Luaプログラミング入門

Nishio

目 次

第1章　はじめに

1.1　対象読者

本書の対象読者はC言語プログラミング（またはC++, C#, JAVA）経験者です．言語の基本的な文法を知っている（if, for, while, 関数の作り方を知っている）程度で十分です．しかしできるだけC言語 (またはC言語に似た言語）を知らない方でも理解できるように説明していくつもりです.

1.2　Lua言語について

本書の対象読者をC（またはCに似た）言語としたのには理由があります．それはたぶんLua言語のみを使ってプログラムを作ることが殆ど無いと思われるからです．Lua言語はC言語から呼び出されて，または組み込まれて使われる言語です.

以下は今度書く

1.3　開発環境を整える

第2章　Lua言語の基礎

この章では画面に Hello World と表示するプログラムを作成していきたいと思います．同時にここで Lua 言語の基礎を学んでいきます．

2.1　画面に文字を表示する

次のソースコード[1]をエディタに貼り付けてください．

```
1   print ("Hello World!")
```

プログラムは 1 行だけです．C 言語とは違い main 関数はありません．このプログラムの実行結果は以下のようになります．

実行結果

Hello World!

画面に文字を表示する時は print 関数を使用します．

書式

print (” なにか表示したい文章をここに書く ”)

print 関数は以下のように書くこともできます．

```
print ("Hello World!")
print ('Hello World!')
print "Hello World!"
print ([[Hello World!]])
```

上から 3 つまでのコードはすべて同じ意味です．文字列をダブルクォーテーションで囲んでもシングルクォーテーションで囲んでもどちらでもよいです．ただし 4 番目だけは少し意味が違います．これは次節のエスケープシークエンスで説明します．

[1] ソースコードとはプログラミング言語で書かれた文章の事です．

C 言語との違い

- プログラムはメイン関数から実行されるわけではなく，上から下に実行される
- printf ではなく print
- コードの終端にセミコロン (;) が無い

2.2 エスケープシークエンス

たとえば，print 文の中の文字列で，改行をしたい場合があるかもしれません．改行を行う場合，特殊な文字を使って改行を行います．改行の場合は\n という文字を使います．\と n 2 つの文字を組み合わせて改行という意味です．[2]

```
print("Hello\nWorld!")
```

このようにして使います．

実行結果

```
Hello
World!
```

改行文字のほかに，表 2.1 がエスケープシークエンスとしてあります．

表 2.1: エスケープシークエンス一覧

記号	意味
\n	改行
\a	BEEP 音（警告音）
\t	タブ
\b	バックスペース
\’	シングルクォーテーション
\"	ダブルクォーテーション
\\	バックスラッシュ（日本語の端末の場合, 円マーク）

さて，先ほど出てきた print ([[Hello World!]]) という print 関数の書き方ですが，これはエスケープ文字を無視して文字列をそのまま出力します．例えば次のようなコードがあったとします．

[2] 日本語の場合バックスラッシュは円マーク ¥ を使います．

```
print([[Hello\nWorld!]])
```

このコードの実行結果は以下のようになります.

実行結果

```
Hello\nWorld!
```

文字列をそのまま出力するので，改行などもそのまま表示されます．例えば次のような場合です.

```
1  print([[Hello
2  World!
3   こんにちは世界
4  ]])
```

このプログラムの実行結果は以下のようになります.

実行結果

```
Hello
World!
こんにちは世界
```

2.3 コメント

C 言語同様プログラム内にコメントを書くことができます.

コメントとは，ソースコード（自分がC言語で書いた文章の事）の中で，そのソースが何を意味しているかを書いておきたいときに使います．たとえば次のようなソースがあったとします.

```
1  print ("Hello World!") --Hello World!と画面に表示する
```

「--Hello World!と画面に表示する」という部分がコメントです．マイナスを二つ重ねています.「--」以降の行末までコメントとなります.

ここで，複数行コメントを書きたい場合があると思います.

```
--この
--部分が
--コメントと
--なります
```

これでもよいですが，Lua では複数行のコメントを書く場合は，--[[と]] で囲むことでコメントとすることができます．

```
--[[
この
部分が
コメントと
なります
]]
```

2.4 例題

2.4.1 コメントの入ったプログラム

このプログラムで注意したい点は「This line is not displayed.」という部分が実行結果には表示されていないことです．コメントアウトされている部分はたとえ命令であっても無視されます．また，C 言語の printf とは違い，print 関数を一度実行すると，文章の終わりに自動的に改行が付け加えられます．

```
1  --print 関数の終わりは勝手に改行される
2  print("This is a test program ")
3  print("without indented. ")
4
5  --[[
6  この部分はコメントアウトされているので表示されない
7  print("This line is not displayed. ")
8  ]]
```

実行結果

```
This is a test program
without indented.
```

2.4.2 長文を表示するプログラム

一行で表示する文字数が多い場合それを同じライン上で書くとコードが読みにくくなる場合があります．この場合の修正方法を学んでいきましょう．

print 関数を一回だけ使い，There are 10 types of people in this world. Those who understand binary and those who don't. Which one are you? という文章を

表示するとしましょう．しかし同じライン上で書くとコードが読みにくくなってしまいます．この場合以下のようにコードを書くこともできます．

```
1  --この用にコードを書くこともできる
2  print("There are 10 types of people in this world." ..
3          "Those who understand binary and those who don't." ..
4          "Which one are you?" )
```

.. という部分に注目してください．これは複数の文字列をつなぎ合わせるときに使用します．詳細は次章で説明します．

実行結果

There are 10 types of people in this world.Those who understand binary and those who don't.Which one are you?

2.4.3 ダブルクォーテーションを表示するプログラム

次のような文字列を表示するプログラムを作りたいとします．

He said, "How dare you do to me like that!"

この場合次のようなコードを作成してみましたがうまくいきませんでした．

```
print("He said, "How dare you do to me like that!"")  --error
```

このコードはエラーが出て実行することができません．ではどのようにすれば"(ダブルクォーテーション）を表示されるのでしょうか？
その答えは先ほども登場したエスケープシークエンスを使うことで解決できます．

```
print("He said, \"How dare you do to me like that!\"")
```

または次のように書いてもよいでしょう．

```
print([[He said, "How dare you do to me like that!"]])
```

このようにできることも覚えておきましょう．

実行結果

He said, "How dare you do to me like that!"

2.5 演習問題

問題 1

自分の名前を画面上に表示するプログラムを作成せよ.

問題 2

ビープ音を鳴らすプログラムを作成せよ.

問題 3

print 関数を一度だけ使い次の文章を表示するプログラムを作成せよ. ただしソースコードが読みやすくなるように工夫すること.

The ICMP source quench message is the TCP/IP equivalent of telling another computer: ”I can’t keep up with all the traffic you’re sending me - slow down, please.” [3]

[3]Firewalls FOR DUMMIES by Brain Komar, Ronald Beekelaar, and Joern Wettern, PhD より引用

第3章　変数について

何か計算をしたとき，その計算結果を覚えておきたいときがあると思います．その時使うものが変数です．変数とは，値を保存することができる箱です．

3.1　変数の型について

Lua には C 言語のような変数の型は一応存在してはいますが，変数の型を意識しなくてもよいような設計となっています．例えば hoge という名前の変数を宣言したら，その変数にはどんな値でも代入できます．例えば整数も代入できますし，少数や文字列なども代入できます．ここで代入とは変数つまり箱の中に値を入れることです (この表現は正確ではないですが，いまの段階ではそう思っていただいてかまいません) .

C 言語との違い

- 変数の型を意識しなくてもよい

3.2　変数の作り方

では，実際変数を作ってみましょう．変数を作りたい場合，次のようにします．

```
hoge = 10
```

これは，hoge という名前の変数を作れという意味です．箱には名前をつけることができます．そしてこの hoge には 10 という値が入っています．C 言語とは違って初期化していない変数は作ることができません．例えば次のようなコードはエラーとなります．

```
hoge --error
```

変数を複数宣言する場合は次のように記述することもできます．

```
foo = 10 bar = 20
```

上記の宣言は

```
foo = 10
bar = 20
```

と全く同じ意味になります．また,

```
foo, bar = 10, 20
```

も同じ意味となります.

C 言語とは違い，**変数はどこでも宣言できます**．また，複数の値を 1 行で書き換える多重代入が可能です.

また，これも重要ですが，Lua の変数は特に指定が無い限り**グローバル変数**となります．グローバル変数の意味は変数のスコープの所で説明します.

C 言語との違い

- 変数はいかなるところでも定義できる
- 変数は指定が無い限りグローバル変数となる

3.3 変数命名の際の注意

変数名を付ける際に，次のようなルールがあります.

- 先頭は英字か_(アンダースコア）でなければならない

  数字ではダメです.

- 同じ名前の変数が 2 つ存在する事は基本的にはできない

  ただし変数のスコープが違えば同じ名前の変数を複数つくれます．スコープの説明は今はしません.

- 予約語は使えない

  予約語 (表 3.3）とは，if や while など，Lua 言語の文法の一部の事です.

表 3.1: 予約語一覧

and	break	do	else	elseif	
end	false	for	function	if	
in	local	nil	not	or	
repeat	return	then	true	until	while

3.4 変数への値の代入

変数に何か値を入れてみましょう．例えば，10 という値を箱に入れたい場合は次のようにします．

```
hoge = 10
```

この = という記号は，式の右側の値を左側に代入せよ，という意味です．数学のイコールという意味ではないので注意してください．

では，数学でよくある x = 10（x は 10 である）といった本当にイコールの意味を使いたい場合はどうすればよいでしょうか．これは，if 文という所でしっかり説明しますが，今軽く説明しておくと，== という記号を使います．イコールを二つ繋げて書くと，本当に数学でいうイコールという意味になります．

Lua では多重代入も可能となっています．

```
hoge, piyo = 10, 5
```

hoge に 10 を代入し，piyo には 5 を代入しています．

3.5 変数の値を表示する

変数を画面上に出力したい場合，どのようにしたらよいでしょうか．例として，int 型の hoge という箱に 10 という値を代入して，その箱の中身を画面上に表示させてみます．

```
hoge = 10
print(hoge)
```

このプログラムを実行すると，次のように画面上に表示されると思います．

実行結果

```
10
```

複数の値を表示させることもできます．

```
hoge = 10
piyo = 20
print(hoge, piyo)
```

このようにカンマ区切りで複数の値が表示できます.

実行結果

10 20

また文字と数値なども一緒に表示させることができます．文字列と数値，または文字列と文字列を結合して表示させる場合，連結演算子 .. を使います.

```
1  hoge = 10
2  print("hoge の値は" .. hoge .. "です")
```

実行結果

hoge の値は 10 です

Lua にはC言語の printf関数と似たものが存在します．それが string.format() 関数です．この関数は文字列の書式を指定して画面に表示します．書式指定子一覧は表 3.2 の通りです.

表 3.2: 書式指定子

記号	意味
%d	整数の 10 進法として出力
%u	符号なし整数の 10 進法として出力
%o	整数の 8 進法として出力
%x	整数の 16 進法として出力
%f	小数点表示
%c	1 文字出力
%%	%を出力

string.format() 関数を使って数値を表示させてみましょう.

```
1  hoge = 10
2  print( string.format( "hoge の 10 進数の値は%d です" ,hoge) )
3  print( "hoge の 16 進数の値は" ..
4           string.format("%x",hoge) .. "です")
```

実行結果

hoge の 10 進数の値は 10 です
hoge の 16 進数の値は a です

他にもいろいろ試して見ましょう.

```
1   hoge = 10
2   piyo = 20
3   foo = 30.256
4
5   print( string.format("hoge の値は%d です",hoge) )
6   print( string.format("piyo の値は%d です",piyo) )
7   print( string.format("値は%d です", 50) )
8   print( string.format("hoge は%d piyo は%d です",hoge,piyo) )
9   print( string.format("foo の値は%f です",foo) )
10  print( string.format("hoge の値は%f です",hoge) )
11  print( string.format("foo の値は%d です",foo) )
```

実行結果

hoge の値は 10 です
piyo の値は 20 です
値は 50 です
hoge は 10 piyo は 20 です
foo の値は 30.256000 です
hoge の値は 10.000000 です
foo の値は 30 です

3.6 演算

3.6.1 四則演算

数学でもお馴染みの四則演算，つまり「足し算」「引き算」「掛け算」「割り算」を Lua 言語でも行うことができます．その方法は簡単です (表 3.7) .

