Lua 5.1 Használati útmutató

készítette Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes

1 - Bevezetés

A Lua egy eljárásokon alapuló, kiegészítő programnyelv, adat leírási lehetőségekkel kibővítve. Magasfokú támogatást nyújt az objektum-orientált, funkcionális vagy adat-vezérelt programozáshoz is. A Lua nyelvet egy erős, 'nehézsúlyú' scriptnyelvnek szánták, amelyet bármilyen program használhat. A Lua függvénykönyvtárként lett létrehozva, és *tiszta* C nyelven lett írva (azaz az ANSI C és C++ nyelvek általános részhalmazaként).

Mivel ez egy kiegészítő nyelv, így nem létezik "fő" program sem: csak a fő kliensbe beágyazva működik, amelyet *beágyazó programnak* vagy egyszerűen hostnak neveznek. Ez a host program megadhatja egyes függvényeknek, hogy hajtsák végre a megadott Lua kódot, Lua változókat írhat és olvashat, valamint C funkciókat regisztrálhat, amelyeket később a Lua kód meghívhat. A C függvények használatával a Lua kiterjeszthető, és így különböző problémák széles tartományával megbirkózhat, így olyan személyre szabott programozási nyelvek készíthetőek, amelyek ugyanazt a szintaktikai keretrendszert használják. A Lua csomag tartalmaz egy host programot, amelynek `lua` a neve. Ez a Lua eljáráskönyvtárát használva biztosítja a teljes, egyedülálló Lua értelmezőt.

A Lua egy ingyenes program, amely nem vállal garanciát semmilyen esetre, ahogy az a licencében is szerepel. Ez a leírás a Lua hivatalos weblapján is megtekinthető: `www.lua.org`.

Mint megannyi másik leírás, ez a dokumentum is hiányos néhány helyen. A Lua kialakításáról szóló fórum szintén megtalálható a Lua weboldalán. Részletesebb programozási ismeretek elsajátításához nyújt segítséget Robert könyve, a *Programming in Lua*.

2 - A nyelv

Ez a rész a szintaktikai és szemantikai részét írja le a Lua nyelvnek. Más szóval, ez a rész határozza meg, mely szimbólumok elfogadottak és érvényesek, ezek hogyan kombinálhatóak, és az egyes kombinációk mit jelentenek.

A nyelv konstruktorai ('létrehozói') a kiterjesztett BNF jelrendszer segítéségével lesznek feltüntetve, ahol az {a} kifejezés értéke 0 vagy több a, és az [a] kifejezés jelentése, hogy a opcionális, elhagyható. A nem-kulcsszavak nem-kulcsszó formában, a kulcsszavak **kulcsszó** formában, a többi kifejezés pedig `**=**´ formában szerepel. A Lua teljes szintaktikája a leírás végén található.

2.1 - Lexikális szabályok

A Lua nyelvben a *nevek* (más néven *azonosítók*) bármilyen betűkből, számokból és aláhúzás karakterből álló karakterlánc lehetnek, azonban nem kezdődhetnek számjeggyel. Ez megegyezik a legtöbb nyelv azonosítóinak szabályaival. (A betű meghatározása a helyi beállításokon múlik: bármely karakter, amely a beállítások szerint alfabetikusnak minősül, használható azonosítóként.) Az azonosítók a változók megnevezésére és a tömb mezők azonosítására használhatóak.

A következő *kulcsszavak* foglaltak, és nem használhatóak azonosítókként:

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

A Lua nyelv nem tesz különbséget kis- és nagybetűk között, így amíg az `and` foglalt szó, az And és az AND két különböző, érvényes név. Megállapodás, hogy azok a nevek, amelyek aláhúzással kezdődnek, és nagybetűkkel folytatódnak (mint pl. a `_VERSION`) a Lua számára fenntartott belső globális változók.

A következő karakterláncok más vezérjelet fejeznek ki:

```
+ - * / % ^ #
== ~= <= >= < > =
( ) { } [ ]
; : , . .. ...
```

A *literális (szó szerinti) karakterláncok* a sima és dupla idézőjelekkel vannak határolva, és a következő C-alapú vezérlősorozatokat tartalmazhatják: '`\a`' (csengő), '`\b`' (törlés), '`\f`' (lapemelés), '`\r`' (kocsivissza), '`\t`' (vízszintes tabulátor), '`\v`' (függőleges tabulátor), '`\\`' (backslash), '`\"`' (idézőjel [dupla idézőjel]), valamint '`\'`' (aposztróf [szimpla idézőjel]). (Ezeken felül, egy backslash után elhelyezett újsor egy újsort eredményez a karakterláncban.) Egy karakter megadható a numerikus értékével, \ddd formában is, ahol ddd maximum három számjegyet jelenthet (ha a numerikus blokkot egy szám követi, akkor pontosan három számjegyet kell használni). A karakterláncok a Lua nyelvben tartalmazhatnak bármilyen 8-bites értéket, beleértve a beágyazott nullát is, amely '`\0`' formában adható meg.

