情報特別演習 I
例外処理を追加した Lua 処理系の実装

河原 悟

平成 28 年 2 月 8 日

目 次

第1章　概要

今回はプログラミング言語 Lua[*1.1] の interpreter をフルスクラッチで構築することにより、interpreter の構築方式を学んだ。また、*CPS*、継続渡し形式という手法を用いた実装により、既存の処理系にはない例外処理機構を追加した（ソースコード 1.1）。

ソースコード 1.1: add new syntax: try-catch statement

```
try
  print("hello")
  local n = 5

  if n < 11 then
    throw()
  end

  print("world")
catch
  print("thrown", n)
end
```

処理系の記述には MoonScript[*1.2] というプログラミング言語を用いた。MoonScript とは Lua に transpile されることを目的とした言語であり、文法が CoffeeScript[*1.3] に類似している。Lua と同様の機能を持ちながら多様な構文を備えており、意味の同じ文を Lua よりも簡潔な記述実装が可能である。

1.1　背景

プログラミング言語 Lua は、実装が軽量でありながら柔軟な記述力を持ち、近年多くのソフトウェアに組み込まれるなど多くの活躍がみられる。

一方、Scala、Ruby、Rust などモダンな言語や、Haskell や OCaml などの関数型言語に搭載されている例外処理機構を Lua は持たない。

例外処理機構の実装には、一部の言語処理系では高階関数や closure を利用した CPS という手法が用いられていることが分かった。

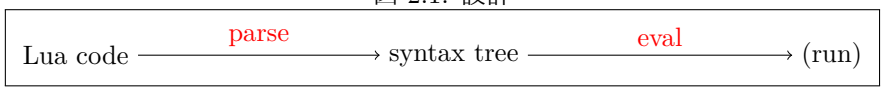
MoonScript は関数がファーストクラスであることから、関数を利用した CPS による例外処理機構の実装は可能であり、また、関数をファーストクラスとして扱えるという Lua/MoonScript の特徴の一つを活かした実装ができると考え、着手した。

[*1.1] http://lua.org
[*1.2] http://moonscript.org
[*1.3] http://coffeescript.org

第2章　設計、実装

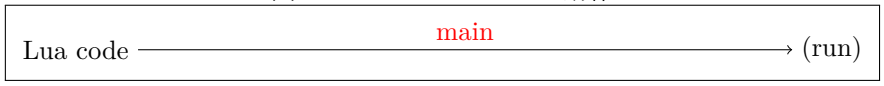
本演習で実装した言語処理系は図 2.1 のような構造になっている。

図 2.1: 設計



parser、evaluator はそれぞれモジュールとして実装されており、REPL にもなっている本体部分が各モジュールをロードするという構成になっている（図 2.2）。

図 2.2: ユーザーからみた動作



2.1　parser

2.1.1　PEG

parser には *PEG* を用いた。PEG とは Parsing Expressin Grammar のアクロニムである。曖昧さがないという点が BNF 記法などの文法定義方法と異なる。

2.1.2　PEG と BNF 記法の比較

例としてプログラミング言語の `if` 文を PEG と BNF 記法で表現する。BNF 記法では以下のようになる（ソースコード 2.1）。

ソースコード 2.1: if statement written in BNF notation

```
Statement ::= ... | IfState | ... ;
IfState   ::= "if" "(" Expression ")" Statement "else" Statement
            | "if" "(" Expression ")" Statement;
```

PEG では以下のように表現できる（ソースコード 2.2）。,

ソースコード 2.2: if statement written in PEG

```
Statement ← ... / IfState / ...
IfState   ← "if" "(" Expression ")" Statement "else" Statement
          / "if" "(" Expression ")" Statement
```

これらに対して以下のような入力を与える（ソースコード 2.3）。

ソースコード 2.3: input code

```
if (cond1) if (cond2) some_statement else some_statement
```

入力 2.3 に対し、2.1 による構文解析では以下のように 2 種類の解釈が可能になる（ソースコード 2.4）。

ソースコード 2.4: two interpretation

```
1 if (cond1) (if (cond2) some_statement else some_statement)
2 if (cond1) (if (cond2) some_statement) else some_statement
```

yacc などのツールではそのツール固有の機能を用いることでこれを一意に定める。
一方 2.2 による PEG の構文解析では解釈が一意に定まる（ソースコード 2.5）。

ソースコード 2.5: only a interpretation

```
1 if (cond1) (if (cond2) some_statement) else some_statement
```

/演算子により複数の選択肢（ここでは"if" "(" Expression ")" Statement "else" Statementか"if" "(" Expression ")" Statementか）がある場合、常に左辺からマッチを試し、失敗したらバックトラックして右辺とマッチするかを試す、という動作を繰り返すため、必ず一意な解釈が可能になる。文法定義の時点で曖昧さがないことに一つ PEG の優位性がうかがえる。

今回は PEG の Lua モジュールである LPeg、その Lua 実装である LuLPeg[*2.1] を用いた。A.1 に parser のソースコードを示す。

2.2　evaluator

2.2.1　CPS

evaluator の実装には CPS、継続渡し形式という手法を用いた。CPS とは Continuation Passing Style のアクロニムである。
例として、整数リストを受け取り、その総乗を返す関数を考える（ソースコード 2.6、2.7）。

ソースコード 2.6: prod

```
1 prod = (t) ->
2   h = table.remove t, 1 -- pop the head from table t
3   switch h
4     when nil then 1
5     else h * prod t
6
7 print prod {1, 2, 3} -- 6
```

ソースコード 2.7: prod_cps

```
1 prod_cps = (t, k = (x) -> x) ->
2   h = table.remove t, 1
3   switch h
4     when nil then k 1
```

[*2.1] https://github.com/pygy/LuLPeg

```
    else prod_cps t, (x) -> k x * h

print prod_cps {1, 2, 3} -- 6
```

整数リストの先頭を取り、再帰的に乗算する。空リストは 1 を返すという実装になっている。2.6 を CPS に変換したものが 2.7 になる。

2.6 は再帰的に戻り値を返している一方で、2.7 は再帰呼出し時に渡す関数 k のなかに計算を入れている。そしてその関数 k が最終的に呼び出されることで全体としての結果を返している。計算を関数の中に入れることを “継続を渡す” と言い、継続渡し形式と呼ばれる所以となっている。