表 3.3: 四則演算

記号	意味
+	足し算
−	引き算
*	掛け算
/	割り算

簡単な計算の例

```
1   x = 10
2   y = 20
3   print("x+y は" .. x + y .. "です")
4   y = y - x
5   print("y の値は" .. y .. "です")
```

このプログラムの実行結果は次のようになります.

実行結果

```
x+y は 30 です
y の値は 10 です
```

3.6.2 その他の演算

値の符号の反転をしたい場合は次のようにします.

```
x = 10
print("反転した値は" .. -x .. "です")
```

実行結果

```
反転した値は-10 です
```

剰余，つまり割り算の余りを求めたい場合は，次のようにします.

```
x = 10 y = 3
print("余りは" .. x % y .. "です")
```

実行結果

```
余りは 1 です
```

表 3.4: 変数の型

型	意味
nil	nil
boolean	論理型
number	数値
string	文字列
function	関数
userdata	ユーザ定義型
thread	スレッド
table	テーブル

3.6.3 べき乗の計算

べき乗とは，例えば x の 2 乗，つまり x*x などのことをいいます．Lua でべき乗の計算をしたい場合， `2^3` と `^` を使って表現します． `2^3` は 2 の 3 乗という意味です．

```
print( "2 の 3 乗は " .. 2 ^ 3 .. "です")
```

実行結果

```
2 の 3 乗は 8 です
```

3.7 動的な変数の型

Lua は変数の型を意識しなくてもよいと説明しました．しかし一応変数の型は存在しています．それは表 3.7 に示した 8 つの基本型です．

nil とは何もない値という意味です．C 言語や JAVA でいう null と同じ意味です．boolean は論理型，つまり true と false の二つの値を持つものです．number は数値を入れることができるものです．string が文字列を表しています．その他の型については現時点では説明しません．

Lua の場合は変数が型を持っているわけではなく，値が型を持っています．つまり，変数に入れる値によって型が動的に変化していくわけです．ここで現在変数が何型であるかを調べるため type() 関数を使用して，型を調べてみましょう．

```
1   hoge = nil
```

```
2   print("hoge の現在の型は" .. type(hoge) .. "です")
3   hoge = "Hello"
4   print("hoge の現在の型は" .. type(hoge) .. "です")
5   hoge = 33
6   print("hoge の現在の型は" .. type(hoge) .. "です")
7   hoge = true
8   print("hoge の現在の型は" .. type(hoge) .. "です")
```

実行結果

hoge の現在の型は nil です
hoge の現在の型は string です
hoge の現在の型は number です
hoge の現在の型は boolean です

このように入れる値によって型が動的に変化します.

3.8 キーボードからの値の読み込み

キーボードから何か値を入力して，変数の中に代入したい場合があると思います．そういった場合に使うのが io.read() 関数です.

```
1   hoge = nil
2   print("何か文字列を入力してください :")
3   hoge = io.read()
4   print("入力された文字は" .. hoge .. "です")
```

実行結果

何か文字列を入力してください :
hello world
入力された文字は hello world です

実行結果

何か文字列を入力してください :
55
入力された文字は 55 です

3.9 例題

3.9.1 メートルをフィートへ変換するプログラム

仕様

メートルからフィートへ変換するプログラムを作成せよ．メートルに 3.2 をかけることによってフィートを求めることができる．

```
1  io.write("メートルの値を入力してください :")
2  meter = io.read()
3  answer = meter * 3.2
4  print( meter .. "メートルは" .. answer .. "フィートです")
```

io.write() 関数は print 関数と使い方は同じです．ただし io.write() 関数は文章の最後に改行が自動付加されません．

実行結果

```
メートルの値を入力してください :27
27 メートルは 86.4 フィートです
```

3.9.2 2 つの値を交換するプログラム

仕様

変数 hoge には 10 が，piyo には 5 が代入されている．この二つの値を交換せよ．

```
1  hoge, piyo = 10, 5
2  print("hoge の値は" .. hoge .. " piyo の値は" .. piyo .. "です")
3  tmp = hoge
4  hoge = piyo
5  piyo = tmp
6  print("hoge の値は" .. hoge .. " piyo の値は" .. piyo .. "です")
```

解答は一通りではありません．多重代入を使う事で tmp 変数を使わなくて済みます．

```
1  hoge, piyo = 10, 5
2  print("hoge の値は" .. hoge .. " piyo の値は" .. piyo .. "です")
3  hoge, piyo = piyo, hoge
4  print("hoge の値は" .. hoge .. " piyo の値は" .. piyo .. "です")
```

実行結果

hoge の値は 10 piyo の値は 5 です
hoge の値は 5 piyo の値は 10 です

3.10 演習問題

問題 1

分を時間に変換するプログラムを作成しなさい．例えば，ユーザが 180 と入力したら 3 と表示する表示するプログラムである．

問題 2

セ氏温度を華氏温度に変換するプログラムを作成せよ．セ氏温度 (F) と華氏温度 (C) には次のような関係がある．

$$F = \frac{9}{5}C + 32 \tag{3.1}$$

第4章　制御文

4.1　if else による条件分岐

if else 構文 (英語で if はもし〜だったら，else はそうでなかったら，という意味) は, 条件分岐をする際に使用する構文です.

4.1.1　if else の書き方

書式

```
if （ 条件式 ） then
    条件式が真であるときの処理
else
    条件式が偽であるときの処理
end
```

条件式にはカッコが無くても構いません.

では, 実際に if else 構文を使ってみましょう.

ここでテストの成績を判定するプログラムを作ってみましょう．もし点数が 60 点以上であったら合格を，そうでなかったら不合格を表示するプログラムを作ります.

```
1   result = 70
2   if result >= 60 then
3           print("合格")
4   else
5           print("不合格")
6   end
```

実行結果

```
合格
```

result が 60 以上であるかを比較する際には，関係演算子>=を使用します.

result の値を 50 にしてみると次のようになります.

実行結果

不合格

else は書かなくてもかまいません.

```
1   hoge = 10
2   if ( hoge == 10 ) then
3           print("Hello world")
4   end
```

このプログラムは hoge が 10 と等しい時にだけ Hello world を出力します.

ここで注意して欲しいのは, hoge が 10 と等しいと判定する場合，hoge = 10 ではなく,hoge == 10 であるということです.

また, 条件を複数書きたい場合, 次のように書きます.

```
1   hoge = 8
2
3   if( hoge == 10 ) then
4           --A の処理
5   elseif( hoge == 9) then
6           --B の処理
7   elseif( hoge == 8 ) then
8           --C の処理
9   else
10          --D の処理
11  end
```

第 2，第 3 の判定を付けたい場合，elseif を使用します．この場合 C の処理が実行されることになります．ここでもし hoge が 9 であるならばBの処理が行われますし,6 ならば D の処理が実行されることになります.

4.1.2 偽条件の判定

さて，次のようなコードを書いたらどのような実行結果となるでしょうか.

```
1   hoge = 1
2   if ( hoge ) then
3           print("first")
```

```
4   end
5
6   hoge = 0
7   if ( hoge ) then
8           print("second")
9   end
10
11  hoge = false
12  if ( hoge ) then
13          print("third")
14  end
15
16  hoge = nil
17  if ( hoge ) then
18          print("fourth")
19  end
```

実行結果

first
second

さて, カッコの中には何か式を書かなくてはなりませんが, この場合は変数名の hoge としか書いてありません. これはどういう意味でしょうか.

実はこういった場合, カッコの中の値が false,nil であるか false,nil 以外であるかで判定します. false,nil でない場合は処理を行い，そうでない場合は処理を行いません.

C 言語との違い

- 0 は偽条件ではない

4.1.3 関係演算子

2 つの変数を比較して, 式が正しいか誤りかを求める時に使う演算子を関係演算子といいます. 例えば 4.1 のようなものが存在しています.

例えば次のようなコードを書いたとします.

表 4.1: 関係演算子

x < y	x が y より小さかったら真, そうでなかったら偽
x > y	x が y より大きかったら真, そうでなかったら偽
x <= y	x が y 以下であれば真, そうでなかったら偽
x >= y	x が y 以上であれば真, そうでなかったら偽
x == y	x が y と等しければ真, そうでなかったら偽
x ~= y	x が y と等しくなければ真, そうでなかったら偽

```
1  hoge = 1
2  if( hoge ~= 1 ) then
3          print("Hello")
4  else
5          print("World")
6  end
```

この場合，hoge は 1 なので，条件が偽となり，World が表示されます.

C 言語との違い

- != ではなく ~=

4.1.4 論理演算子

次のようなコードを書いたとします.

```
1  hoge, piyo = 10, 20
2
3  if( hoge == 10 ) then
4          if( piyo == 20 ) then
5                  print("hello")
6          end
7  end
```

この式は hoge が 10 であり, かつ piyo が 20 である時,Hello が表示されます．ここで論理演算子というものが存在します (表 4.2）.

つまり, 先ほどの例をこの論理演算子を使って書き直すと次のようになります.

表 4.2: 論理演算子

x == 10 and y == 20	x が 10 かつ y が 20
x == 10 or y == 20	x が 10 または y が 20
not x	x の否定

```
1   hoge, piyo = 10, 20
2
3   if( hoge == 10 and piyo == 20) then
4           print("Hello")
5   end
```

ここで,x の否定とはどういうことを意味しているでしょうか.

```
1   hoge = 10
2   if( not hoge ) then
3           print("Hello")
4   else
5           print("World")
6   end
```

not hoge は hoge が 10 でない場合は条件が真となり，そうでない場合は条件が偽となります．よってこのプログラムは条件が偽となり，World と表示されます.

C 言語との違い

- &&, ||, ! ではなく and, or, not

4.2 while による繰り返し

書式

```
while ( 条件式 ) do
    処理
end
```

条件式にはカッコが無くてもかまいません.

繰り返し処理を行いたい場合は while 構文を使います．条件式が真の場合，処理が繰り返し行われます．たとえば Hello world と 10 回表示するプログラムを while を使って作ってみます.

```
1   i = 1
2   while i <= 10 do
3           print ( i .. "回目:Hello world!" )
4           i = i + 1
5   end
```

実行結果

```
1 回目:Hello world!
2 回目:Hello world!
3 回目:Hello world!
4 回目:Hello world!
5 回目:Hello world!
6 回目:Hello world!
7 回目:Hello world!
8 回目:Hello world!
9 回目:Hello world!
10 回目:Hello world!
```

条件式は if と同じです. if と同じということは次のようなコードを書いた場合どうなるでしょうか.

```
1   while true do
2           print ( "Hello world!" )
3   end
```

これは, 条件は常に真ということになります．つまり無限ループに陥ります.

4.3 for による繰り返し

for 構文には Numeric for と Generic for の 2 種類が存在します．Generic for は難しいので，また後の章で説明することとします．ここでは Numeric for について説明していきます.

Numeric for も while 同様繰り返しの処理を行う場合に使用します．ただし，while とは違い，終了条件が数値でしか指定できません.