A literális karakterláncokban, melyeket dupla (v. szimpla) idézőjelek zárnak le, a dupla (szimpla) idézőjeleket, az újsor karaktert, a backslash-t, vagy a beágyazott

nullát vezérlőkarakterrel kell ellátni. A többi karakter közvetlenül beilleszthető a karakterláncba. (Néhány vezérlőkarakter problémát okozhat a fájlrendszer számára, de a Lua-nak nincs problémája ezekkel).

A literális karakterláncok hosszú formában is meghatározhatóak a *hosszú zárójelek* segítségével. Az *n szintű nyitó hosszú zárójel* megadásakor a nyitó szögletes zárójel után *n* egyforma jel következik, melyet még egy nyitó szögletes zárójel követ. Tehát a 0. szintű nyitó hosszú zárójel kezdő formulája `[[`, az első szintűé `[=[`, és így tovább. A *záró hosszú zárójel* megadása hasonló módon történik, például, a negyedik szintű záró hosszú zárójel formulája `]====]`. A hosszú karakterlánc bármely szintű nyitó hosszú zárójellel kezdődik, és az ugyanolyan szintű záró hosszú zárójelig tart. A literális karakterláncok szögletes zárójelezett formája több sorban is szerepelhet, ebben az esetben nem dolgozza fel az esetleges vezérlőkaraktereket, és figyelmen kívül hagyja a köztes, bármilyen szintű hosszú zárójeleket. Bármit tartalmazhatnak, kivéve a megfelelő szintű záró hosszú zárójelet.

Kényelmi okokból, abban az esetben, ha a nyitó hosszú zárójelet közvetlenül egy újsor követ, az újsor karakter nem kerül bele a karakterláncba. Például, egy ASCII-t használó rendszer (ahol '`a`' kódja 97, az újsor kódja 10, és '`1`' kódja 49), a következő öt literális karakterlánc ugyanazt határozza meg:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

A *numerikus konstansok* leírhatóak opcionális decimális résszel és egy opcionális decimális hatványkitevővel. A Lua szintén elfogadja a hexadecimális konstansokat, ez esetben ezek `0x` előtaggal kezdődnek. Érvényes numerikus konstansoknak tekinthetőek a következő kifejezések:

```
3 3.0 3.1416 314.16e-2 0.31416E1 0xff 0x56
```

A *megjegyzés* (komment) dupla gondolatjellel (--) kezdődik, bárhol a karakterláncon kívül. Ha a -- jelek után következő szöveg nem nyitó hosszú zárójellel kezdődik, akkor rövid megjegyzésről beszélünk, ellenkező esetben ez egy hosszú megjegyzés, amely az azonos szintű záró hosszú zárójelig tart. A hosszú megjegyzések legtöbbször egy kódrészlet ideiglenes kiiktatására szolgálnak.

2.2 - Értékek és típusok

A Lua egy *dinamikus típusú nyelv*. Ez azt jelenti, hogy a változóknak nincsenek típusaik, csak az értékeknek. A nyelvben nincsenek típus definíciók sem, minden érték magának alakítja ki a típusát.

A nyelvben minden érték *első osztályú érték*. Ez azt jelenti, hogy az értékek változókban tárolhatóak, függvények paramétereiként használhatóak és visszatérési értékek is lehetnek.

Nyolc típus létezik a Lua nyelvben: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, és *table*. *Nil* a típusa a **nil** értéknek, amelynek fő tulajdonsága, hogy különbözik az összes többi értéktől; általában a használható érték hiányát jelenti. A *boolean* típusnak két értéke lehet: **false** (hamis) és **true** (igaz). Mind a **nil**, mind a **false** hamissá tesz egy feltételt; minden egyéb érték igazzá. A *number* típus valódi (dupla-pontosságú lebegőpontos) számokat jelöl. (Nem nehéz olyan Lua értelmezőt készíteni, amely más típusú számokat használ, például szimpla pontosságú lebegőpontost vagy hosszú egész számost; lásd a `luaconf.h` fájlt.) A *string* típus egy karaktertömböt hoz létre. A Lua 8-bit alapú: a karakterláncok bármilyen 8-bites karaktert tartalmazhatnak, beleértve a beágyazott nullát ('`\0`') is (lásd §2.1).