2.3　例外処理

この手法を用いた例外処理の実装について、実際のソースコードを見ながら入力 1.1 に対する出力を考える（ソースコード B.1）。

1.1 に対し、まず parser（ソースコード A.1）は以下の抽象構文木を返す（ソースコード 2.8）。木は table により表現されている。

ソースコード 2.8: generated syntax tree

```
{{
  body: {{
      "print", {'"hello"' },
      label: "funcall" -- print("hello")
    }, {
      { "n" }, {"5" },
      label: "localvarlist" -- local n = 5
    }, {
      body: {{
          "throw", {},
          label: "funcall"
        }},
      cond: {
        "n", "11",
        label: "exp",
        op: "<"
      },
      label: "if" -- if n < 11
    }, {
      "print", {'"world"' },
      label: "funcall" -- print("world")
    }},
  catchbody: {{
      "print", {'"thrown"', "n" },
      label: "funcall" -- print("thrown", n)
    }},
  label: "try"
}}
```

これを 417 行目に定義されている eval 関数に渡す（ソースコード 2.10）。抽象構文木の先頭を取り出し、344 行目に定義されている eval_body に渡される（ソースコード 2.9）。

ソースコード 2.9: part of eval_body

```
343 ......
344 eval_body = (body, env, k0, k) -> switch body.label
345   when "try"
346     k1 = loop: k0, excep: (args, env) ->
347       eval_exp args, env, k0, (t) ->
348         env._ERR = t
349         eval body.catchbody, env, k0, noop
350     k eval body.body, env, k1, (e) ->
351       for k, v in pairs e do env[k] = v if env[k]
352 ......
```

ソースコード 2.10: part of eval

```
416 ......
417 eval = (syntaxtree, env = {}, k0 = (cp_tbl kinit), k = noop) ->
418   unless syntaxtree and syntaxtree[1]
419     k env
420   else
421     syntaxtree = cp_tbl syntaxtree
422     env = setmetatable (cp_tbl env), __index: __ENV, __mode: 'kv'
423     eval_body remove(syntaxtree, 1), env, k0, -> eval syntaxtree, env, k0, k
424 ......
```

まず 1 番目の要素の label が"try"なので、switch 文によりソースコード 2.9 346 行目からの処理に移る。

k1 はループや例外発生時に呼び出す関数の table である。key が excep の要素が例外時に呼ばれ、excep の中（ソースコード 2.9 349 行目）で catch ブロックに飛ぶという定義になっている。

ソースコード 2.8 の 8 行目まで実行し、“hello” を出力し n = 5 という代入が行われたとする。2.8 の 8~19 行目が if n < 11 then throw() end の抽象構文木となっている。ここで n は代入により 5 という値なので、if ブロック内の throw() が呼ばれる。

関数呼び出しの処理をする、257 行目に定義されている eval_funcall をみる（ソースコード 2.11）。

ソースコード 2.11: part of eval_funcall

```
256 ......
257 eval_funcall = (func, args, env, k0, k) ->
258   if func == "throw" then return k0.excep args[1], env
259 ......
```

throw はそもそも定義がなく、ソースコード 2.11 258 行目にあるように、渡された k0.excep に値を適用するに過ぎない。ここで呼ばれる k0.excep とは、ソースコード 2.9 346 行目に定義した k1.excep であり、これにより try ブロックの残りを無視して catch ブロックに飛ぶ。

この通常の継続 k を実行するか、k を破棄して k0.excep に飛ぶかという選択が今回の例外処理機構の醍醐味となっている。

2.4　main

先述の通り、parser と evaluator はモジュールとして記述され、本体部分が読み込んでいる（ソースコード 2.12）。

ソースコード 2.12: part of llix

```
3  ......
4  parse = require"parse"
5  eval = require"eval"
6  ......
```

コマンドライン引数にファイルをを渡すと、そのファイルを実行し、引数がなければ REPL として動作する。REPL では、入力の補完や履歴の再利用を可能にするため、lua-linenoise[*2.2] というモジュールを、出力をよりヒューマンリーダブルなものにするため、inspect[*2.3] というモジュールを用いた。

また、Lua には GC が備わっているが、実行速度向上のため、入力の評価の前後で GC を停止/再開している（ソースコード 2.13、ソースコード 2.14）。

ソースコード 2.13: lazy_gc_stop

```
6  ......
7  lazy_gc_stop = (...) =>
8    collectgarbage "stop"
9    with {@ ...}
10     collectgarbage "restart"
11 ......
```

ソースコード 2.14: parseval

```
64 ......
65 parseval = (obj) ->
66   collectgarbage "stop"
67
68   if type(obj) == "table"
69     ok, cont = pcall eval, obj
70     if ok
71       not obj.quiet and iprint cont._llix_tmp
72       eval parse"_llix_tmp = nil"
73     else print"failed to parse"
74   else
75     tree = parse obj
76     ok, cont = pcall eval, tree
77
78     unless ok then print cont
79     else
80       if tree[1].label == "varlist" then vars = tree[1][1]
81
82   collectgarbage "restart"
83 ......
```

[*2.2] https://github.com/hoelzro/lua-linenoise
[*2.3] https://github.com/kikito/inspect.lua