書式

```
for 初期値，終了値，増加量 do
    処理
end
```

増加量は省略することができます．その場合，自動的に増加量は 1 となります．

前節で登場した Hello world を 10 回表示するプログラムを for を使って書き直すと次のようになります．

```
1   for i = 1, 10, 1 do
2           print( i .. "回目:Hello world!")
3   end
```

i が 1 で始まり，i が 10 になるまで，i を 1 ずつ増加させていきます．つまり，i が 10 以下の場合に繰り返しが行われます．

実行結果

```
1 回目:Hello world!
2 回目:Hello world!
3 回目:Hello world!
4 回目:Hello world!
5 回目:Hello world!
6 回目:Hello world!
7 回目:Hello world!
8 回目:Hello world!
9 回目:Hello world!
10 回目:Hello world!
```

増加量は省略することができます．その場合は，自動的に 1 が割り当てられます．

```
1   for i = 1, 10 do
2           print( i .. "回目:Hello world!")
3   end
```

実行結果は先ほどと同じです．

増加量を 4 に変更してみましょう．どのような実行結果となるでしょうか．

```
1   for i = 1, 10, 4 do
2           print( i .. "回目:Hello world!")
3   end
```

実行結果

```
1 回目:Hello world!
5 回目:Hello world!
9 回目:Hello world!
```

for を使う上で注意しなければならないことがあります．それは，初期値で定義した変数（ループ変数）は for 文の中でのみ有効だということです．つまり，for を抜けた時点でループ変数は消滅してしまいます．

例えば次のようなコードを書いたとします．

```
1   for i = 1, 10, 4 do
2           print( i .. "回目:Hello world!")
3   end
4   if i == nil then
5           print ("i は定義されていません")
6   end
```

for 文で使用している i と if 文で使用している i は別物です．実行結果は以下のようになります．

実行結果

```
1 回目:Hello world!
5 回目:Hello world!
9 回目:Hello world!
i は定義されていません
```

i は定義されていないので，nil となっています．

4.4 repeat による繰り返し

repeat を使ってループを行うこともできます．repeat は while と違い，repeat 内の処理は最低 1 回は必ず行われます．処理を行ったあと，条件式を判定し，条件が真の場合，ループが終了します．

書式

```
repeat
    処理
until（条件式）
```

条件式が偽の場合，処理が繰り返し行われます．真の場合ではないので注意してください．

```
1   i = 1
2   repeat
3           print (i .. "回目:Hello world!")
```

```
4           i = i + 1
5   until ( i >= 5 )
```

実行結果

```
1 回目:Hello world!
2 回目:Hello world!
3 回目:Hello world!
4 回目:Hello world!
```

4.5 break を使ったループの脱出

繰り返し処理を実行している間に break 文を使うと，いつでもループから脱出することができます．例えば次のようなプログラムを書いたとします．

```
1   for i = 0, 10 do
2           print( "i の値:" .. i )
3           if ( i == 5 ) then
4                   break
5           end
6   end
7   print("脱出")
```

実行結果

```
i の値:0
i の値:1
i の値:2
i の値:3
i の値:4
i の値:5
脱出
```

break は必ずブロックの最後でしか使うことができません．ブロックの最後とは，具体的に以下のような場所です．

- end の手前
- repeat - until の until の手前 等

上記の場所以外では break は定義できません．しかしデバッグ中に，例えば処理の途中で break したい場合があるかもしれません．次のようなコードがあったとします．

```
1   for i = 0, 10 do
2           print("なにか処理")
3           --本当はここで break したい
4           print("別の処理")
5   end
6   print("脱出")
```

まあ，このようなコードを書くことは普段ないとは思いますが．

とにかく，途中で break したい場合は，ブロックを for の途中に作ればよいのです．ブロックを作るには，do - end を使用します．

書式

```
do
    処理
end
```

よって，次のように書き直せば，処理の途中で break が行えます．

```
1   for i = 0, 10 do
2           print("なにか処理")
3           do
4                   break;
5           end
6           print("別の処理")
7   end
8   print("脱出")
```

実行結果

```
なにか処理
脱出
```

4.6 Lua に存在しない制御文

多くのプログラミング言語には存在していても，Lua には存在していない制御文があります．

例えばswitch 構文はLua にはありません．switch はテーブルを使うことで表現できますが，ここではその方法は割愛します．

また，繰り返し処理中で使用する continue 文も存在しません．

4.7 例題

4.7.1 九九表を作成するプログラム

仕様

九九を表示するプログラムを作成せよ．九九表は以下のように表示させる．

```
 1   2   3   4   5   6   7   8   9
 2   4   6   8  10  12  14  16  18
 3   6   9  12  15  18  21  24  27
 4   8  12  16  20  24  28  32  36
 5  10  15  20  25  30  35  40  45
 6  12  18  24  30  36  42  48  54
 7  14  21  28  35  42  49  56  63
 8  16  24  32  40  48  56  64  72
 9  18  27  36  45  54  63  72  81
```

```
1  for i = 1, 9 do
2          for j = 1, 9 do
3                  io.write( string.format("%3d", i*j) )
4          end
5          io.write("\n")
6  end
```

io.write 関数は print 関数と同じ機能を果たします．ただし，文章の最後に改行が自動付加されません．

string.format の中に%3d というものを使いましたが，これは 3 文字分のスペースを確保して表示するという意味です. このようにすると綺麗に表示することができます．5 文字分のスペースを確保したい場合は,%5d とすればよいです．

4.7.2 文字コード表を表示するプログラム

仕様

0〜255 までの文字コードを表示するプログラムを作成せよ．10 進法の値，16 進法の値，文字を表示することとする．

```
0 0
1 1

（略）

65 41 A
66 42 B

（略）
```

```
1  for hoge = 0, 255 do
2          print( string.format("%3d %3x %3c", hoge, hoge, hoge) )
3  end
```

4.7.3 足し算，引き算を行うプログラム

仕様

利用者の選択により足し算, 引き算ができるプログラムを作成せよ．

```
1  print("1 : 足し算を行う")
2  print("2 : 引き算を行う")
3  io.write("番号を入力してください :")
4  select = io.read()
5
6  io.write("1 番目の数 :")
7  hoge = io.read()
8  io.write("2 番目の数 :")
9  piyo = io.read()
10
11 if( select == "1" ) then
```

```
12          print( hoge .. " + " .. piyo .. " = " .. hoge + piyo )
13   elseif( select == "2" ) then
14          print( hoge .. " - " .. piyo .. " = " .. hoge - piyo )
15   end
```

注意点が一つあります．それは，io.read 関数で読み取ることができるのは文字です．よって，select は String 型となっています．なので，

```
if( select == 1 ) then
```

とはできません．

実行結果

1 : 足し算を行う
2 : 引き算を行う
番号を入力してください :1
1 番目の数 :15
2 番目の数 :10
15 + 10 = 25

実行結果

1 : 足し算を行う
2 : 引き算を行う
番号を入力してください :2
1 番目の数 :15
2 番目の数 :10
15 - 10 = 5

4.8 演習問題

問題 1

キーボードから数字を受け取り, その絶対値を表示するプログラムを作成せよ．絶対値とは数字 0 からの距離のことである．例えば，-7 ならば 0 からの距離は 7 なので絶対値は 7 である．

第5章　関数

5.1　関数について

Lua に限らず，多くのプログラミング言語に欠かせない概念の一つとして関数というものが存在します．実は今まで関数を何回も使っていました．例えば print ですが，これは関数です．io.read も関数です．関数とはいくつもの命令を集め 1 つの塊として定義したものです．同じ処理を何回も行いたい場合は，関数を 1 つ定義して呼び出すだけでよいです．

5.2　関数の作り方

では，関数を作ってみましょう．関数は次のように定義します．

書式

```
function 関数名 ( 引数 )
    処理
end
```

ここで，二つの値を引数として渡し，その和を求める関数を作ってみましょう．

```
1  function sum(x, y)
2          return x + y
3  end
4
5  hoge = 10
6  piyo = 20
7
8  result = sum( hoge, piyo )
9  print( hoge .. " + " .. piyo .. " = " .. result )
```

実行結果

```
10 + 20 = 30
```

まず sum 関数を使う前に，関数を定義しておかなければならない事に注意してください．関数には好きな名前をつけてください．ただし，すでに使われている関数名や変数名は使わないでください．例えば，print などの名前をつけないでください．関数命名のルールは変数の場合と同じです．

C 言語との違い

- プロトタイプ宣言は無い

さて，関数を呼び出す際に `sum( hoge, piyo )` としました．関数を呼び出す際には引数と呼ばれるものを渡します．この引数とはどういったものでしょうか．

図 5.1 を使ってその説明をします．

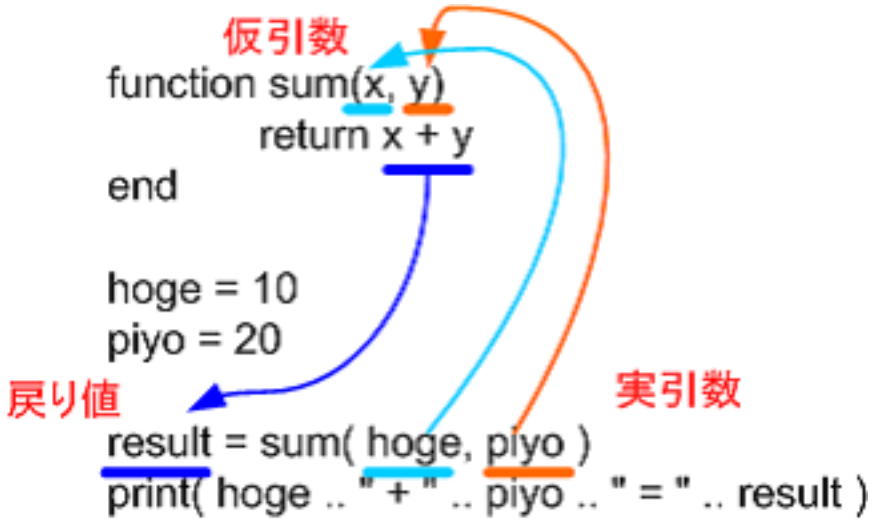

図 5.1: 関数の引数と戻り値

図 5.1 は引数の様子を示した図です．呼び出した側の引数を実引数といい，呼び出された側の引数を仮引数といいます．このとき，実引数の値が仮引数の値にコピーされます．つまり，

```
1  function func(x, y)
2          x = 10
3          y = 20
4  end
5
6  hoge = 12
7  piyo = 22
8  func( hoge, piyo )
```

```
9
10   print(hoge, piyo )
```

としても，x,y は hoge, piyo のコピーなので，中身をいくら書き換えても呼び出しもとの hoge, piyo には影響を及ぼしません.

引数に関して，もしかしたら何も受け取る必要が無い場合があるかもしれません．そういった場合は，引数の中は空にしておきます.

関数の戻り値に関しては次節で説明します.

5.3 関数の戻り値

関数には戻り値と呼ばれるものが存在しています．前節のプログラムと図 5.1 を見てください.

return は値を関数を呼び出した側に返す命令です．よって，x + y の計算結果を呼び出し元に返しています．つまり，変数 result には計算結果が代入されます.

Lua は複数の値を戻り値として返すことができます．例えば，次のようなことができます.

```
1   function func()
2           return 10, 20
3   end
4
5   hoge, piyo = func()
6
7   print(hoge, piyo)
```

実行結果

```
10 20
```

hoge には 10 が，piyo には 20 が代入されます.