A Lua meg tud hívni (és kezelni is tud) Lua és C nyelven írt függvényeket is. (lásd §2.5.8).

Az *userdata* típus végtelen számú C adat tárolását biztosítja LUA változókban. Ez a típus egy nyílt memóriaterület címére hivatkozik, és nincs előre meghatározott művelete a Lua számára, kivéve a hozzárendelést és az azonosítási tesztet. Azonban a *metatömbök* használatával a programozó megadhat műveleteket az userdata értékek számára (lásd §2.8). Ezek az értékek Lua-ból nem módosíthatóak és nem is hozhatóak létre, csak a C API-n keresztül. Ez garantálja a host program adatainak sértetlenségét.

A *thread* típus segítségével független futási szálak hozhatóak létre, és korutinok valósíthatóak meg (lásd §2.11). A Lua szálak nem összekeverendőek az operációs-rendszerbeli szálakkal. A nyelv minden rendszeren támogatja a korutinok használatát, még azokon is, amelyek nem támogatják a szálakat.

A *table* típus asszociatív tömböt jelent, így tehát a tömbök indexei (azonosítói) nem csak számok, hanem bármilyen más értékek is lehetnek (kivéve a **nil**-t). A táblák heterogének is lehetnek, így az értékek is bármilyen típusúak lehetnek (kivéve a **nil**-t) A tábla az egyetlen adatstruktúra mechanizmus a nyelvben; képviselhetnek rendezett tömböket, szimbólumtárakat, halmazokat, bejegyzéseket, gráfokat, fákat, stb. Bejegyzések készítésekor a Lua a mező nevét használja indexként. A nyelv támogatja az elérést `a.name` és `a["name"]` formában is. A tömbök létrehozásának több módja is van a nyelvben (lásd §2.5.7).

Ahogy az indexek, úgy az értékek is bármilyen típusúak lehetnek (kivéve a **nil**-t). Bizonyos esetekben, mivel a függvények első-osztályú értékek, a tömbök mezői tartalmazhatnak függvényeket is. Így ezek a tömbök szintén tartalmazhatnak *eljárásokat* is. (lásd §2.5.9).

A tömbök, függvények, szálak és a (teljes) userdata értékek *objektumok*: a változók nem *tartalmazzák* ezeket az értékeket, csak a *hivatkozásukat*. Az értékadások, a paraméterátadások és a függvények visszatérési értékei mindig befolyásolják ezeknek az értékeknek a hivatkozásait; ezek a műveletek nem foglalnak magukban semmilyen másolatot.

A függvénykönyvtár `type` függvényének visszatérési értéke az adott érték típusát leíró karakterlánc.

2.2.1 - Átalakítások

A Lua automatikusan biztosít átalakítási lehetőséget a karakterlánc és a szám értékek között futási időben is. Bármilyen számtani művelet végrehajtásakor, amely karakterláncon hajtódna végre, a Lua az átalakítási szabályokat követve megpróbálja a karakterláncot számmá alakítani. Fordított esetben, amikor egy szám van használatban olyan helyen, ahol karakterláncnak kellene következnie, a szám karakterlánccá lesz alakítva, elfogadható formában. A számok karakterlánccá alakításához a string eljáráskönyvtár `format` függvénye használatos (lásd `string.format`).

2.3 - Változók

A változók azok a helyek, ahol az értékek tárolhatóak. A Lua nyelvben háromféle változó létezik: globális változók, lokális változók és tömb mezők.

Egy egyszerű név jelölhet globális vagy lokális változót is (vagy egy függvény szabályszerű paraméterét, ahol így egy egyéni lokális változót jelent):

```
var ::= Name
```

A név jelzi az azonosítót, a §2.1. pontban foglaltak szerint.

A változók alapértelmezés szerint globálisak, hacsak nem szándékosan lokálisként vannak deklarálva. (lásd §2.4.7). A lokális változók *lexikális kiterjedésűek:* az adott hatáskörön belül definiált függvényekben szabadon elérhetőek (lásd §2.6).

Egy változó az első értékadás előtt **nil** értékkel rendelkezik.