第3章　評価

3.1　動作

次のようなテストケースを考えた（ソースコード 3.1）。

Code 3.1: llix/test/arithmetic_eval.lua

```
local P, S, V, R, C, Ct, match do
  local _obj_0 = require("lpeg")
  P, S, V, R, C, Ct, match = _obj_0.P, _obj_0.S, _obj_0.V, _obj_0.R, _obj_0.C, _obj_0.Ct,
      _obj_0.match
end

local Space = S(' \n\t')
local Num = C(P('-') ^ -1 * R('09') ^ 1) * Space ^ 0
local TermOp = C(S('+-')) * Space ^ 0
local FactOp = C(S('*/%')) * Space ^ 0
local HatOp = C(P('^')) * Space ^ 0
local Open = '(' * Space ^ 0
local Close = ')' * Space ^ 0

local G = P{
  "Exp",
  Exp = Ct(V('Term') * (TermOp * V('Term')) ^ 0),
  Term = Ct(V('Fact') * (FactOp * V('Fact')) ^ 0),
  Fact = Ct(V('Hat') * (HatOp * V('Hat')) ^ 0),
  Hat = Num + Open * V('Exp') * Close
}
G = Space ^ 0 * G * -1

local op_table = {
  ["+"] = function(l, r) return l + r end,
  ["-"] = function(l, r) return l - r end,
  ["*"] = function(l, r) return l * r end,
  ["/"] = function(l, r) return l // r end,
  ["%"] = function(l, r) return l % r end,
  ["^"] = function(l, r) return l ^ r end
}

local function eval(x)
  if type(x) == "string" then
```

```
    return tonumber(x)
  else
    local n1 = eval(table.remove(x, 1))
    try -- catch zero division
      for i = 1, #x, 2 do
        local op = x[i]
        local n2 = eval(x[i + 1])

        if op == "/" and n2 == 0 then throw() end

        n1 = op_table[op](n1, n2)
      end
      return n1
    catch
      print "ZeroDivision Error"
    end
  end
end

local function parse(str)
  return G:match(str)
end

try -- using TRY-CATCH instead of )io.exit())
  print"CALC>>"
  while true do
    io.write "$ "
    local ok, l = pcall(io.read)
    if l == "exit" or not ok then
      throw()
    end

    if not l then
      print()
    elseif #l >= 1 then
      try -- evaled or not
        local t = parse(l)
        if not t then throw() end

        local e = eval(t)
        if e then
          print("ANS " .. tostring(e))
        end
      catch
        print "!!parse error!!"
```

```
79        end
80      end
81    end
82  catch
83    print "<<EXIT"
84  end
```

二項演算子+-*/%^と数字、括弧を用いた言語とその評価器（電卓）の実装である。以下にこの言語の文法を PEG で示す（ソースコード 3.2）。

ソースコード 3.2: the grammar of arithmetic evaluator written in PEG

```
1  Grammar <- Exp
2      Exp <- (Term TermOp Term)*
3     Term <- (Fact FactOp Fact)*
4     Fact <- (Hat HatOp Hat)*
5      Hat <- Num / Open Exp Close
6    Space <- ' ' / '\n' / '\t'
7      Num <- '-'? [0-9]+ Space*
8  TermOp <- ('+' / '-') Space*
9  FactOp <- ('*' / '/' / '%') Space*
10  HatOp <- '^' Space*
11   Open <- '(' Space*
12  Close <- ')' Space*
```

中間期法を用いた整数の演算である。ゼロ除算、パースエラー時に例外を用いている（ソースコード 3.1 37 行目、69 行目）。

実際の動作は以下のようになる（ソースコード 3.3）。

ソースコード 3.3: run

```
1  $ llix test/arithmetic_eval.lua
2  CALC>>
3  $ 3 * (5 - 9)
4  ANS -12
5  $ ( + 5 12 )
6  !!parse error!!
7  $ 3 / 0
8  ZeroDivision Error
9  $
10 $ ^C<<EXIT
```

これよりソースコード 3.3 程度の規模のものを動作させることに成功した。3.3 の 9 行目のステップの時点でのメモリ使用量は 16MB 程度であり、さらに数倍程度の規模のコードを実行することができると予想される。

3.2　実行速度

本演習では実行速度に注力せずに実装した。たとえばここでフィボナッチ数を求める関数 `fib` の純朴な実装（ソースコード 3.5）を用いて、`htt://lua.org` の提供する Lua 処理系と実行速度の比較をおこなう。また、実行速度の計測は以下の環境でおこなった（表 3.1）。

表 3.1: 実行環境

Lua 処理系	Lua 5.3.2
OS	ArchLinux x86_64, Kernel 4.3.3-3
CPU	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHza

Code 3.4: llix/test/speedtest/fib.lua

```
1 function fib(n)
2   if n < 2 then return n
3   else return fib(n - 1) + fib(n - 2) end
4 end
5
6 fib(10)
```

ソースコード 3.5: fib

```
1 function fib(n)
2   if n < 2 then return n
3   else return fib(n - 1) + fib(n - 2) end
4 end
```

`fib(10)` を 10000 回おこなった平均が以下になる（表 3.2）。

表 3.2: `fib(10)`の実行速度の比較

処理系	平均時間 (sec)
Lua	0.0010825
本演習	0.16158

第4章　まとめ

4.1　成果

4.1.1　処理系の実装

`while`、`for`、`until`、`if`、`function` 構文、代入や変数のスコープ、関数の再帰の実装が完了した。また、処理系を CPS を用いて実装することにより、あらたに例外処理の構文を追加することができた。これにより、ソースコード 3.1 程度のプログラムを動作させることが可能になった。

4.1.2　OSS への寄与

今回使用したツールやモジュールは全てオープンソースであり、だれでも開発に参加できるかたちになっている。本演習中、使用したモジュール inspect にバグが見つかったため、修正、報告をした[*4.1] 。加えて、ホスト言語となった MoonScript へ送ったいくつかの Pull Request がマージされた[*4.2] 。

そして本演習で実装した処理系も MIT ライセンスで公開した[*4.3] 。Lua のパッケージマネージャー Luarocks[*4.4] にも公開した[*4.5] ため、インストールが容易になっている。

4.2　課題

4.2.1　実行速度

3.2 に記述した通り、実行速度は既存の処理系と比べ $\frac{1}{150}$ 倍となっている。今回のような meta-circular interpreter は、実行速度が元の処理系よりも数十倍遅くなる。

これに対して、compiler の作成、及び compile による最適化があげられる。CPS のボトルネックは関数呼び出しとなっており、その最適化ができればより高速に動作すると考えられる。

この場合、ターゲットが機械語となり、スクリプト言語である Lua よりも、低レイヤーに関する記述に適した他言語の使用が望まれる。

4.2.2　Lua への完全対応

本演習で実装した Lua 処理系は Lua に完全には対応していない。既知のバグはいくつかあり、それらを直してくなどすることで、完全な Lua 処理系を目指している。

これにより、例外処理を用いて自身を実装し、動作することが可能になる。

[*4.1] https://github.com/kikito/inspect.lua/pull/21
[*4.2] https://github.com/leafo/moonscript/pulls?q=is%3Apr+author%3ANymphium++is%3Amerged
[*4.3] https://github.com/Nymphium/llix
[*4.4] http://luarocks.org
[*4.5] https://luarocks.org/modules/nymphium/llix