では，次のようなコードを書いた場合どうなるでしょうか.

```
1   function func()
2           return 10, 20
3   end
4
5   hoge = func()
6
7   print(hoge, piyo)
```

この場合，第 2 の戻り値である 20 は使われず消滅します．

実行結果

```
10 nil
```

また，次のようなコードを書いた場合はどうでしょうか．

```
1  function func()
2          return 10
3  end
4
5  piyo = 20
6
7  hoge, piyo = func()
8
9  print(hoge, piyo)
```

第 2 の戻り値はありません．この場合，piyo には nil が代入されます．piyo は 20 ではないので注意してください．

実行結果

```
10 nil
```

5.4 何も返さない関数

戻り値の関数，つまり何も返さない関数を作ることもできます．その場合 return を書かなければ良いのです．

```
1  function func()
2          print("関数が呼び出されました")
3  end
4
5  func()
```

実行結果

```
関数が呼び出されました
```

また，明示的に戻り値が無いことを示すため，return だけを書くこともできます．

```
1   function func()
2           print("関数が呼び出されました")
3           return
4   end
5
6   func()
```

上記の二つのコードは全く同じものです.

5.5 変数に関数を代入する

Lua の変数にはどんな値でも代入することができます. たとえ関数であろうとも代入が可能です. 例えば次のようなことも可能です.

```
1   function sum(x, y)
2           return x + y
3   end
4
5   function mul(x, y)
6           return x * y
7   end
8
9   hoge = sum
10  print("hoge is " .. type(hoge) );
11  print("10 + 20 is " .. hoge(10, 20) )
12
13  hoge = mul
14  print("10 * 20 is " .. hoge(10, 20) )
```

type 関数は現在の変数の型を調べるものでしたね. hoge には sum 関数や mul 関数を代入しています. そして, hoge を使い関数にアクセスすることもできます. C 言語をご存知の方は関数ポインタのようなことができると思ってください.

実行結果

```
hoge is function
10 + 20 is 30
10 * 20 is 200
```

5.6 関数中に関数を定義する

Lua では関数内に関数を定義することも可能です.

```
1  function func(x)
2          local function get()
3                  return x
4          end
5
6          local function add(value)
7                  x = x + value
8          end
9          return get, add
10 end
11
12 firstGetValue, firstAddValue = func(10)
13 secondGetValue, secondAddValue = func(30)
14
15 print("first value : " .. firstGetValue() )
16 print("second value : " .. secondGetValue() )
17
18 firstAddValue( 15 )
19 secondAddValue( 20 )
20
21 print("first value : " .. firstGetValue() )
22 print("second value : " .. secondGetValue() )
```

func 関数は get と add 関数を戻り値として返しています．ここで local という新しいキーワードが登場しました．この local については有効範囲の章で詳しく説明します．簡単に説明しておきますと，local というキーワードを付けることでローカル変数（またはローカル関数）を作成することができます．要は，get 関数も add 関数も func 関数内からでしか呼び出せないということです.

実行結果

```
first value : 10
second value : 30
first value : 25
second value : 50
```

5.7 無名関数

名前無し関数を作成することも可能です．例えば次のようなコードを書くことができます．

```
1   function createSquare()
2           return function(x)
3                   return x*x
4           end
5   end
6
7   square = createSquare()
8   print("10 * 10 is " .. square(10, 10) )
```

createSquare 関数は戻り値に x*x を行う関数を返しています．

実行結果

```
10 * 10 is 100
```

5.8 標準ライブラリ

Lua には標準関数と呼ばれるものが存在します．これは，よく使われる処理があらかじめ関数として用意されています．例えば print 関数や io.read 関数などが標準関数にあたります．

5.8.1 基本ライブラリ

ここでは，基本ライブラリに用意されている関数の一部を紹介します．

assert 関数

定義

```
assert (v [, message])
```

assert 関数は v が偽条件ならばエラーを発生させる関数です．message にはエラーメッセージを渡します．エラーメッセージは省略可能です．

```
1   hoge = true
2   assert( hoge, "Error #1")
3   hoge = false
4   assert( hoge, "Error #2")
```

実行結果

```
lua51m.exe: test.lua:4: Error #2
stack traceback:
[C]: in function 'assert'
test.lua:4: in main chunk
[C]: ?
```

dofile 関数

定義

```
dofile (filename)
```

dofile 関数は filename で指定したファイルを実行する関数です．例えば以下のようにして使います．今回は二つの Lua ファイル，test.lua と call.lua にコードを記述しています．

```
1   -- call.lua
2   function testFunc()
3           print ("Hello")
4           return "World!"
5   end
```

```
1   -- test.lua
2   dofile("call.lua")
3
4   hoge = testFunc()
5   print( hoge )
```

実行結果

```
Hello
World!
```

dofile で call.lua を読み込む事により，test.lua 側で testFunc 関数を利用することができます．

type 関数

定義

type (v)

type 関数は，データの型を返す関数です.

```
1   hoge = nil
2   print("hoge : " .. type(hoge))
3   hoge = 10
4   print("hoge : " .. type(hoge))
5   hoge = true
6   print("hoge : " .. type(hoge))
7   hoge = "Hello"
8   print("hoge : " .. type(hoge))
```

実行結果

```
hoge : nil
hoge : number
hoge : boolean
hoge : string
```

tonumber 関数

定義

tonumber (e [, base])

tonumber 関数は引数 e を数値に変換する関数です．数値変換が可能な場合は数値を返し，そうでなければ nil を返します.

また，base を指定すると，e を base 進法の数値として変換します．省略すると，10 が自動的に指定されます.

```
1   hoge = "25"
2   print("hoge :" .. hoge .. " type :" .. type( hoge ) )
3   piyo = tonumber( hoge )
4   print("piyo :" .. piyo .. " type :" .. type(piyo) )
```

実行結果

```
hoge :25 type :string
piyo :25 type :number
```

tostring 関数

定義

tostring (e)

tostring 関数は e を文字列に変換する関数です.

```
1  hoge = 25
2  print("hoge :" .. hoge .. " type :" .. type( hoge ) )
3  piyo = tostring( hoge )
4  print("piyo :" .. piyo .. " type :" .. type(piyo) )
```

実行結果

```
hoge :25 type :number
piyo :25 type :string
```

数値から文字列に変換する際は，わざわざ tostring 関数を使用しなくても，空の文字列を追加するだけでよいです.

```
piyo = hoge .. ""
```

5.8.2 文字列操作ライブラリ

定義

書式

実行結果

第6章　C言語との連携

この章ではLuaとC言語の連携について解説していきます.

6.1　Luaスタック

LuaとC言語を連携させるためには，スタックの仕組みを理解しておかなければなりません．LuaからC言語側にデータを渡す，またはC言語側からLuaにデータを渡す場合，Luaスタックと呼ばれるものを使用します.

6.1.1　スタックとは

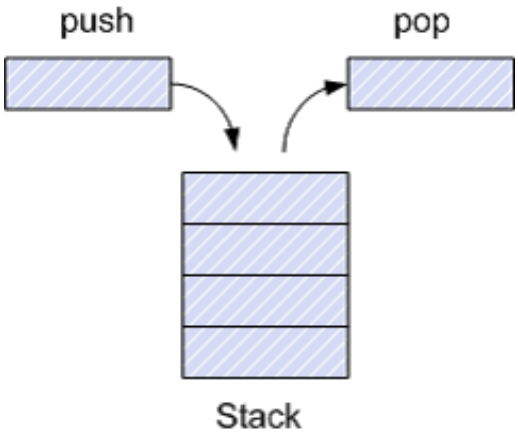

図 6.1: スタック

スタック (Stack) は図6.1で表すようなデータ構造です．データは上から順に積み上げられていきます.

スタックにデータを積み上げる動作をプッシュ(Push) するといいます．図6.2はスタックに値10をプッシュしている様子を表しています.

またスタックの上部のデータを一つ削除する動作をポップ (Pop) するといいます．図6.3はスタックから値をポップしている様子を表しています.

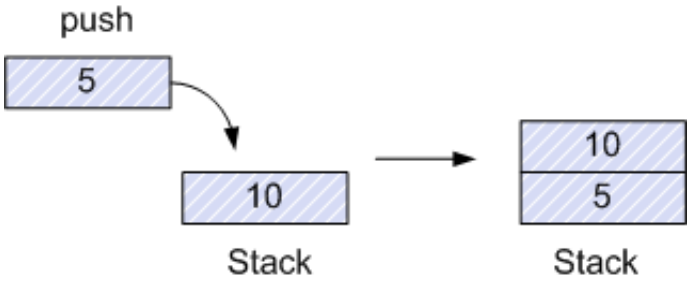

図 6.2: 値のプッシュ

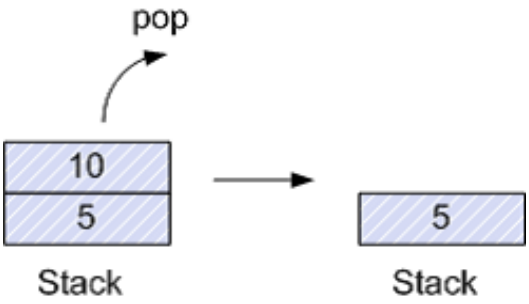

図 6.3: 値のポップ

6.1.2 Lua の初期化

次のコードは C 言語から Lua を利用するためのプログラムです．このプログラムは Lua を初期化したあとは何もしません．

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   int main (void)
8   {
9           lua_State* L = luaL_newstate();
10
11          //ここに処理を書く
12
13          lua_close(L);
14          return 0;
15  }
```

Lua を呼び出すためには 3 つのヘッダファイル及び Lua の dll が必要です．

lua_State は Lua インタプリタの状態全体を保持する不透明な構造体です．luaL_newstate() 関数を呼び出すと Lua の新しい状態を作り出し，Lua が利用可能な状態となります．Lua を利用するためには必ずこの関数を呼び出すこととなります．

lua_State 構造体である L の中には Lua インタプリタの現在の状態が保管されています．Lua を C 言語側から利用する際には必ず利用します．

また Lua を終了する場合は lua_close() 関数を呼び出してください．この関数を呼び出すことでステート内で使われていたすべての動的メモリを解放します．

6.1.3 Lua スタックへ値をプッシュする

Lua スタックへ値をプッシュする場合，lua_push***関数を呼び出します．***の部分にはデータ型が入ります．例えば boolean 型をプッシュする場合は boolean が，number 型をプッシュする場合は number が入ります．

```
lua_pushboolean(L, 1);
lua_pushnumber(L, 10.5);
lua_pushinteger(L, 3);
```

```
lua_pushstring(L, "Hello world");
lua_pushnil(L);
```

Lua 関係の関数は殆どの場合，第 1 引数に lua_State を渡す必要があります.

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
7  void dumpStack(lua_State* L)
8  {
9          int i;
10         //スタックに積まれている数を取得する
11         int stackSize = lua_gettop(L);
12         for( i = stackSize; i >= 1; i-- ) {
13                 int type = lua_type(L, i);
14                 printf("Stack[%2d-%10s] : ", i, lua_typename(L,type) );
15
16                 switch( type ) {
17                 case LUA_TNUMBER:
18                         //number 型
19                         printf("%f", lua_tonumber(L, i) );
20                         break;
21                 case LUA_TBOOLEAN:
22                         //boolean 型
23                         if( lua_toboolean(L, i) ) {
24                                 printf("true");
25                         }else{
26                                 printf("false");
27                         }
28                         break;
29                 case LUA_TSTRING:
30                         //string 型
31                         printf("%s", lua_tostring(L, i) );
32                         break;
33                 case LUA_TNIL:
34                         //nil
```

```
35                              break;
36                      default:
37                              //その他の型
38                              printf("%s", lua_typename(L, type));
39                              break;
40                      }
41                      printf("\n");
42              }
43              printf("\n");
44      }
45
46      int main (void)
47      {
48              lua_State* L = luaL_newstate();
49
50              lua_pushboolean(L, 1); //true を push
51              dumpStack(L);
52              lua_pushnumber(L, 10.5); //10.5 を push
53              dumpStack(L);
54              lua_pushinteger(L, 3); //3 を push
55              dumpStack(L);
56              lua_pushnil(L); //nil を push
57              dumpStack(L);
58              lua_pushstring(L, "Hello world"); //hello world を push
59              dumpStack(L);
60
61              lua_close(L);
62              return 0;
63      }
```