A szögletes zárójelek egy tömb indexét fejezik ki:

```
var ::= prefixexp `[´ exp `]´
```

A globális változók és tömb mezők elérésének módja megváltoztatható a metatömbök segítségével. Az indexelt `t[i]` változó elérése ugyanaz, mint a `gettable_event(t,i)` hívás. (lásd a §2.8 fejezetben a `gettable_event` függvény leírását. Ez a függvény nem definiálható és nem elérhető a Lua nyelvből, itt csak magyarázó jelleggel szerepel.)

A `var.Name` kifejezés ugyanazt jelenti, mint a `var["Name"]`:

```
var ::= prefixexp `.´ Name
```

Minden globális változó egy általános Lua tömb mezőjeként jön létre, amit *környezeti tömbök*nek vagy *környezetek* nevezünk (lásd §2.9). Minden egyes függvény saját hivatkozással kapcsolódik a környezethez, így az adott függvényben minden globális változó erre a környezeti tömbre fog hivatkozni. Egy függvény a létrehozásakor

megörökli a környezetet attól a függvénytől, amely azt létrehozta. Egy Lua függvény környezeti tömbjének lekérésére a `getfenv` függvény szolgál. Ennek megváltoztatására a `setfenv` hívás használható. (A C függvények környezetei csak a debug függvénytáron keresztül módosíthatóak (lásd §5.9).)

Az `x` globális változó elérése ugyanaz, mint az `_env.x` hívás, ami szintén egyenlő a

```
gettable_event(_env, "x")
```

hívással, ahol az `_env` a futó függvény környezete. (Lásd a §2.8 fejezetben a `gettable_event` függvény leírását. Ez a függvény nem definiálható és nem hívható meg a Lua-ból. Ehhez hasonlóan, az `_env` változó sem definiálható a Lua nyelvből, itt csak magyarázó jelleggel szerepel.)

2.4 - Utasítások

A Lua az utasítások majdnem egyezményes halmazát támogatja, a Pascal vagy C nyelvekhez hasonlóan. Ez a halmaz tartalmazza az értékadásokat, vezérlő struktúrákat, függvényhívásokat és változó deklarációkat.

2.4.1 - Csonkok

A végrehajtás mértékegysége a Lua nyelvben a *csonk*. A csonk az utasítások egyszerű sorozata, melyek végrehajtása sorrendben történik. Minden utasítást opcionálisan egy pontosvessző követhet:

```
chunk ::= {stat [`;´]}
```

Üres utasítás nem létezik, így a '`;;`' kifejezés érvénytelen.

A Lua a csonkot egy névtelen függvény testeként kezeli, melynek változó számú argumentuma van (lásd §2.5.9). Így a csonk létrehozhat lokális változókat, argumentumokat kaphat, és visszatérési értékkel is rendelkezhet.

A csonk tárolható fájlban vagy a host programban karakterláncként is. Egy csonk a végrehajtásakor először elő-fordítódik és utasításokká alakul a virtuális gép számára, majd a lefordított kódot végrehajtja a virtuális gép értelmezője.

A csonkok lehetnek elő-fordítottak is, bináris formában; további információkkal a `luac` program szolgál. A forráskódként és lefordított programként szereplő programok felcserélhetőek; a Lua automatikusan észleli a fájl típusát és ettől függően viselkedik.

2.4.2 - Blokkok

A blokk állítások sorozatából áll; a blokk szintaktikai felépítése ugyanaz, mint a csonké:

```
block ::= chunk
```

Egy blokk meghatározható nyílt módon is, ebben az esetben csak egy állítást eredményez:

```
stat ::= do block end
```

A nyílt blokkok a lokális deklarációk hatáskörének vezérlésekor lehetnek hasznosak. A nyílt blokkok arra is használhatóak, hogy egy másik blokkban **return** vagy **break** utasítást helyezzünk el (lásd §2.4.4).

2.4.3 - Értékadás

A Lua nyelvben engedélyezett a többszörös értékadás. Ebben az esetben az értékadás szintaktikája a következőképpen alakul: baloldalon szerepel a változók listája, jobboldalon pedig az értékek listája. Az elemek mindkét oldalon vesszővel vannak elválasztva:

```
stat ::= varlist1 `=´ explist1
varlist1 ::= var {`,´ var}
explist1 ::= exp {`,´ exp}
```

A kifejezések a §2.5. pontban vannak tárgyalva.

Az értékadás előtt az értékek listája a változók listájához lesz *igazítva*. Ha több érték van megadva a szükségesnél, a felesleges értékek figyelmen kívül lesznek hagyva. Ha kevesebb érték van a szükségesnél, az értékek listája ki lesz bővítve annyi **nil** értékkel, amennyi szükséges. Ha a kifejezések listája egy függvényhívással végződik, akkor ennek a hívásnak az összes visszatérési értéke a megadott változó(k)ba kerül(nek), még az igazítás előtt (kivéve amikor a hívás zárójelben szerepel; lásd §2.5).