付 録 A　parser

Code A.1: llix/parse.moon

```
import P,S, V,
  C, Cb, Cc, Cg, Cs, Cmt, Ct, Cf, Cp
  locale, match from require'lulpeg'
import insert, remove from table

locale = locale!
K = (k) -> P(k) * -(locale.alnum + P'_')
CV = (pat) -> C V pat
CK = (pat) -> C K pat
CP = (pat) -> C P pat
CS = (pat) -> C S pat
CtV = (pat) -> Ct V pat
opt = (pat) -> (pat)^-1
ast = (pat) -> (pat)^0

-- spaces(a, b, c, ..., z) ==> a * V'Space' * b * V'Space' * c * ... * V'Space' * z
spaces = (head, ...) ->
  args = {...}

  not args[1] and head or head * V'Space' * spaces unpack args

keywords = (head, ...) ->
  args = {...}

  not args[1] and K(head) or K(head) + keywords unpack args

-- lbl_tbl(lbl, l1, l2, l3, ..., ln) (c1, c2, c3, ..., cn, cn+1, ...) ==> {label: lbl, l1: c1, l2:
    c2, ..., ln:cn, cn+1, cn+2, ...}
lbl_tbl = (lbl, ...) ->
  tags = {...}
  (...) -> with args = {label: lbl, ...}
    if type(args[1]) == "string" and #args[1] < 1
      remove args, 1
    else for i = 1, #args
      if t = tags[i]
        cont = args[i]
        args[t] = cont if #cont > 0
```

```
37        args[i] = nil
38
39  -- unificate ({else{if}}, {else{if}}, ... ==> {else{if{else{if ...)
40  gen_nesttbl = (...) ->
41    gn = (...) ->
42      args = {...}
43      with tail = remove args
44        if args[1]
45          insert(args[#args][1], tail)
46
47          return gen_nesttbl unpack args
48
49    t = gn ...
50
51    type(t) == "table" and t or nil
52
53  -- "x", "+", "y", "-", "z"... ==> {op:"-", {op:"+", "x", "y"}, "z"}
54  gen_binoptbl = (a, b, c, ...) ->
55    unless c
56      b and {op:a, label:"exp", b} or a
57    else gen_binoptbl {op:b, label:"exp", a, c}, ...
58
59  -- "not", "not", "not", "true" ==> {op: "not", {op:"not", {op: "not", "true"}}}
60  gen_unoptbl = (...) ->
61    args = {...}
62    val = remove args
63    t = {op:remove(args), label:"exp", val}
64
65    insert args, t
66
67    #args < 2 and (t.op and t or val) or gen_unoptbl unpack args
68
69  gen_exp = (next, pat) -> V(next) * ast(V'Space' * pat * V'Space' * V(next)) / gen_binoptbl
70
71  gen_tblaccess = (a, ...) -> #{...} < 1 and a or {label: "tableaccess", (type(a) == "string" and a\
       gsub("\"", "") or a), gen_tblaccess ...}
72
73  -- parse(Funcbody) ==> {....}, {....}, {....} constantly make three tables
74  -- normalize_funcbody = (...) ->
75    -- body = {...}
76    -- args = remove body, 1
77    -- cont = remove body
78
79    -- if cont.label == "return"
80      -- args, body, cont
```

```
  -- else
    -- insert(body, cont)
    -- args, body


llix = P{
  opt(P'#' * ast(1 - P'\n') * P'\n') * V'Space' * CtV'Chunk' * V'Space' * -P(1)
  Keywords: keywords 'and', 'break', 'do', 'else', 'elseif', 'end', 'false', 'for',
    'function', 'if', 'in', 'local', 'nil', 'not', 'or', 'repeat', 'return', 'then', 'true', 'until
        ', 'while', 'try', 'catch'

  Chunk: ast(V'Space' * V'Stat' * opt(V'Space' * P';')) * opt(V'Space' * V'Laststat' * opt(V'Space'
      * P';'))
  Block: V'Chunk'
  Space: ast(locale.space + V'Comment')
  Comment:
    (P'--' * V'Longstring' +
    P'--' * ast(P(1) - P'\n') * (P'\n' + -P(1))) /->

  Number:
    P'0x' * (locale.xdigit)^1 * -(locale.alnum + P'_') +
    locale.digit^1 * opt(P'.' * locale.digit^1) * opt(S'eE' * locale.digit^1) * -(locale.alnum + P'
        _') +
    P'.' * locale.digit^1 * opt(S'eE' * locale.digit^1) * -(locale.alnum + P'_')

  Longstring:
    C P{
      V'open' * C(ast(P(1) - V'closeeq')) * V'close' / 2
      open: '[' * Cg(ast(P'='), 'init') * P'[' * opt(P'\n')
      close: ']' * C(ast(P'=')) * ']'
      closeeq: Cmt(V'close' * Cb'init', (_, _, a, b) -> a == b)
    }

  String:
    (((P"\"" * C(ast(P"\\" * P(1) + (1 - P"\"")))) * P"\"") +
    (P"'" * C(ast(P"\\" * P(1) + (1 - P"'")))) * P"'")) / (str) -> "\"#{str}\"") +
    (V"Longstring" / (a) -> a)

  Fieldsep: P',' + P';'
  Name: (locale.alpha + P'_') * ast(locale.alnum + P'_') - V'Keywords'
  Stat:
    spaces(K'do', V'Block', K'end') / lbl_tbl'do' +
    spaces(K'while', V'Exp', K'do', CtV'Block', K'end') / lbl_tbl('while', 'cond', 'body') +