ここではLuaスタックの状態を見るためdumpStack関数を作成しました．この関数内の処理は現段階では理解しなくて結構です．

スタックの一番下は1番目となっています．値をプッシュするごとに2番目,3番目と積み重ねられていきます．dumpStack関数はスタックの位置，スタックに格納されているデータ型，及びデータの中身を表示します．ただし，データの中身を表示できるのは，number,boolean,string,nilに限ります．

実行結果

```
Stack[ 1-   boolean] : true

Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 3-    number] : 3.000000
Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 4-       nil] :
Stack[ 3-    number] : 3.000000
Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 5-    string] : Hello world
Stack[ 4-       nil] :
Stack[ 3-    number] : 3.000000
Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true
```

6.1.4 Lua スタックから値をポップする

Lua スタックから値をポップしてみましょう．値をポップするには lua_pop() 関数を利用します．

```
lua_pop(L, 1); //上から 1 つ pop する
lua_pop(L, 2); //上から 2 つ pop する
```

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
7  void dumpStack(lua_State* L)
8  {
9          (略)
```

```
10  }
11
12  int main (void)
13  {
14          lua_State* L = luaL_newstate();
15
16          lua_pushboolean(L, 1); //true を push
17          lua_pushnumber(L, 10.5); //10.5 を push
18          lua_pushinteger(L, 3); //3 を push
19          dumpStack(L);
20
21          lua_pop(L, 1); //値を 1 つ pop
22          dumpStack(L);
23
24          lua_pushnil(L); //nil を push
25          lua_pushstring(L, "Hello world"); //hello world を push
26          dumpStack(L);
27
28          lua_pop(L, 2); //値を 2 つ pop
29          dumpStack(L);
30
31          lua_close(L);
32          return 0;
33  }
```

実行結果

```
Stack[ 3-    number] : 3.000000
Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 4-    string] : Hello world
Stack[ 3-       nil] :
Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true

Stack[ 2-    number] : 10.500000
Stack[ 1-   boolean] : true
```

6.1.5 Lua スタックの操作

スタック操作系の関数を紹介します.

```
int lua_gettop(lua_State* L);
void lua_settop(lua_State* L, int index);
void lua_pushvalue(lua_State* L, int index);
void lua_remove(lua_State* L, int index);
void lua_insert(lua_State* L, int index);
void lua_replace(lua_State* L, int index);
```

lua_gettop 関数はスタックに積まれている数を取得します (図 6.4).

lua_settop 関数はスタックに積まれている値の数を変更します (図 6.5). スタックの値を削除（ポップ）するのに使われています.

index には変更後のスタックの数を指定します. ただし, マイナスの index は上から積まれたスタックの数を表しています (図 6.6).

つまり,

```
lua_settop(L, -1); //一番上のスタック
```

は一番上のスタックまで残すという意味です. つまりこの操作をしてもスタックには変化がありません. スタックを操作する時にはマイナスの index を指定する方法がよく使われます.

スタックを空にするには index に 0 を指定します.

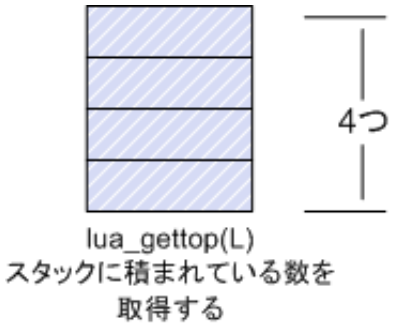

図 6.4: lua_gettop 関数

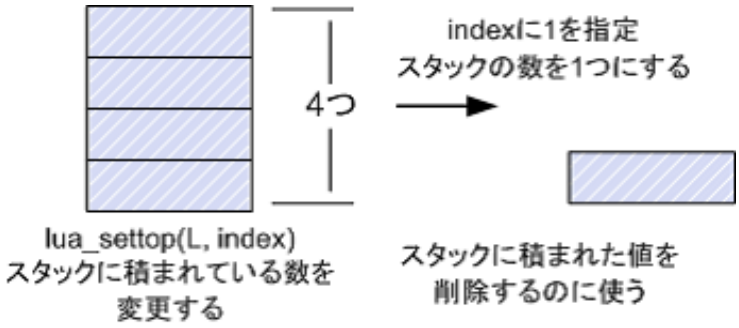

図 6.5: lua_settop 関数

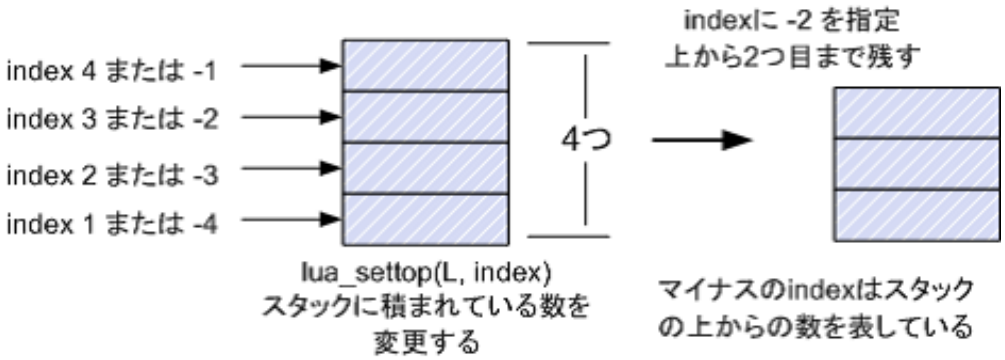

図 6.6: lua_settop 関数

```
lua_settop(L, 0); //スタックをすべて削除
```

スタックの値をポップする場合は lua_pop 関数を利用しましたが (図 6.7)，この関数は実は lua_settop 関数のマクロとなっています.

```
#define lua_pop(L, n)  lua_settop(L, -(n) - 1)
```

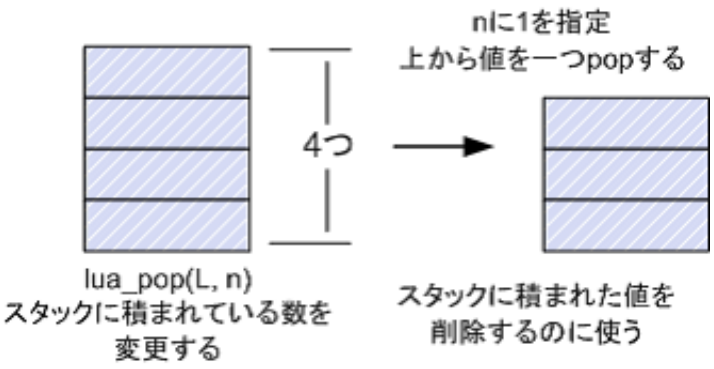

図 6.7: lua_pop 関数

lua_pushvalue 関数は指定した index の値をコピーしてスタックにプッシュします (図 6.8).

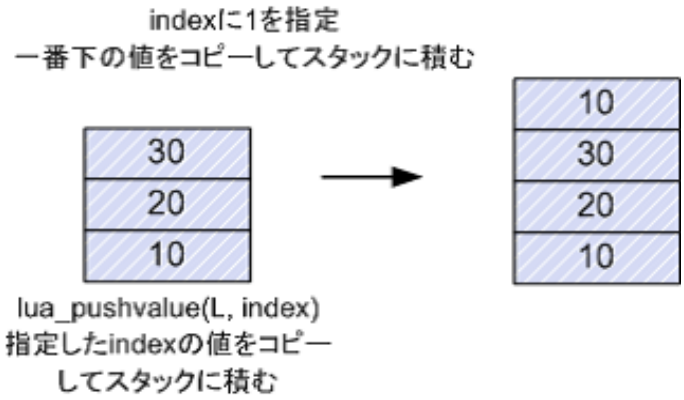

図 6.8: lua_pushvalue 関数

lua_remove 関数は指定した index の値を削除します (図 6.9)．要素を削除したら，指定した index の上に乗っていた要素をずらして隙間を埋めます.

lua_insert 関数はスタックの一番上にある要素を指定した index に挿入します (図 6.10).

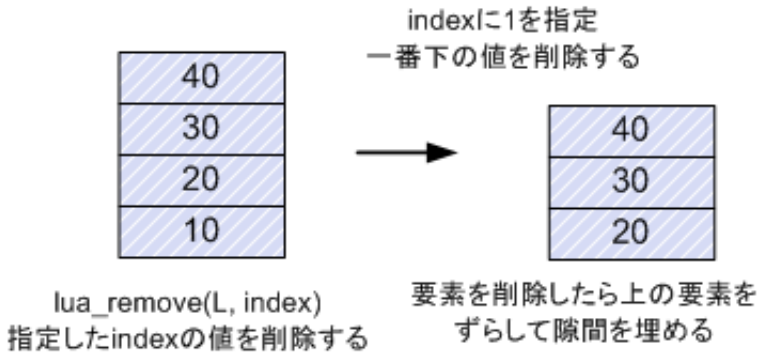

図 6.9: lua_remove 関数

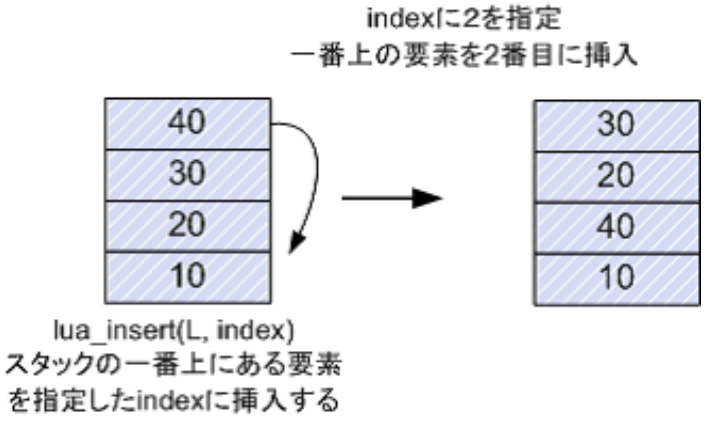

図 6.10: lua_insert 関数

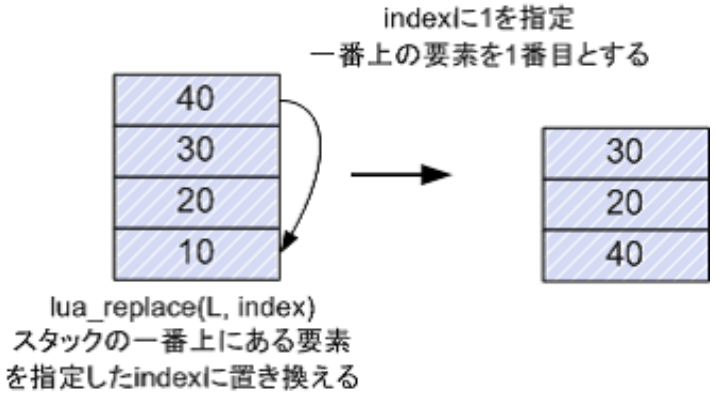

図 6.11: lua_replace 関数

lua_replace 関数はスタックの一番上の要素を指定した index に置き換えます (図 6.11).

スタック操作系の関数のサンプルコードをお見せします.

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   void dumpStack(lua_State* L)
8   {
9           (略)
10  }
11
12  int main (void)
13  {
14          lua_State* L = luaL_newstate();
15
16          lua_pushnumber(L, 10); //10 push
17          lua_pushnumber(L, 20); //20 push
18          lua_pushnumber(L, 30); //30 push
19          lua_pushnumber(L, 40); //40 push
20          dumpStack(L);
21
22          //スタックの要素数取得
23          printf("現在のスタックの数 : %d\n", lua_gettop(L));
24
25          //上から 3 つ目の要素をコピーしてスタックに積む
26          printf("pushvalue -3\n");
27          lua_pushvalue(L, -3);
28          dumpStack(L);
29
30          //上から 3 つ目の要素を削除する
31          printf("remove -3\n");
32          lua_remove(L, -3);
33          dumpStack(L);
34
35          //スタックトップの値を 2 番目に挿入する
```

```
36          printf("insert 2\n");
37          lua_insert(L, 2);
38          dumpStack(L);
39
40          //スタックトップの値を 2 番目の要素とする
41          printf("replace 2\n");
42          lua_replace(L, 2);
43          dumpStack(L);
44
45          lua_close(L);
46          return 0;
47  }
```