Az értékadási utasítás először kiértékeli az összes kifejezést, majd csak ezután hajtja végre az értékadást. Így a következő kód:

```
i = 3
i, a[i] = i+1, 20
```

az `a[3]` mezőt 20-ra állítja anélkül, hogy az `a[4]` mező értékét módosítaná, mivel `i` az `a[i]` kifejezésben kiértékelődött (3-nak) még azelőtt, hogy a 4-es értéket megkapná. Hasonlóan, a következő sor:

```
x, y = y, x
```

felcseréli `x` és `y` értékét.

A globális változók és tömb mezők értékadásának tulajdonságai metatömbök segítségével megváltoztathatóak. Egy indexelt tömb értékadásakor a `t[i] = val` kifejezés megegyezik a `settable_event(t,i,val)` kifejezéssel. (Lásd a §2.8 fejezetben a `settable_event` függvény leírását. Ez a függvény nem definiálható és nem hívható meg a Lua-ból, itt csak magyarázó jelleggel szerepel.)

A globális változók értékadása (`x = val`) megegyezik az `_env.x = val` értékadással, amely megegyezik a

```
settable_event(_env, "x", val)
```

hívással, ahol `_env` a futó függvény környezete. (az `_env` változó nem definiálható a Lua nyelvből, itt csak magyarázó jelleggel szerepel.)

2.4.4 - Vezérlő szerkezetek

A vezérlő szerkezetek: az **if**, a **while**, és a **repeat** a szokásos tulajdonságokkal rendelkeznek és a már ismerős szintaktikát használják:

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
```

A Lua nyelvben is létezik **for** utasítás, két hatókörben is (lásd §2.4.5).

Egy vezérlő szerkezet feltételes kifejezése bármilyen visszatérési értékkel rendelkezhet. Mind a **false**, mind a **nil** érték hamisnak minősül. Minden **nil**-től és **false**-tól különböző érték igaznak minősül (így tehát, bármilyen szokatlan, a 0 szám és az üres karakterlánc is igaznak minősül).

A **repeat**–**until** ciklus nem ér véget az **until** kulcsszónál, hanem csak annak feltétele után. Emiatt a feltétel felhasználhat olyan lokális változókat is, amelyek a ciklus blokkjában lettek deklarálva.

A **return** utasítás szolgál arra, hogy az egyes függvények és csonkok (amely csak egy függvényből áll) visszatérési értékekkel rendelkezhessenek. A függvényeknek és a csonkoknak lehet több visszatérési értékük is, ebben az esetben a **return** utasítás szintakszisa a következő:

```
stat ::= return [explist1]
```

A **break** utasítás egy **while**, **repeat** vagy **for** ciklus megszakítására szolgál, a ciklus utáni következő utasításra ugorva:

```
stat ::= break
```

A **break** mindig a legbelső ciklust szakítja meg.

A **return** és **break** utasítások egy blokkban csak a *legutolsó* helyen szerepelhetnek. Ha nagyon fontos, hogy a **return** vagy a **break** a blokk közepén szerepeljen, akkor egy nyílt belső ciklust kell használni, mivel a `do return end` és a `do break end` kifejezésekben a **return** és **break** utasítások már az utolsó helyen szerepelnek a (belső) blokkjukban.

2.4.5 - For utasítás

A **for** utasításnak két formulája létezik: egy numerikus (számbeli) és egy generikus (általános).

A numerikus **for** ciklus addig ismétli a megadott kódrészletet, amíg a vezérlő változó végigfut egy számtani soron. Ennek a szintaktikája a következő:

```
stat ::= for Name `=´ exp `,´ exp [`,´ exp] do block end
```

A *block name*-ig ismétlődik, amelynek kezdő értéke az első *exp*, záró értéke a második *exp*, lépésköze pedig a harmadik *exp*. Pontosabban, a következő **for** utasítás:

```
for var = e1, e2, e3 do block end
```

megegyezik a következő kóddal:

