Token Filters:
A Macro Facility for Lua

Luiz Henrique de Figueiredo

IMPA and Lua.org

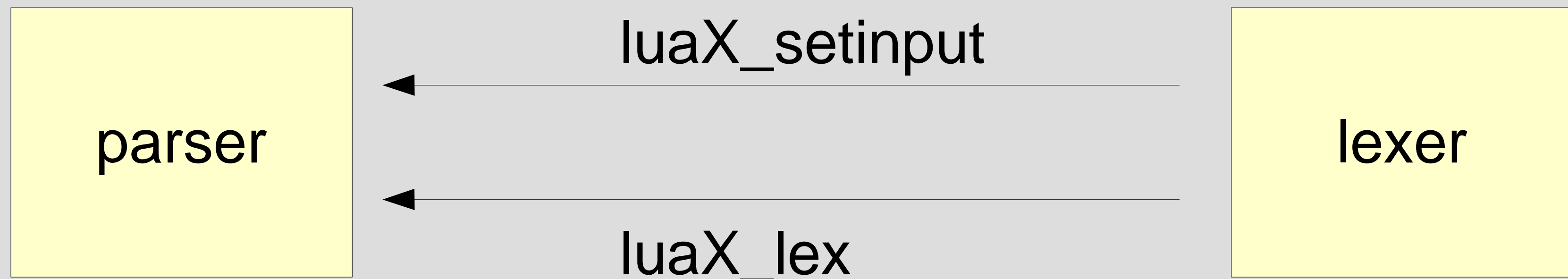
Lua Workshop 2005

Macros for Lua?

- Why
  - enhance extensibility
  - increase expressiveness
  - another metamechanism
- Why not
  - too hard to please everyone ☺
  - which syntax?
  - avoid bloat and slowdown of parsing
    - provide as add-on core module
    - use in `luac`, not in core