    spaces(K'repeat', CtV'Block', K'until', V'Exp') / lbl_tbl('repeat', 'body', 'cond') +
```

```
122     spaces(K'if', V'Exp', K'then', CtV'Block', (ast(spaces(K'elseif', V'Exp', K'then', CtV'Block')
            / lbl_tbl('if', 'cond', 'body', 'elsebody') / lbl_tbl'else' * V'Space') *
123       opt(K'else' * V'Space' * CtV'Block' / lbl_tbl'else' * V'Space')/((e) -> e) / gen_nesttbl), K'
            end') / lbl_tbl('if', 'cond', 'body', 'elsebody') +
124     spaces(K'for', CV'Name', P'=', V'Exp', P',', V'Exp') * spaces(opt(V'Space' * P',' * V'Space' *
            V'Exp')/((e) -> e) , K'do', CtV'Block', K'end') / lbl_tbl('for', 'var', 'cnt', 'to', 'step'
            , 'body') +
125     spaces(K'for', CtV'Namelist', K'in', CtV'Explist', K'do', CtV'Block', K'end') / lbl_tbl('iter',
             'namelist', 'explist', 'body') +
126     spaces(K'function', V'Funcname', V'Funcbody', K'end') / lbl_tbl('funcdef', 'name', 'args', '
            body') +
127     spaces(K'local', K'function', CV'Name', V'Funcbody', K'end') / lbl_tbl('localfuncdef', 'name',
            'args', 'body') +
128     K'local' * V'Space' * CtV'Namelist' * opt(V'Space' * P'=' * V'Space' * CtV'Explist') / lbl_tbl'
            localvarlist' +
129     spaces(CtV'Varlist', P'=', CtV'Explist') / lbl_tbl'varlist' +
130     V'Funcall' +
131     spaces(K'try', CtV'Block', K'catch', CtV'Block', K'end') / lbl_tbl('try', 'body', 'catchbody')
132
133   Laststat: K'return' * (opt(V'Space' * V'Explist')) / lbl_tbl'return' + K'break' / -> label:'break
          '
134   Namelist: CV'Name' * ast(V'Space' * P',' * V'Space' * CV'Name')
135   Varlist: V'Var' * ast(V'Space' * P',' * V'Space' * V'Var')
136   Value:
137     CK'nil' +
138     CK'false' +
139     CK'true' +
140     CV'Number' +
141     V'String' +
142     CP'...' +
143     V'Funcdef' +
144     V'Tableconstructor' +
145     V'Funcall' +
146     V'Var' +
147     spaces(P'(', V'Exp', P')')
148
149   Exp: V'lor'
150   lor: gen_exp 'land', CK'or'
151   land: gen_exp 'cmp', CK'and'
152   cmp: gen_exp 'or', C(P'<=' + P'>=' + P'~=' + P'==' + S'<>')
153   or: gen_exp 'xor', CP'|'
154   xor: gen_exp 'and', CP'~'
155   and: gen_exp 'shift', CP'&'
156   shift: gen_exp 'cnct', C(P'<<' + P'>>')
157   cnct: gen_exp 'term', CP'..'
```

```
  term: gen_exp 'fact', CS'+-'
  fact: gen_exp 'hat', C(P'//' + S'*/%')
  hat: gen_exp 'expend', CP'^'
  expend: ast(C((K'not') + S'-~#') * V'Space') * V'Value' / gen_unoptbl + gen_exp 'Value', V'Exp'
  Explist: V'Exp' * ast(V'Space' * P',' * V'Space' * V'Exp')
  Index:
    spaces(P'[', V'Exp', P']') +
    P'.' * V'Space' * (CV'Name' / (n) -> "\"#{n}\"")

  Colonfunc: P':' * V'Space' * CV'Name' * V'Space' * V'Callargs' / lbl_tbl 'colonfunc', 'func', '
      args'
  Call: V'Callargs' + V'Colonfunc' -- * V'Space' * V'Callargs' / lbl_tbl'colonfunc'
  Prefix: spaces(P'(', V'Exp', P')') + CV'Name'
  Suffix: V'Call' + V'Index'
  Var: (V'Prefix' * ast(V'Space' * V'Suffix' * #(V'Space' * V'Suffix')) * V'Space' * V'Index' + CV'
      Name') / gen_tblaccess
  Funcall: V'Prefix' * ast(V'Space' * V'Suffix' * #(V'Space' * V'Suffix')) / gen_tblaccess * V'
      Space' * V'Call' / lbl_tbl'funcall'
  Funcname: CV'Name' * ast(V'Space' * P'.' * V'Space' * CV'Name') * opt(V'Space' * P':' * V'Space'
      * CV'Name')

  Callargs:
    Ct(P'(' * V'Space' * opt(V'Explist' * V'Space') * P')' +
    (V'Tableconstructor' + V'String'))

  Funcdef: K'function' * V'Space' * V'Funcbody' * V'Space' * K'end' / lbl_tbl('annonymousfuncdef',
      'args', 'body')
  Funcbody: spaces P'(', (opt(V'Parlist') / lbl_tbl'args'), P')', CtV'Block'

  Parlist: (V'Namelist' * opt(V'Space' * P',' * V'Space' * CP'...') + CP'...')
  Tableconstructor: P'{' * V'Space' * (opt(V'fieldlist' * V'Space') / lbl_tbl'constructor') * P'}'
  fieldlist: V'Field' * ast(V'Space' * V'Fieldsep' * V'Space' * V'Field') * opt(V'Space' * V'
      Fieldsep')
  Field:
    Ct(spaces(P'[', CtV'Exp', P']', P'=', V'Exp')) +
    Ct(spaces(CV'Name', P'=', V'Exp')) + V'Exp'
}

(msg) ->
  tree = {llix\match msg}

  if h = tree[1]
    #tree > 1 and tree or h
  else nil, "Failed to parse"
```