```
do
    local _var, _limit, _step = tonumber(e1), tonumber(e2), tonumber(e3)
    if not (_var and _limit and _step) then error() end
    while (_step>0 and _var<=_limit) or (_step<=0 and _var>=_limit) do
        local var = _var
        block
        _var = _var + _step

    end
end
```

A következőket érdemes még megjegyezni:

- Mindhárom vezérlőkifejezés csak egyszer kerül kiértékelésre, a ciklus kezdete előtt. Mindhárom eredményének számnak kell lennie.
- A `_var`, a `_limit`, és a `_step` nem létező változók, csak magyarázó jelleggel szerepelnek.
- Ha a harmadik kifejezés (a lépésköz) hiányzik, 1-es lépésköz lesz használva.
- A **break** utasítással megszakítható a **for** ciklus.
- A ciklusban létrehozott `var` változó lokális a ciklus számára; nem használható a **for** vége után, vagy ha az félbeszakad. Ha szükség van erre a változóra, akkor egy másik változóba kell helyezni mielőtt a ciklus megszakad vagy véget ér.

A generikus **for** utasítás függvényeken használható, és *iterátor*nak nevezik. Minden egyes iterációkor az iterátor függvény lesz meghívva hogy egy új értéket generáljon, így a ciklus akkor áll meg, ha ez a visszatérési érték **nil**. A generikus **for** ciklus szintaktikája a következő:

```
stat ::= for namelist in explist1 do block end
namelist ::= Name {`,´ Name}
```

A következő **for** utasítás:

```
for var_1, ˇˇˇ, var_n in explist do block end
```

megegyezik a következő kóddal:

```
do
    local _f, _s, _var = explist
    while true do
        local var_1, ˇˇˇ, var_n = _f(_s, _var)
        _var = var_1
        if _var == nil then break end
        block
    end
end
```

A következőket érdemes még megjegyezni:

- Az `explist` csak egyszer kerül kiértékelésre. Ez egy *iterátor függvényt*, egy *állapotot* és az *iterátor változóhoz* tartozó kezdőértéket eredményez.
- Az `_f`, az `_s`, és az `_var` nem létező változók, a nevek itt csak magyarázó jelleggel szerepelnek.
- A **break** utasítással megszakítható a **for** ciklus.
- A ciklusban létrehozott `var_i` változó lokális a ciklus számára; nem használható a **for** vége után, vagy ha az félbeszakad. Ha szükség van erre a változóra, akkor egy másik változóba kell helyezni mielőtt a ciklus megszakad vagy véget ér.

2.4.6 - Függvényhívás utasításként

A lehetséges mellék-műveletek engedélyezéséhez a függvények utasításként is végrehajthatóak:

```
stat ::= functioncall
```

Ebben az esetben a visszatérési értékek figyelmen kívül lesznek hagyva. A függvényhívások részletesebben a §2.5.8 fejezetben vannak tárgyalva.

2.4.7 - Lokális deklarációk

A lokális változók egy blokkon belül bárhol deklarálhatóak. A deklaráció kezdeti értékadás is lehet egyben:

```
stat ::= local namelist [`=´ explist1]
```

Ha értékadás is történik, akkor annak szintaktikája megegyezik a többszörös értékadással (lásd §2.4.3). Egyébként minden változó **nil** értékkel lesz létrehozva.

A csonk is egyfajta blokk (lásd §2.4.1), így egy nyílt blokkon kívüli csonkban lokális változók deklarálhatóak. Ezeknek a lokális változóknak a hatóköre a csonk végéig terjed.

A lokális változók láthatósági szabályai a §2.6. részben találhatóak.

2.5 - Kifejezések

Az alap kifejezések a Lua nyelvben a következőek:

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= `...´
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | `(´ exp `)´
```

A számok és a literális karakterláncok a §2.1 fejezetben vannak leírva; a változók a §2.3 fejezetben; a függvénydefiníciók a §2.5.9 fejezetben; a függvényhívások a §2.5.8 fejezetben; a tömb konstruktorok a §2.5.7 fejezetben. A vararg kifejezések, amelyeket három pont jelöl ('. . .'), csak a vararg függvényekben használhatóak, ezek a §2.5.9 fejezetben vannak leírva.

A bináris operátorok számtani műveletei jelekből (lásd §2.5.1), relációs műveleti jelekből (lásd §2.5.2), logikai operátorokból (lásd §2.5.3), és az összefűző operátorból (lásd §2.5.4) állnak. Az unáris operátorok csoportjába tartozik az unáris minusz (lásd §2.5.1), az unáris **not** (lásd §2.5.3), valamint az unáris *hossz operátor* (lásd §2.5.5).