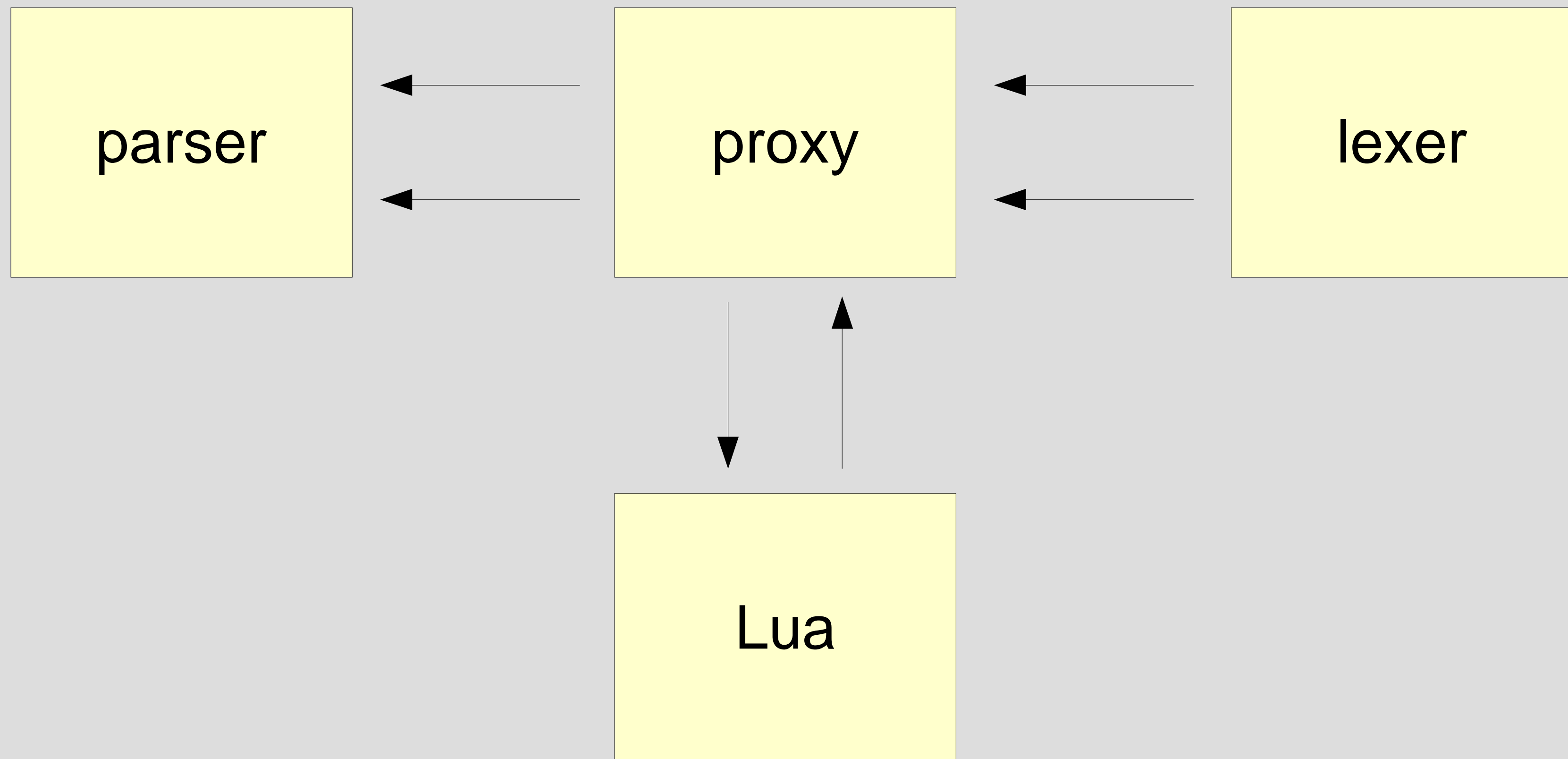
Add-on core module

- Does not change Lua core
- Needs to access core objects and functions



Add-on core module

- Does not change Lua core
- Needs to access core objects and functions



Token filters

- Lua function called whenever parser needs a token.
- Lua function gets a function to be called to get a token.
- Tokens represented as triples: line number, token, and associated value (for `<number>`, `<string>` and `<name>`).

Examples of filtering:
what can be done easily

- Simple value replacement

  $*name* → t[*name*]

  $*name* → f(*name*)

- Abbreviations

  proc → local function

- Bypass restrictions

  .*keyword* → ['*keyword*']

Examples of filtering:
what can be done with some work

- New syntactic sugar

```
lambda (params) expr end   →
function (params) return expr end

unless cond do →
if not (cond) then
```

- Code generation

```
TRACE expr ;   →
do local _ = expr
   print("TRACE expr",_) end
```

Examples of filtering:
what cannot be done

- Syntactical restrictions

```
lambda (params) expr end
```

- New syntactical constructs

```
lambda (params) expr
case (expr) expr: statmnt ... end
```

- Good error messages
  - but probably good enough

Examples of filters:

debug

```
function FILTER(get)
  FILTER=function ()
    local line,token,value=get()
    print(">>>",line,token,value,"\n")
    return line,token,value
  end
end
```

Examples of filters: proc → local function

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
         put(line,"local")
         put(line,"function")
     else
         put(line,token,value)
     end
   end
end                 -- put == coroutine.yield
```

Combining filters

```
local function pipe(f,get,put)
  put = put or coroutine.yield
  local F=coroutine.wrap(f)
  F(get,put)
  return F
end

function FILTER(get)
  FILTER=pipe(f3,pipe(f2,pipe(f1,get)))
end

-- lexer | f1 | f2 | f3 | parser
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
         put(line,"local")
         put(line,"function")
     else
         put(line,token,value)
     end
   end
end                 -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
          put(line,"local")
          put(line,"function")
     else
          put(line,token,value)
     end
   end
end                  -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
         put(line,"local")
         put(line,"function")
     else
         put(line,token,value)
     end
   end
end                -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
         put(line,"local")
         put(line,"function")
     else
         put(line,token,value)
     end
   end
end                    -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
          put(line,"local")
          put(line,"function")
     else
          put(line,token,value)
     end
   end
end                    -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
          put(line,"local")
          put(line,"function")
     else
          put(line,token,value)
     end
   end
end                    -- put == coroutine.yield
```

Filter protocol

```
function f3(get,put)
   put()
   while true do
     local line,token,value=get()
     if token=="<name>" and value=="proc"
     then
         put(line,"local")
         put(line,"function")
     else
         put(line,token,value)
     end
   end
end                    -- put == coroutine.yield
```

Conclusions

- Many simple tasks easy
  - tokens, not characters
  - lexer does not flag unknown characters
    - can use unused chars to flag start of constructs
    - cannot create new complex tokens
- Some tasks harder
  - need to track nesting levels
  - need to track balancing parentheses
  - need to buffer long sequences of tokens
  - hard to find where expressions end
- Coroutines are nice for implementing filters

Many loose ends to tie

- Add ways to ask parser info
  - easy to decide whether name refers to a global, a local or an outer local
- How to stop and resume filtering?
- Write a “def” macro
- Nesting of simple m4-like expansion

```
$f(x,$g(a,b,c),y,z)
```

  - dynamically change “get” function?