付 録 B　evaluator

Code B.1: llix/eval.moon

```
import insert, remove from table
_insert = insert

insert = (t, i, o) -> with t
  unless o then _insert t, i
  else _insert t, i, o

-- for debug
inspect = require"inspect"
iprint = (...) -> print (inspect {...})\match("^{ (.*) }$")
---

type_s = (obj) -> type(obj) == "string"
type_t = (obj) -> type(obj) == "table"
is_funcall = (t) -> type_t(t) and t.label == "funcall"
noop = => @
cp_tbl = (t) -> {k, (type_t v) and (cp_tbl v) or v for k, v in pairs t}

strmguard = (str, t) ->
  for i = 1, #t, 2
    unless type_s(t[i]) or type(t[i + 1]) == "function"
      error "strmguard failed"

    if str\match t[i]
      return t[i + 1]!

  t.default! if t.default

strip_string = (str) ->
  with dq = str\match "^\"(.*)\""
    unless dq then dq = str\gsub("^%[(=*)%[(.*)%]%1%]", "%2")

deeval_str = (x) ->
  if type_s x
    noop x\gsub "^(.*)", "\"%1\""
  else x or "nil"

```

```
38 binop_table = {"<=", ">=", "~=", "==", "<", ">", "..",
39   "+", "-", "*", "/", "%", "^", ">>", "<<", "|", "&"}
40 binop = setmetatable {op, (load"return function(l, r) return l #{op} r end")! for op in
       *binop_table},
41   __index:
42     "//": (l, r) ->
43       if r == 0
44         -- XXX: why doesn't )error) work at repl ... ??
45         -- io.stderr\write "attempt to divide by zero\n"
46         error "attempt to divide by zero"
47       else l // r
48   __call: (op, left, right) => @[op] left, right
49
50 uniop_table = {"-", "~", "#", "not"}
51 uniop = setmetatable {op, (load"return function(e) return #{op}(e) end")! for op in *uniop_table},
52   __index: (op) -> (load"return function(e) return #{op}(e) end")!
53   __call: (op, e) => @[op] e
54
55 -- call stack
56 funstack =
57   pushcresume: (fun) => coroutine.resume (insert @, {coroutine.create fun})[#@][1]
58   stopregret: (ret) => coroutine.yield (insert @[#@], ret)[#@][1]
59   pop: => remove @
60
61 -- environment initialize
62 __ENV =
63   :_VERSION
64   arg: cp_tbl arg
65   :assert
66   bit32: cp_tbl bit32
67   :collectgarbage
68   coroutine: cp_tbl coroutine
69   debug: cp_tbl debug
70   :dofile
71   :error
72   :getmetatable
73   io: cp_tbl io
74   :ipairs
75   :load
76   :loadfile
77   :loadstring
78   math: cp_tbl math
79   :module
80   :next
81   os: cp_tbl os
```

```
  :package
  :print
  :pairs
  :pcall
  :rawequal
  :rawget
  :rawset
  :require
  :select
  :setmetatable
  string: cp_tbl string
  table: cp_tbl table
  :tonumber
  :tostring
  :type
  :unpack
  utf8: cp_tbl utf8
  :xpcall

__ENV.package.loaded = with __ENV
  ._G = __ENV
  ._ENV = __ENV

-- for mutual recursive functions
local *

-- tbl = {"t", "i"} ==> env["t"]["i"]
-- )is_dec == true) ==> env["t"], "i"
expand_tbl = (tbl, is_dec, env, k0, k) ->
  if type_t tbl[2]
    switch tbl[2].label
      when nil
        env._llix_tmp_tbl = env._llix_tmp_tbl and env._llix_tmp_tbl[tbl[2][1]] or env[tbl[1]][tbl[2
            ][1]]

        expand_tbl tbl[2], is_dec, env, k0, k
      when "exp" then eval_exp tbl[2], env, k0, (e) ->
        tbl[2] = e
        expand_tbl tbl, is_dec, env, k0, k
      when "funcall" then eval_funcall tbl[2][1], tbl[2][2], env, k0, (e) ->
        tbl[2] = remove e, 1
        expand_tbl tbl, is_dec, env, k0, k
  else
    f = (e) ->
      if tmpt = env._llix_tmp_tbl then tmpt[e]
```

```
126       else
127         if p = env[tbl[1]] then p[e]
128         else
129           -- XXX: why doesn't )error) work at repl ... ?????
130           -- io.stderr\write "attempt to index a nil valie (local '#{tbl[1]}')\n"
131           error "attempt to index a nil valie (local '#{tbl[1]}')"
132
133     unless is_dec
134       if type(tbl[2]) == "number" then k f tbl[2]
135       elseif m = tbl[2]\match "^\"(.*)\"$" then k f m
136       elseif m = tonumber tbl[2] then k f m
137       else eval_exp tbl[2], env, k0, (e) -> k f e
138     else k (env._llix_tmp_tbl and env._llix_tmp_tbl or env[tbl[1]]), tbl[2]
139
140
141 -- t = {x, y, z} ==> {evaled_ x, evaled_y, evaled_z}
142 eval_tbl = (fields, pos = 1, env, k0, k) ->
143   unless fields[1] then k fields
144   else
145     local key, val
146     head = remove(fields, 1)
147     tblf = (val, pos) -> (key) ->
148       eval_exp val, env, k0, (x) -> eval_tbl fields, pos, env, k0, (y) ->
149         k with y do y[key] = x
150
151     switch type head
152       when "table"
153         switch head.label
154           when not fields[1] and "funcall"
155             return eval_funcall head[1], head[2], env, k0, (t) -> k t
156           when "tableaccess", "constructor"
157             key = pos
158             val = (head.label == "tableaccess" and head or label: "constructor")
159             pos += 1
160           when nil
161             key = type_t(head[1]) and remove(head[1]) or deeval_str head[1]
162             val = head[2]
163
164         eval_exp key, env, k0, (tblf val, pos)
165       when "string"
166         key = pos
167         val = head
168         pos += 1
169
170         if val == "..." and not fields[1]
```

```
171         eval_exp key, env, k0, (key) -> eval_args env[head], env, k0, (vars) ->
172             eval_tbl fields, pos, env, k0, (y) ->
173               k with y do for i = 1, #vars do y[i + key] = vars[i]
174       else (tblf val, pos) key
175
176 eval_exp = (exp, env, k0, k) -> switch type(exp)
177   when "string" then k strmguard exp, {
178     "^%d", -> tonumber exp
179     "^%.%d", -> tonumber exp
180     "^nil$", -> nil
181     "^true$", -> true
182     "^false$", -> false
183     "^[_a-zA-Z]", -> env[exp] or nil
184     "^%.%.%.$", -> eval_args env[exp][1], env, k0, (e) -> k remove e, 1
185     default: -> strip_string exp
186   }
187   when "table" then switch exp.label
188     when "constructor"
189       exp.label = nil
190
191       eval_tbl exp, _, env, k0, k
192     when "funcall" then eval_funcall exp[1], exp[2], env, k0, (t) -> k (remove t, 1)
193     when "annonymousfuncdef" then eval_funcdef exp, env, k0, k
194     when "tableaccess" then k ({expand_tbl exp, _, env, k0, noop})[1]
195     when "exp"
196       import op from exp
197
198       if exp[2] then switch exp.op
199         when "or" then eval_exp exp[1], env, k0, (do_l) ->
200           if do_l then k do_l
201           else eval_exp exp[2], env, k0, k
202         when "and" then eval_exp exp[1], env, k0, (do_l) ->
203           if do_l then eval_exp exp[2], env, k0, k
204           else k do_l
205         else eval_exp exp[1], env, k0, (left) ->
206           eval_exp exp[2], env, k0, (right) ->
207             k binop op, left, right
208       else eval_exp exp[1], env, k0, (exp) -> k uniop op, exp
209     else k exp
210   else k exp
211
212 -- TODO: local recursive function such as: local f = function...
213 eval_funcdef = (def, env, k0, k) ->
214   nenv = cp_tbl env
215   def.name or= ""
```

```
216
217   nenv[def.name] = (...) ->
218     gargs = {...}
219
220     if args = def.args
221       for i = 1, #args
222         unless args[i] == "..."
223           nenv[args[i]] = remove gargs, 1
224         else
225           nenv[args[i]] = cp_tbl gargs
226           break
227
228     funstack\pushcresume -> eval def.body, nenv, k0
229     ret = funstack\pop!
230
231     ret[2] and unpack ret[2] or nil
232
233   k nenv[def.name]
234
235 eval_args = (arglist, env, k0, k) ->
236   unless arglist[1]
237     k arglist
238   else
239     head = remove arglist, 1
240     argf = -> eval_exp head, env, k0, (x) ->