実行結果

```
Stack[ 4-    number] : 40.000000
Stack[ 3-    number] : 30.000000
Stack[ 2-    number] : 20.000000
Stack[ 1-    number] : 10.000000

現在のスタックの数 : 4
pushvalue -3
Stack[ 5-    number] : 20.000000
Stack[ 4-    number] : 40.000000
Stack[ 3-    number] : 30.000000
Stack[ 2-    number] : 20.000000
Stack[ 1-    number] : 10.000000

remove -3
Stack[ 4-    number] : 20.000000
Stack[ 3-    number] : 40.000000
Stack[ 2-    number] : 20.000000
Stack[ 1-    number] : 10.000000

insert 2
Stack[ 4-    number] : 40.000000
Stack[ 3-    number] : 20.000000
Stack[ 2-    number] : 20.000000
Stack[ 1-    number] : 10.000000

replace 2
Stack[ 3-    number] : 20.000000
Stack[ 2-    number] : 40.000000
Stack[ 1-    number] : 10.000000
```

6.1.6 Lua スタックのサイズ

Lua スタックに値を積むことができる数には上限があります．スタックサイズの上限を超えて値を積むとスタックオーバーフローとなってしまいます．通常は LUA_MINSTACK 個 (20 に設定されている) だけスタックに値を積むことができます．普通に使う分にはスタックの容量を気にする必要はないですが，もし大量の

値を積みたい場合は注意が必要です.

次のサンプルコードはスタックオーバーフローが発生する例です.

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   int main (void)
8   {
9           //このコードはエラーが発生する
10          int i=0;
11          lua_State* L = luaL_newstate();
12
13          for(i=0; i < 100; i++) {
14                  lua_pushnumber(L, i + 1);
15          }
16          lua_close(L);
17          return 0;
18  }
```

スタックの容量を増やすには lua_checkstack 関数を使用します.

```
int lua_checkstack (lua_State* L, int extra)
```

この関数はスタックに extra 個の値の空きがあることを保障するための関数です. つまりスタックの容量を伸ばすための関数です. extra 個だけ値を積むことができることが保障できれば true(=1) を返し，そうでなければ false(=0) を返します.

先ほどのエラーが発生したコードに lua_checkstack 関数を加えると正常に動作します.

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   void dumpStack(lua_State* L)
8   {
```

```
 9          (略)
10  }
11
12  int main (void)
13  {
14          int i=0;
15          lua_State* L = luaL_newstate();
16
17          if( lua_checkstack(L, 100) == 0 ) {
18                  printf("スタックの容量が確保できませんでした\n");
19                  return 1;
20          }
21
22          for(i=0; i < 100; i++) {
23                  lua_pushnumber(L, i + 1);
24          }
25          dumpStack(L);
26          lua_close(L);
27          return 0;
28  }
```

実行結果

```
Stack[100-    number] : 100.000000
Stack[99-    number] : 99.000000
(略)
Stack[ 3-    number] : 3.000000
Stack[ 2-    number] : 2.000000
Stack[ 1-    number] : 1.000000
```

6.1.7 Lua スタックの値の参照

スタックに積まれている値の型を見るには lua_type 関数を使用します.

```
int lua_type (lua_State* L, int index)
```

この関数は指定した index の型を返します（表 6.1）. lua_type が返す型は lua.h 内で定義されています. 有効でない index が渡された場合は LUA_TNONE を戻り値として返します.

表 6.1: lua_type 関数の戻り値

型	戻り値
nil	LUA_TNIL
number	LUA_TNUMBER
boolean	LUA_TBOOLEAN
string	LUA_TSTRING
table	LUA_TTABLE
function	LUA_TFUNCTION
userdata	LUA_TUSERDATA
thread	LUA_TTHREAD
lightuserdata	LUA_TLIGHTUSERDATA

また，指定した型かどうかを判定する lua_is***関数も存在しています．***の部分には型が入ります．

```
int lua_is*** (lua_State* L, int index)
```

例えば index で指定したスタックの要素が boolean かを判定したい場合は，

```
int lua_isboolean (lua_State* L, int index)
```

とします．返却値が true，つまり 1 であれば boolean であり，false, つまり 0 が返されれば boolean 型ではないということになります．これらの関数は実は lua_type 関数を利用したマクロとなっています．

また，指定した index の要素を指定した型で取得する関数 lua_to***関数も存在します．***の部分には型が入ります．

```
int lua_to*** (lua_State* L, int index)
```

6.1.8 Lua スタック上のテーブルの格納と参照

Lua スタック上のテーブルの値を書き換えたり参照したりすることもできます．

```
void lua_settable (lua_State *L, int index);
void lua_setfield (lua_State *L, int index, const char *k);
void lua_gettable (lua_State *L, int index);
void lua_getfield (lua_State *L, int index, const char *k);
```

スタック上に存在しているテーブルにキーと値を格納するには lua_settable 関数か lua_setfield 関数を使用します.

lua_settable 関数はスタックトップには格納する値を，スタックトップの一つ下にはキーとなる値を入れておき，これらのペアを index で指定したテーブルに格納します（図 6.12）．また，スタックに積まれていたキーと値はスタック上から削除されます.

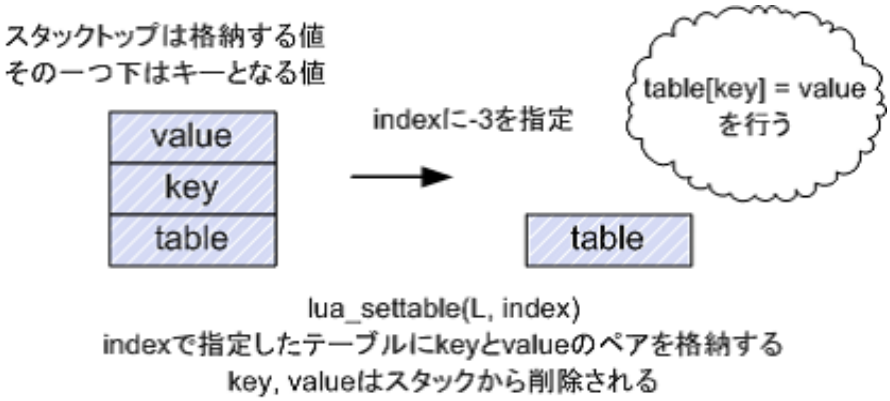

図 6.12: lua_settable 関数

lua_setfield 関数はキーを関数側から指定できます．スタックトップには格納したい値を積んでおきます．この関数は lua_settable 関数とは違い，キーには文字列しか指定できません（図 6.13）.

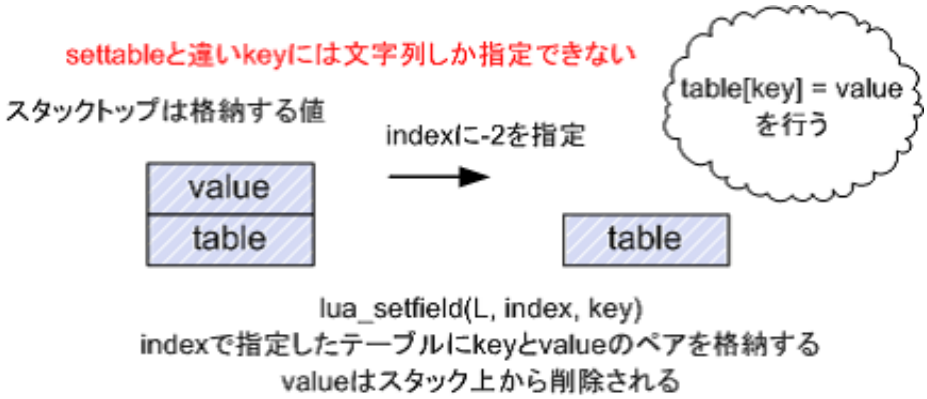

図 6.13: lua_setfield 関数

lua_gettable 関数は index で指定したテーブルからキーと対になっている値を取得しスタックに積みます．キーはスタックトップに積んでおきます（図 6.14）.

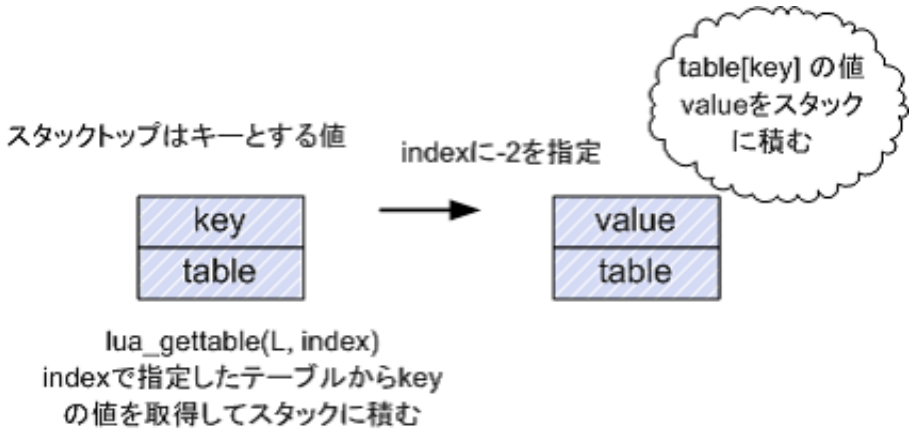

図 6.14: lua_gettable 関数

lua_getfield 関数はキーを関数側から指定できます．この関数も lua_gettable 関数とは違い，キーには文字列しか指定できません（図 6.15）．

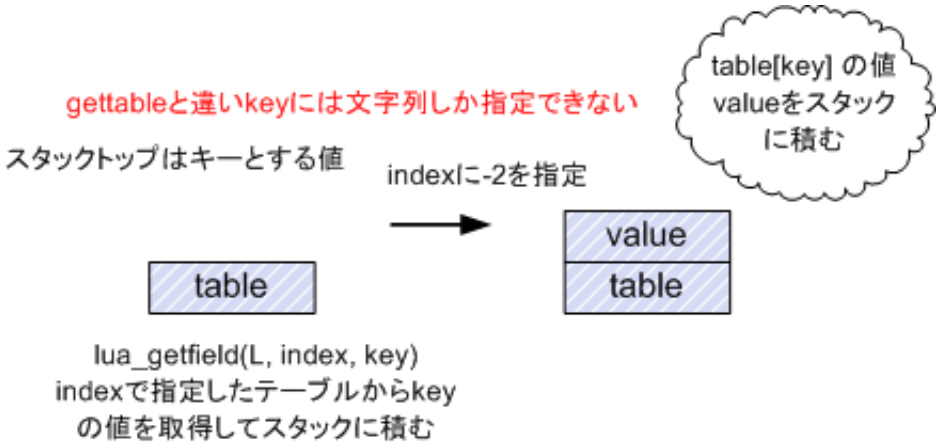

図 6.15: lua_getfield 関数

テーブルは C 言語側から作成することができます．空のテーブルを作成したい場合は lua_newtable 関数を呼び出します（図 6.16）．

```
void lua_newtable (lua_State *L);
```

この関数は実は lua_createtable 関数のマクロとなっています．

```
void lua_createtable (lua_State *L, int narr, int nrec);
```

