fall" and not contain(term, caseparm))
            or  (casetype == "range-fall" and range(term, caseparm))
            or  (casetype == "!range-fall" and not range(term, caseparm))
        then
            casebody(term)
        end end
    end
end
```

Example Usage:

```
switch( string.lower(slotname), {
   {"", "sk", function(_)
      PLAYER.sk = PLAYER.sk+1
   end },
   {"in", {"str","int","agl","cha","lck","con","mhp","mpp"}, function(_)
      PLAYER.st[_] = PLAYER.st[_]+1
   end },
   {"default", "", function(_)end} --ie, do nothing
})
```

Another form

```
function switch (self, value, tbl, default)
    local f = tbl[value] or default
    assert(f~=nil)
    if type(f) ~= "function" then f = tbl[f] end
    assert(f~=nil and type(f) == "function")
    return f(self,value)
end
```

It avoids repeating functions, since if tbl's entry is a string/number, it follows this value as the case to seek for. I would call this *multiple case statements*. Example usage:

```
local tbl = {hello = function(name,value) print(value .. " " .. name .. "!") end,
bonjour = "hello", ["Guten Tag"] = "hello"}

switch("Steven","hello",tbl,nil) -- prints 'hello Steven!'
switch("Jean","bonjour",tbl,nil) -- prints 'bonjour Jean!'
switch("Mark","gracias",tbl,function(name,val) print("sorry " .. name .. "!") end) -- prints 'sorry Mark!'
```

A Case Statement implemented with Token Filter Macros

My feeling is that `switch` is the wrong model, but that we should look at Pascal's `case` statement as more appropriate inspiration. Here are some possible forms:

```
case (k)
  is 10,11: return 1
  is 12: return 2
  is 13 .. 16: return 3
  else return 4
endcase
......
case(s)
    matches '^hell': return 5
    matches '(%d+)%s+(%d+)',result:
        return tonumber(result[1])+tonumber(result[2])
    else return 0
endcase
```

You can provide a number of values after is, and even provide a range of values. matches is string-specific, and can take an extra parameter which is filled with the resulting captures.

This case statement is a little bit of syntactical sugar over a chain of elseif statements, so its efficiency is the same.

This is implementable using token filter macros (see LuaMacros; the source contains an example implementation), so people can get a feeling for its use in practice. Unfortunately, there is a gotcha; Lua complains of a malformed number if there is no whitespace around ... Also result has to be global.

Metalua's pattern matching

MetaLua comes with an extension that performs structural pattern matching, of which switch/case is just a special case. The example above would read:

```
-{ extension 'match' } -- load the syntax extension module
match k with
| 10 | 11               -> return 1
| 12                    -> return 2
| i if 13<=i and i<=16 -> return i-10 -- was 3 originally, but it'd a shame not to use bindings!
| _                     -> return 4
end
```

No special handling currently exists for regular expressions string matching, although it can be worked around by guards. Proper support can be added quite easily, and will likely be included in a future release.

Relevant resources:

* Step-by-step tutorial about implementing a pattern matching extension[3], and the corresponding sources[4].

* The latest optimized implementation[5]

Object Oriented Approach

You can find full code in SwitchObject.

```lua
local fn = function(a, b) print(tostring(a) .. ' in ' .. tostring(b)) end
local casefn = function(a)
        if type(a) == 'number' then
                return (a > 10)
        end
end
local s = switch()
s:case(0, fn)
s:case({1,2,3,4}, fn)
s:case(casefn, fn)
s:case({'banana', 'kiwi', 'coconut'}, fn)
s:default(fn)

s:test('kiwi') -- this does the actual job
```