241       eval_args arglist, env, k0, (y) ->
242         _insert y, 1, x
243         k y
244
245     unless arglist[1]
246       if is_funcall head
247         eval_funcall head[1], head[2], env, k0, (x) ->
248           eval_args arglist, env, k0, (y) ->
249             k with y do for i = 1, #x do _insert y, x[i]
250       elseif head == "..."
251         eval_args env[head], env, k0, (t) -> k t
252       else argf!
253     else argf!
254
255 -- TODO redefine load and require
256 -- TODO: f():g()
257 eval_funcall = (func, args, env, k0, k) ->
258   if func == "throw" then return k0.excep args[1], env
259
260   run = (args) -> (func) ->
```

```
261     -- TODO setmetatable don't work well
262     eval_args args, env, k0, (fmt_args) ->
263       -- if #fmt_args < 1
264         -- k {func _}
265       -- else
266         -- TODO: why )k) "ignore first "nil"
267       k {func unpack fmt_args}
268
269   if type_t func
270     return switch func.label
271       when "tableaccess" then expand_tbl func, _, env, k0, (func) ->
272         ((args) -> (run args) (env[func] and env[func] or func)) with args
273           if .label == "colonfunc"
274             s = func
275             func = func[.func]
276             args = .args or {}
277             _insert args, 1, s
278       when "annonymousfuncdef" then eval_funcdef func, env, k0, run args
279       when "funcall" then eval_exp func, env, k0, (f) -> (run args) f
280
281   ((args) -> (run args) (env[func] and env[func] or func)) with args
282     if .label == "colonfunc"
283       s = func
284
285       if (type_s func) and (func\match"^\"" or func\match"^%[=*%[")
286         func = string[.func]
287       else func = env[func][.func]
288
289       args = .args or {}
290       _insert args, 1, s
291
292 -- insert all the elements which is evaled in body to ret
293 eval_return = (body, ret = {}, env, k0, k) ->
294   unless body[1]
295     if funstack[1] then k funstack\stopregret ret
296     else k ret
297   else
298     body = cp_tbl body
299     head = remove body, 1
300
301     if not body[1] and is_funcall head
302       eval_funcall head[1], head[2], env, k0, (t) ->
303         for i in *t
304           _insert ret, i
305         eval_return body, ret, env, k0, k
```

```
306     else eval_exp head, env, k0, (x) -> eval_return body, insert(ret, x), env, k0, k
307
308 -- bind one of the element in right corresponding to the name in the left to regtbl
309 -- eval_varlist {a, b, c, ...}, {"a", "b", "c", ..}, regtbl,...
310 -- ==> regtbl = a: evaled_a, b: evaled_b, c: evaled_c
311 eval_varlist = (left, right = ["" for _ = 1, #left], regtbl, env, k0, k) ->
312   f = (t) -> eval_varself left, env, k0, (varnames) ->
313     for i = 1, #varnames do switch type varnames[i]
314       when "string"
315         if (env != regtbl) and rawget env, varnames[i]
316           env[varnames[i]] = t[i]
317         else regtbl[varnames[i]] = t[i]
318       when "table" then varnames[i][1][varnames[i][2]] = t[i]
319
320   k if not right[2] and is_funcall regtbl[1]
321     eval_funcall right[1][1], right[1][2], env, k0, f
322   else eval_args right, env, k0, f
323
324 -- {{"t", "1", label: "tableaccess"}, "k", ...} ==> {{{}, 1}, "k", ...}
325 eval_varself = (varnames, env, k0, k) ->
326   unless varnames[1] then k varnames
327   else
328     var = remove varnames, 1
329     switch type var
330       when "string"
331         eval_varself varnames, env, k0, (v) ->
332           k insert v, 1, var
333       when "table" then switch var.label
334         when "tableaccess" then expand_tbl var, true, env, k0, (t, key) ->
335           eval_varself varnames, env, k0, (v) ->
336             eval_exp key, env, k0, (ke) ->
337               k insert v, 1, {t, ke}
338         else eval_varself varnames, env, k0, (v) ->
339           k insert v, 1, var
340       else eval_varself varnames, env, k0, (v) ->
341         k insert v, 1, var
342
343 -- k0 ... table(excep(try-catch), loop(while/for/repeat))
344 eval_body = (body, env, k0, k) -> switch body.label
345   when "try"
346     k1 = loop: k0, excep: (args, env) ->
347       eval_exp args, env, k0, (t) ->
348         env._ERR = t
349         eval body.catchbody, env, k0, noop
350     k eval body.body, env, k1, (e) ->
```

```
351       for k, v in pairs e do env[k] = v if env[k]
352   when "do" then k eval body, (cp_tbl env), k0, (e) ->
353     for k, v in pairs e do env[k] = v if env[k]
354   when "return" then eval_return body, _, env, k0, k
355   when "break" then k0.loop!
356   when "varlist" then eval_varlist body[1], body[2], __ENV, env, k0, k
357   when "localvarlist" then eval_varlist body[1], _, env, env, k0, ->
358     eval_varlist body[1], body[2], env, env, k0, k
359   when "funcall" then eval_funcall body[1], body[2], env, k0, k
360   when "tableaccess" then expand_tbl body, _, env, k0, noop
361   when "funcdef" then k eval_funcdef body, env, k0, (f) -> __ENV[body.name] = f
362   when "localfuncdef" then k eval_funcdef body, env, k0, (f) -> env[body.name] = f
363   when "while"
364     whilef = (env) -> eval_exp body.cond, env, k0, (cond) ->
365       cond and (eval body.body, env, {loop: k, excep: k0.excep}, ((env) -> whilef(env))) or k!
366     whilef env
367   when "repeat"
368     repf = (env) -> eval body.body, env, {loop: k, excep: k0.excep}, (env) ->
369       eval_exp body.cond, env, k0, (cond) -> if cond then k! else repf env
370     repf env
371   when "for"
372     import var, cnt, to, step, body from body
373
374     eval_exp cnt, env, k0, (cnt) ->
375       env[var] = cnt
376       step or= 1
377
378       eval_exp step, env, k0, (step) -> eval_exp to, env, k0, (forend) ->
379         forf = (e1) ->
380           unless e1[var] > forend
381             eval body, e1, {loop: k, excep: k0.excep}, (e2) ->
382               cnt += step
383               e2[var] = cnt
384               for k, v in pairs e2 do env[k] = v if env[k]
385               forf e2
386           else k!
387         forf env
388   when "iter"
389     init = {}
390     import namelist, explist, body from body
391
392     eval_varlist {"_f", "_s", "_var"}, explist, init, env, k0, ->
393       import _f, _s, _var from init
394
395       -- TODO: update environment every loop, also iter
```

```
396       -- XXX: DON'T update explist
397       iterf = (_var) -> (env) ->
398         eval_funcall _f, {_s, deeval_str _var}, env, k0, (res) ->
399           eval_varlist namelist, [deeval_str x for x in *res], env, env, k0, ->
400             if _var = env[namelist[1]]
401               eval body, env, {loop: k, excep: k0.excep, fun: k0.fun}, iterf _var
402             else k!
403
404       (iterf _var) env
405     when "if"
406       import cond, body, elsebody from body
407
408       eval_exp cond, env, k0, (cond) ->
409         if cond then eval body, env, k0, k
410         elseif elsebody then eval (elsebody[1].label == "if" and elsebody or elsebody[1]), env, k0, k
411         else k!
412
413 kinit =
414   loop: -> error "not in loop"
415   excep: -> error "call throw out of try-catch"
416
417 eval = (syntaxtree, env = {}, k0 = (cp_tbl kinit), k = noop) ->
418   unless syntaxtree[1]
419     k env
420   else
421     syntaxtree = cp_tbl syntaxtree
422     env = setmetatable (cp_tbl env), __index: __ENV, __mode: 'kv'
423     eval_body remove(syntaxtree, 1), env, k0, -> eval syntaxtree, env, k0, k
424
425 eval
```