Mind a függvényhívások, mind a vararg kifejezések több értéket is eredményezhetnek. Ha a kifejezés utasításként fut le (lásd §2.4.6) (csak függvényhívásoknál elérhető), akkor a visszatérési lista nulla elemre korlátozódik, tehát az összes visszatérési érték figyelmen kívül lesz hagyva. Ha egy kifejezés egy másik kifejezésen belül, vagy egy kifejezés-lista közepén szerepel, akkor a visszatérési értékek csak egy elemre korlátozódnak, így az elsőn kívül az összes visszatérési érték figyelmen kívül lesz hagyva. Ha egy kifejezés a kifejezéslista utolsó eleme, akkor nem hajtódik végre a korlátozás, kivéve, ha a hívás zárójelekkel történik.

Következzen néhány példa:

```
 f() -- 0 eredményre korlátozva
 g(f(), x) -- f() 1 eredményre korlátozva
 g(x, f()) -- a g x-et és az f() függvény visszatérési értékeit kapja
 a,b,c = f(), x -- f() 1 eredményre korlátozva (c nil lesz)
 a,b = ... -- a kapja az első vararg paramétert, b kapja
           -- a másodikat (a és b is lehet nil, ha nincs megfelelő vararg
paraméter)
 a,b,c = x, f() -- f() 2 eredményre korlátozva
 a,b,c = f() -- f() 3 eredményre korlátozva
 return f() -- f() összes visszatérési értéke visszatér
 return ... -- Az összes fogadott vararg paraméter visszatér
 return x,y,f() -- visszatérési értéke x, y, és az f() függvény
visszatérési értékei
 {f()} -- visszatérési értéke az f() függvény visszatérési értékeiből
alkotott tömb
 {...} -- visszatérési értéke a vararg paraméterekből alkotott tömb
 {f(), nil} -- f() 1 eredményre korlátozva
```

Egy zárójelbe tett kifejezés mindig csak egy értékkel tér vissza. Így a `(f(x,y,z))` akkor is csak egy értékkel tér vissza, ha az `f` függvény egyébként több visszatérési

értékkel is rendelkezik. (Az `(f(x,y,z))` hívás eredménye tehát az `f` függvény első visszatérési értéke, vagy **nil**, ha nincs visszatérési érték.)

2.5.1 - Számtani műveleti jelek

A Lua nyelvben elérhetőek a szokásos számtani műveleti jelek: a bináris + (összeadás), `-` (kivonás), `*` (szorzás), `/` (osztás), `%` (modulo), és a `^` (hatványozás); az unáris `-` (negáció). Ha az operandusok számok, vagy olyan karakterláncok, amelyek számmá alakíthatóak (lásd §2.2.1), akkor a műveletek a szokásos tulajdonságokkal rendelkeznek. Hatványozáskor bármilyen hatványkitevő használható. Például, az `x^(-0.5)` kifejezés `x` négyzetgyökének negáltját eredményezi. A modulo művelet definíciója a következő:

```
a % b == a - math.floor(a/b)*b
```

Így ez az osztás maradékát adja eredményül, amely a hányadost a mínusz végtelen irányába kerekíti.

2.5.2 - Relációs műveleti jelek

A Lua nyelv relációs műveleti jelei a következőek:

```
== ~= < > <= >=
```

Ezek az operátorok mindig **false** vagy **true** értéket eredményeznek.

Az egyenlőség (`==`) operátor először összehasonlítja az operandusok típusát. Ha az eredmények eltérőek, az eredmény **false**. Egyéb esetben az operandusok értékei kerülnek összehasonlításra. A számok és a karakterláncok összehasonlítása a szokványos módon történik. Az objektumok (table, userdata, thread típusok és függvények) a hivatkozásaik által lesznek összehasonlítva: két objektum csak akkor tekinthető egyenlőnek, ha mindkettő *ugyanaz* az objektum. Minden alkalommal, amikor egy új objektum létrejön (table, userdata, thread típus vagy függvény), ez az új objektum mindig különbözni fog az előzően létrehozott objektumoktól.

A Lua table és userdata típusainak összehasonlítása megváltoztatható a "eq" metaeljárás használatával (lásd §2.8).

A §2.2.1 pontban szereplő átalakítási szabályok *nem vonatkoznak* az egyenlőségi összehasonlításra. Emiatt a `"0"==0` értéke **false**, valamint a `t[0]` és a `t["0"]` azonosítók két különböző mezőt határoznak meg a tömbben.

A `~=` művelet az egyenlőség (`==`) tagadása.