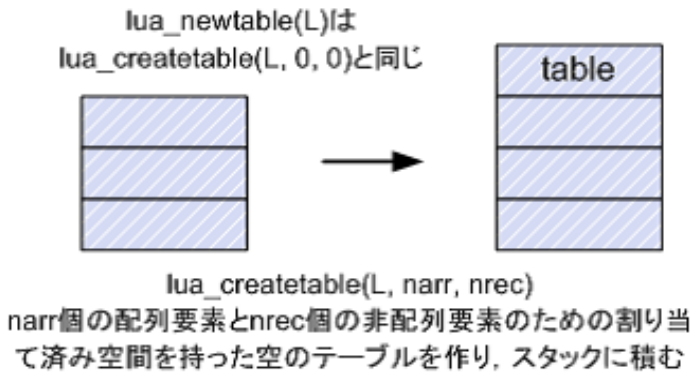

図 6.16: lua_createtable 関数

引数 narr には配列要素の個数を，nrec には非配列要素の個数を指定します．この個数を指定するとあらかじめ割り当て済みの空間が与えられます．要はテーブルがたくさん要素を持つことがあらかじめ分かっている場合に役にたつ関数です．`lua_newtable(L)` は `lua_createtable(L, 0, 0)` とマクロで定義されています．通常テーブルを作る際は lua_newtable 関数で十分です．

以上の関数を使ったサンプルコードをお見せします．

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   void dumpStack(lua_State* L)
8   {
9           (略)
10  }
11
12  int main (void)
13  {
14          int i=0;
15          lua_State* L = luaL_newstate();
16
17          lua_newtable(L); //新しい table の作成
18          lua_pushnumber(L, 10); //key
19          lua_pushstring(L, "hello"); //value
```

```
20          dumpStack(L);
21
22          // table[10] = "hello" を実行
23          lua_settable(L, -3);
24          dumpStack(L);
25
26          //value
27          lua_pushstring(L, "world");
28          //table["key"] = "world" を実行
29          lua_setfield(L, -2, "key");
30          dumpStack(L);
31
32          lua_pushnumber(L, 10);
33          //table[10] の値を取り出しスタックに積む
34          lua_gettable(L, -2);
35          dumpStack(L);
36
37          //table["key"] の値を取り出しスタックに積む
38          lua_getfield(L, -2, "key");
39          dumpStack(L);
40
41          lua_close(L);
42          return 0;
43  }
```

実行結果

```
Stack[ 3-    string] : hello
Stack[ 2-    number] : 10.000000
Stack[ 1-     table] : table

Stack[ 1-     table] : table

Stack[ 1-     table] : table

Stack[ 2-    string] : hello
Stack[ 1-     table] : table

Stack[ 3-    string] : world
Stack[ 2-    string] : hello
Stack[ 1-     table] : table
```

6.2 C から Lua を呼び出す

この節では C 言語側から Lua の変数や関数を呼び出す方法を説明していきます.

6.2.1 Lua ファイルを読み込む

まずは C 言語側から Lua で書かれたコードを読み込んでみましょう. 次のような Lua ソースコードを作成したとします. このコードのファイル名は sample.lua です.

```
1   --sample.lua
2
3   --グローバル変数
4
5   NAME = "HELLOWORLD"
6   SIZE = 640
```

この Lua ファイルを読み込むための C 言語コードは以下のようになります.

```
1   #include <stdio.h>
2
3   #include "lua.h"
```

```
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   int main (void)
8   {
9           //Lua を開く
10          lua_State* L = luaL_newstate();
11          //Lua の標準関数を使用できる状態にする
12          luaL_openlibs(L);
13          //Lua ファイル sample.lua を読み込む
14          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
15                  printf("sample.lua を開けませんでした\n");
16                  printf("error : %s\n", lua_tostring(L, -1) );
17                  return 1;
18          }
19          //ここに何か処理を書く
20
21          lua_close(L);
22          return 0;
23  }
```

このプログラムは sample.lua を読み込むだけで何も処理は行いません．

ここで新たに 3 つの新しい関数が登場しました．まず，luaL_openlibs 関数ですが，この関数は Lua 内で標準関数が利用できるようにするものです．この関数を呼び出さないと Lua 内で print 関数などが使用できません．

luaL_loadfile 関数ですが，次のような定義となっています．

```
int luaL_loadfile (lua_State *L, const char *filename);
```

filename には開きたい Lua ファイルを指定します．この関数はファイルが読み込めなかった場合（ファイルが存在しない，文法が間違っていた等），LUA_ERRFILE を返します．

lua_pcall 関数ですが，この関数は本来 Lua の関数を呼ぶためのものです．ここでは使い方を詳しく説明しませんが，loadfile を行った後ファイルを実行する役目を果たしています．

luaL_openlibs 関数及び lua_pcall 関数は成功すると 0 を返します．この戻り値を使ってファイルが正しく読み込めたかどうかを判定できます．また，エラーが発生した場合，エラーの詳細をスタックに積んでくれます．よってスタックトップの値を参照すればエラーの原因を特定できます．

では，試しに luaL_loadfile 関数の引数に sample.lua ではなく存在しないファイル test.lua を指定してみましょう．結果は以下のようになります．

実行結果

```
sample.lua を開けませんでした
error : cannot open test.lua: No such file or directory
```

また Lua ファイルの 2 行目に適当な文法の文字列を挿入してみます．挿入した文字列は `if a-b=c then end` です．

実行結果

```
sample.lua を開けませんでした
error : sample.lua:2: 'then' expected near '='
```

6.2.2 Lua のグローバル変数を取得する

前節で登場した sample.lua から変数 NAME と SIZE の値を取得してみましょう．NAME は文字列で取得し，SIZE は NUMBER で取得します．

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   int main (void)
8   {
9           //Lua を開く
10          lua_State* L = luaL_newstate();
11          //Lua の標準関数を使用できる状態にする
12          luaL_openlibs(L);
13          //Lua ファイル sample.lua を読み込む
14          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
15                  printf("sample.lua を開けませんでした\n");
16                  printf("error : %s\n", lua_tostring(L, -1) );
17                  return 1;
18          }
19
20          //PATH を取得しスタックに積む
```

```
21          lua_getglobal(L, "NAME");
22          //SIZE を取得しスタックに積む
23          lua_getglobal(L, "SIZE");
24
25          if( !lua_isstring(L, -2) || !lua_isnumber(L, -1) ) {
26                  printf("正しく値が取得できませんでした\n");
27                  return 1;
28          }
29          printf("NAME : %s\n", lua_tostring(L, -2));
30          printf("SIZE : %d\n", lua_tointeger(L, -1));
31
32          lua_close(L);
33          return 0;
34  }
```

実行結果

```
NAME : HELLOWORLD
SIZE : 640
```

lua_getglobal 関数はグローバル変数を取得する変数で以下のように定義されています.

```
void lua_getglobal (lua_State *L, const char *name);
```

name には取得したいグローバル変数の変数名を指定します．実はこの関数は lua_gettable 関数のマクロとなっています.

```
#define lua_getglobal(L,s)  lua_getfield(L, LUA_GLOBALSINDEX, s)
```

LUA_GLOBALSINDEX は Lua のグローバル変数がある領域テーブルを指しています.

さて，上記のサンプルですがスタックには NAME と SIZE が残ったままになっています．実際のプログラムでは NAME,SIZE を使用したあとはきちんとポップしておきましょう.

6.2.3 Lua 関数を呼び出す

C 言語側から Lua の関数を呼び出してみましょう．次のような add 関数を Lua で作成したとします．ファイル名は sample.lua です.

```
1   --sample.lua
2
3   function add(x, y)
4           print("x : ".. x .. " y : " .. y .. "を受け取りました")
5           return x + y
6   end
```

add 関数は二つの引数 x,y を受け取り，その合計値を返却します．では C 言語側のサンプルコードをお見せします．

```
1   #include <stdio.h>
2
3   #include "lua.h"
4   #include "lualib.h"
5   #include "lauxlib.h"
6
7   int main (void)
8   {
9           int x = 10, y = 5;
10
11          //Lua を開く
12          lua_State* L = luaL_newstate();
13          //Lua の標準関数を使用できる状態にする
14          luaL_openlibs(L);
15          //Lua ファイル sample.lua を読み込む
16          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
17                  printf("sample.lua を開けませんでした\n");
18                  printf("error : %s\n", lua_tostring(L, -1) );
19                  return 1;
20          }
21          //add 関数をスタックに積む
22          lua_getglobal(L, "add");
23          //第 1 引数 x
24          lua_pushnumber(L, x);
25          //第 2 引数 y
26          lua_pushnumber(L, y);
27
28          //add(x, y) を呼び出す 引数 2 個，戻り値 1 個
29          if(lua_pcall(L, 2, 1, 0) != 0) {
```

```
30                      printf("関数呼び出し失敗\n");
31                      printf("error : %s\n", lua_tostring(L, -1) );
32                      return 1;
33              }
34              if( lua_isnumber(L, -1) ) {
35                      printf("結果 : %d\n", lua_tointeger(L, -1) );
36                      lua_pop(L,1); //戻り値をポップ
37              }
38
39              lua_close(L);
40              return 0;
41      }
```

実行結果

```
x : 10 y : 5 を受け取りました
結果 : 15
```

ここで登場した関数が lua_pcall 関数です（図 6.17）.

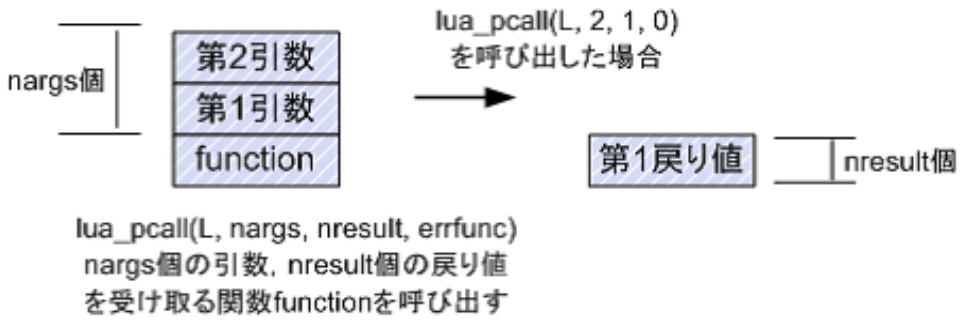

図 6.17: lua_pcall 関数

この関数は次のように定義されています.

```
lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

nargs には関数に渡す引数の数を，nresults は関数から受け取る戻り値の数を指定します．errfunc は通常は 0 のままでかまいません.

さて，関数を呼ぶには次のような手順が必要です．まず，始めに呼び出したい関数を取得しスタックに積みます．次にその関数に渡す引数を順番どおりに積んでいきます．つまり第 1 引数は一番初めにプッシュしなければなりません．そして lua_pcall を呼び出すとその関数を実行します．関数を呼び出した後は，スタッ

クに積まれていた関数，及び引数はスタック上から削除されます．関数からの戻り値がある場合は戻り値の順番どおり，つまり最初の戻り値は最初にスタックに積まれていきます．

また，lua_pcall 関数は関数の呼び出しに失敗するとエラーコードをスタックに積みます．

6.2.4 Lua にグローバル変数を登録する

C 言語側から Lua のグローバル変数を登録してみましょう．ここでは TEST という変数（値は 10 とする）を登録してみます．

まずは次のようなコード，sample.lua を作ります．