付 録 C　main

Code C.1: llix/llix

```
#!/usr/bin/env moon

-- getopt = require'alt_getopt'
parse = require"parse"
eval = require"eval"

lazy_gc_stop = (...) =>
  collectgarbage "stop"
  with {@ ...}
    collectgarbage "restart"

-- read file and run
if fn = arg[1]
  fileread = (fl) ->
    fh = assert io.open fl
    with fh\read '*a'
      fh\close!

  t, err = parse "do #{fileread fn} end"
  unless t then error "#{err} #{fn}"

  cont = lazy_gc_stop pcall, eval, t
  if cont[1] == false then error cont[2]

  os.exit!

L = require'linenoise'
inspect = require'inspect'
iprint = (...) -> print (inspect {...})\match("^{ (.*) }$")
import remove, insert, concat from table

-- add cadidates for completion
candidates = (line, env) ->
  i1 = line\find '[.\\%w_]+$'

  unless i1 then return

```

```
  front = line\sub 1, i1 - 1
  partial = line\sub i1
  with res = {}
    prefix, last = partial\match '(.-)([^.\\]*)$'

    if #prefix > 0 then for w in prefix\sub(1, -2)\gmatch '[^.\\]+'
      env = env[w]
      unless env then return

    prefix = front .. prefix
    append_candidates = (env) ->
      for k in pairs env do if (last == '') or k\sub(1, #last) == last
        insert(res, prefix..k)

    if type(env) == 'table' then append_candidates env

    with mt = getmetatable env do if mt and type(.__index) == 'table'
      append_candidates .__index

vars = {}
completion_handler = (env) -> (c, s) ->
  if cc = candidates s, env
    for name in *cc do L.addcompletion c, name
  for k in *vars
    L.addcompletion c, k

-- parse and eval
parseval = (obj) ->
  collectgarbage "stop"

  if type(obj) == "table"
    ok, cont = pcall eval, obj
    if ok
      not obj.quiet and iprint cont._llix_tmp
      eval parse"_llix_tmp = nil"
    else print"failed to parse"
  else
    tree = parse obj
    ok, cont = pcall eval, tree

    unless ok then print cont
    else
      if tree[1].label == "varlist" then vars = tree[1][1]

  collectgarbage "restart"
```

```
83
84  ---- repl body
85  histfile = os.getenv"HOME" .. "/.llix_history"
86  block = {}
87  prompt =
88    p: ">"
89    deepen: => @p = @p\rep 2
90    reset: => @p = ">"
91
92  get_line = ->
93    with line = L.linenoise prompt.p .. " "
94      if line and line\match '%S' then L.historyadd line
95
96  L.setcompletion completion_handler(_G)
97
98  io.write "llix - Lightweight Lua Interpreter eXtended ",
99    "(MoonScript version #{(require 'moonscript.version').version}",
100   " on #{_VERSION})\n\n"
101
102 unless L.historyload histfile
103   io.stderr\write "failed to load commandline history\n"
104
105 while true
106   line = get_line!
107
108   if not line then break
109   elseif #line < 1 then continue
110
111   if t = parse line
112     t.quiet = true
113     parseval t
114   elseif tt = parse "_llix_tmp = #{line}"
115     parseval tt
116   else
117     prompt\deepen!
118     insert block, line
119
120     while line and #line > 0
121       line = get_line!
122       insert block, line
123
124     prompt\reset!
125     parseval concat block, '\n'
126     block = {}
127
```

```
128 unless L.historysave histfile
129   io.stderr\write "failed to save commandline history\n"
```