```
1  --sample.lua
2
3  function show()
4          if TEST == nil then
5                  print("TEST は登録されていません")
6          else
7                  print("TEST の値は" .. TEST .. "です")
8          end
9  end
```

そして C 言語側のコードは次のようになります．

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
7  void showTest(lua_State* L)
8  {
9          //show 関数をスタックに積む
10         lua_getglobal(L, "show");
11         //show 関数を呼び出す 引数 0 個，戻り値 0 個
12         if(lua_pcall(L, 0, 0, 0) != 0) {
13                 printf("関数呼び出し失敗\n");
14                 printf("error : %s\n", lua_tostring(L, -1) );
15                 lua_pop(L, 1);
```

```
16          }
17  }
18
19  int main (void)
20  {
21          int x = 10, y = 5;
22
23          //Lua を開く
24          lua_State* L = luaL_newstate();
25          //Lua の標準関数を使用できる状態にする
26          luaL_openlibs(L);
27          //Lua ファイル sample.lua を読み込む
28          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
29                  printf("sample.lua を開けませんでした\n");
30                  printf("error : %s\n", lua_tostring(L, -1) );
31                  return 1;
32          }
33
34          //Lua の show 関数を呼び出す
35          showTest(L);
36          //変数 TEST(値は 10) を Lua に登録
37          lua_pushnumber(L, 10);
38          lua_setglobal(L, "TEST");
39          //TEST 表示
40          showTest(L);
41
42          lua_close(L);
43          return 0;
44  }
```

実行結果

```
TEST は登録されていません
TEST の値は 10 です
```

Lua の show 関数を呼び出すための C 言語関数 showTest を作成しました．1 回目に show 関数が呼び出された時は，変数 TEST はまだ存在しないので nil となっています．2 回目に呼び出された時は，C 言語側から TEST=10 として登録されているので，値が表示されます．

さて，グローバル変数を登録するには lua_setglobal 関数を使用します．

```
void lua_setglobal (lua_State *L, const char *name);
```

この関数はスタックトップの値を name として Lua に登録します. 実は lua_setglobal 関数は lua_setfield 関数のマクロとして定義されています.

```
#define lua_setglobal(L,s)   lua_setfield(L, LUA_GLOBALSINDEX, s)
```

6.3 Lua から C を呼び出す

この節では Lua 側から C の関数を呼び出す方法について説明していきます.

6.3.1 C 関数を呼び出す

Lua 側から C 言語の関数 add を呼び出してみます.

Lua 側では次のようなソースを書いたとします. Lua ファイル名は sample.lua です.

```
1  --sample.lua
2
3  x,y = 5, 10
4  result = add(x, y)
5  print("x + y は " .. result .. " です")
```

では C 言語側の関数をお見せします.

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
7  int l_add(lua_State* L)
8  {
9          //第 1 引数 int として取得
10         int x = luaL_checkint(L, -2);
11         //第 2 引数
12         int y = luaL_checkint(L, -1);
13         int result = x + y;
14
```

```
15          printf("%d + %d を計算します\n", x, y);
16
17          //戻り値をスタックに積む
18          lua_pushnumber(L, result);
19          return 1; //戻り値の数を指定
20  }
21
22  int main (void)
23  {
24          int x = 10, y = 5;
25
26          //Lua を開く
27          lua_State* L = luaL_newstate();
28          //Lua の標準関数を使用できる状態にする
29          luaL_openlibs(L);
30
31          //l_add 関数を add 関数として Lua に登録
32          lua_register(L, "add", l_add);
33
34          //Lua ファイル sample.lua を読み込む
35          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
36                  printf("sample.lua を開けませんでした\n");
37                  printf("error : %s\n", lua_tostring(L, -1) );
38                  return 1;
39          }
40
41          lua_close(L);
42          return 0;
43  }
```

実行結果

```
5 + 10 を計算します
x + y は 15 です
```

Lua 側で C 言語関数を呼び出すと，l_add 関数の Lua スタックに引数が積まれます．C 言語側の引数に直接値が渡されるわけではないので注意してください．第 1 引数は始めにプッシュされ，第 2 引数はその次にプッシュされます．よって C 言語側からこのスタックの値を取得すればよいわけです．

スタックの値を取得する時に luaL_check***関数を使用しました．***には Lua 変数のデータ型が入ります．この関数は指定したスタックの値を***型であるかを判定してから取得します．もし***型で無かった場合はエラーが発生します．

試しに add 関数の第 1 引数に文字列を渡してみました．結果は以下のようになりました．

実行結果

```
sample.lua を開けませんでした
error : sample.lua:4: bad argument #-2 to
'add' (number expected, got string)
```

l_add 関数から戻り値を返す場合も Lua スタックに値を積みます．l_add 関数の戻り値には戻り値の数を指定します．

さて，Lua 側から C 関数を呼び出すためには，Lua に関数を登録しなければなりません．その際使用するのが lua_register 関数です．

```
void lua_register (lua_State *L, const char *name, lua_CFunction f);
```

name には Lua から呼び出す関数名を登録します．f には C 言語関数を渡します．
実はこの関数は以下のようにマクロとなっています．

```
#define lua_register(L,n,f) (lua_pushcfunction(L, f), lua_setglobal(L, n))
```

lua_pushcfunction 関数は C 言語関数 f をスタックにプッシュするものです．

6.3.2 関数毎の Lua スタック

前節の l_add 関数のスタック上の値について考えて見ましょう．l_add 関数を呼び出した後の Lua スタックには引数 x,y や戻り値 result が残ったままになっているでしょうか．もし残ったままならポップしなければいけないのかという疑問が残っています．

ここで次のようなコードを作ってみます．前節でも登場した dumpStack 関数を使用しています．

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
```

```
 7  void dumpStack(lua_State* L)
 8  {
 9          int i;
10          //スタックに積まれている数を取得する
11          int stackSize = lua_gettop(L);
12          for( i = stackSize; i >= 1; i-- ) {
13                  int type = lua_type(L, i);
14                  printf("Stack[%2d-%10s] : ", i, lua_typename(L,type) );
15
16                  switch( type ) {
17                  case LUA_TNUMBER:
18                          //number型
19                          printf("%f", lua_tonumber(L, i) );
20                          break;
21                  case LUA_TBOOLEAN:
22                          //boolean型
23                          if( lua_toboolean(L, i) ) {
24                                  printf("true");
25                          }else{
26                                  printf("false");
27                          }
28                          break;
29                  case LUA_TSTRING:
30                          //string型
31                          printf("%s", lua_tostring(L, i) );
32                          break;
33                  case LUA_TNIL:
34                          //nil
35                          break;
36                  default:
37                          //その他の型
38                          printf("%s", lua_typename(L, type));
39                          break;
40                  }
41                  printf("\n");
42          }
43          printf("\n");
44  }
```

```
45
46  int l_add(lua_State* L)
47  {
48          //第1引数 int として取得
49          int x = luaL_checkint(L, -2);
50          //第2引数
51          int y = luaL_checkint(L, -1);
52          int result = x + y;
53
54          printf("%d + %d を計算します\n", x, y);
55
56          //戻り値をスタックに積む
57          lua_pushnumber(L, result);
58
59          //ダンプ2箇所目
60          printf("2番目\n");
61          dumpStack(L);
62
63          return 1; //戻り値の数を指定
64  }
65
66  int main (void)
67  {
68          int x = 10, y = 5;
69
70          //Lua を開く
71          lua_State* L = luaL_newstate();
72          //Lua の標準関数を使用できる状態にする
73          luaL_openlibs(L);
74
75          //l_add 関数を add 関数として Lua に登録
76          lua_register(L, "add", l_add);
77
78          //適当な値をプッシュ
79          lua_pushnumber(L, 444);
80          lua_pushnumber(L, 555);
81
82          //ダンプ1箇所目
```

```
83          printf("1 番目\n");
84          dumpStack(L);
85
86          //Lua ファイル sample.lua を読み込む
87          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
88                  printf("sample.lua を開けませんでした\n");
89                  printf("error : %s\n", lua_tostring(L, -1) );
90                  return 1;
91          }
92
93          //ダンプ 1 箇所目
94          printf("3 番目\n");
95          dumpStack(L);
96
97          lua_close(L);
98          return 0;
99  }
```

3 箇所から dumpStack 関数を呼び出しています．さて，実行結果は以下の通りとなりました．

実行結果

```
1 番目
Stack[ 2-    number] : 555.000000
Stack[ 1-    number] : 444.000000

5 + 10 を計算します
2 番目
Stack[ 3-    number] : 15.000000
Stack[ 2-    number] : 10.000000
Stack[ 1-    number] : 5.000000

x + y は 15 です
3 番目
Stack[ 2-    number] : 555.000000
Stack[ 1-    number] : 444.000000
```

実は Lua スタックは関数ごとに用意されています．つまり，l_add 関数からは main 関数側のスタックは見えないわけです．また，l_add 関数のスタックは Lua が

勝手に削除してくれます．よって，l_add 関数上のスタックはわざわざポップする必要はありません．

6.3.3 テーブルに C 言語関数を登録する

前節では，関数をグローバル関数として定義しました．しかし，関数が増えてくると名前の衝突などが起こってしまう場合があります．そういった事態を避けるため，テーブル内に関数を定義するといった方法が取られます．

次のようなサンプルコードを書いてみます．まずは Lua 側のコードです．

```
1  --sample.lua
2
3  x, y = 5, 10
4  print( x .. " + " .. y .. " = " .. myMath.add(x, y) )
5  print( x .. " * " .. y .. " = " .. myMath.mul(x, y) )
```

myMath テーブルにある add 関数と mul 関数を呼び出しています．

では，テーブルに関数を登録する C のサンプルコードをお見せします．

```
1  #include <stdio.h>
2
3  #include "lua.h"
4  #include "lualib.h"
5  #include "lauxlib.h"
6
7  int l_add(lua_State* L)
8  {
9          //第 1 引数 int として取得
10         int x = luaL_checkint(L, -2);
11         //第 2 引数
12         int y = luaL_checkint(L, -1);
13         int result = x + y;
14
15         printf("%d + %d を計算します\n", x, y);
16
17         //戻り値をスタックに積む
18         lua_pushnumber(L, result);
19
20         return 1; //戻り値の数を指定
```

```
21  }
22
23  int l_mul(lua_State* L)
24  {
25          //第1引数 int として取得
26          int x = luaL_checkint(L, -2);
27          //第2引数
28          int y = luaL_checkint(L, -1);
29          int result = x * y;
30
31          printf("%d * %d を計算します\n", x, y);
32
33          //戻り値をスタックに積む
34          lua_pushnumber(L, result);
35
36          return 1; //戻り値の数を指定
37  }
38
39  //登録する関数
40  static const struct luaL_Reg myMathLib [] = {
41          {"add", l_add},
42          {"mul", l_mul},
43          {NULL, NULL} //最後は必ず NULL のペア
44  };
45
46  int main (void)
47  {
48          int x = 10, y = 5;
49
50          //Lua を開く
51          lua_State* L = luaL_newstate();
52          //Lua の標準関数を使用できる状態にする
53          luaL_openlibs(L);
54
55          //add, mul を myMath テーブルに登録
56          luaL_register(L, "myMath", myMathLib);
57
58          //Lua ファイル sample.lua を読み込む
```

```
59          if( luaL_loadfile(L, "sample.lua") || lua_pcall(L, 0, 0, 0) ) {
60                  printf("sample.lua を開けませんでした\n");
61                  printf("error : %s\n", lua_tostring(L, -1) );
62                  return 1;
63          }
64
65          lua_close(L);
66          return 0;
67  }
```

実行結果

```
5 + 10 を計算します
5 + 10 = 15
5 * 10 を計算します
5 * 10 = 50
```

luaL_reg 構造体は以下のように定義されています.

```
typedef struct luaL_Reg {
        const char *name;
        lua_CFunction func;
} luaL_Reg;
```

name 側には Lua 側に登録したい関数名を，func には C 言語関数を渡します.

myMath テーブルを新たに作成し，luaL_reg 構造体で定義した関数を登録するためには luaL_register 関数を使用します.