A rendező operátorok a következőképpen működnek: ha mindkét argumentum szám, akként lesznek összehasonlítva. Ha mindkét paraméter karakterlánc, az összehasonlítás a helyi beállításoknak megfelelően fog lezajlani. Egyéb esetben a Lua megpróbálja meghívni az "lt" vagy az "le" metaeljárást (lásd §2.8).

2.5.3 - Logikai operátorok

A logikai operátorok a Lua nyelvben az **and** (és), **or** (vagy), és a **not** (nem). A vezérlő szerkezetekhez hasonlóan (lásd §2.4.4) minden logikai operátor a **false**-t és a **nil**-t hamisnak, minden egyéb értéket igaznak tekint.

A **not** negációs operátor mindig **false** vagy **true** értékkel tér vissza. Az **and** konjunktív (egyesítő) operátor visszatérési értéke az első argumentuma, ha az **false** vagy **nil**; egyéb esetben az **and** visszatérési értéke a második paraméter. Az **or** diszjunktív (szétválasztó) operátor visszatérési értéke az első argumentum, ha annak értéke **nil**-től és **false**-tól különbözik, egyébként a visszatérési értéke a második argumentum. Mind az **and**, mind az **or** gyorsított kiértékelést használ, azaz a második operandus csak akkor kerül kiértékelésre, ha az valóban szükséges. Következzen néhány példa:

```
10 or 20 --> 10
10 or error() --> 10
nil or "a" --> "a"
nil and 10 --> nil
false and error() --> false
false and nil --> false
false or nil --> nil
10 and 20 --> 20
```

(a fenti leírásban a --> jelzi az előtte lévő kifejezés eredményét.)

2.5.4 - Összefűzés

A karakterláncok összefűzésére szolgáló operátort a Lua nyelvben két ponttal jelölik ('..'). Ha mind a két operandus karakterlánc vagy szám, akkor azok a §2.2.1 pontban leírt szabályok szerint karakterláncokká lesznek átalakítva. Egyéb esetben a "concat" metaeljárás lesz meghívva (lásd §2.8).

2.5.5 - A hossz operátor

A hossz operátort az unáris `#` operátor jelöli. Egy karakterlánc hossza az elfoglalt bájtok száma (mivel minden karakter egy bájtnak felel meg).

A `t` tömb hosszának értéke bármilyen `n` egész szám lehet, amely megfelel annak a feltételnek, hogy `t[n]` nem **nil** és `t[n+1]` **nil**; továbbá ha `t[1]` **nil**, `n` értéke nulla is lehet. Egy általános tömb esetén, ahol 1-től az adott `n`-ig nem nil értékek szerepelnek, a hossz értéke pontosan `n`, azaz az utolsó érték indexe. Ha a tömb "lyukas" (azaz **nil** érték(ek) vannak nem-nil értékek között), akkor a `#t` értéke bármelyik **nil** értéket megelőző index lehet (tehát az első **nil** érték a tömb végét jelenti).

2.5.6 - Precedencia

A műveleti jelek precedenciája (elsősége, sorrendje) a Lua nyelvben a következő táblázatot követi, az alacsonyabbtól a magasabb prioritás felé haladva:

```
or
and
```

```
< > <= >= ~= ==
..
+ -
* / %
not # - (unáris)
^
```

Természetesen zárójelek használatával módosítható a kifejezések sorrendje. Az összefűző ('. .') és a hatványozási ('^') operátorok jobbról asszociatívak. Minden más bináris operátor balról asszociatív.

2.5.7 - Tömb konstruktorok

A tömb konstruktorok olyan kifejezések, amelyik tömböket hoznak létre. Minden alkalommal, amikor egy konstruktor kiértékelődik, létrejön egy tömb. A konstruktorok segítségével üres tömbök hozhatóak létre, vagy olyanok, amelyek előre meghatározott mezőket tartalmaznak. A konstruktorok általános szintakszisa a következő:

```
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
fieldsep ::= `,´ | `;´
```

Minden egyes `[exp1] = exp2` kifejezés egy új bejegyzést ad a tömbhöz, `exp1` kulccsal és `exp2` értékkel. A `name = exp` formula ugyanazt jelenti, mint amit a `["name"] = exp`. Végül, az `exp` mezők megegyeznek az `[i] = exp` kifejezéssel, ahol i egy 1-től induló, egymást követő egész számsorozatot jelent. Más formátumú mezők nem befolyásolják ezt a számolást. Például:

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

ugyanaz, mint a következő kód:

```
do
    local t = {}
    t[f(1)] = g
    t[1] = "x" -- 1st exp
    t[2] = "y" -- 2nd exp
    t.x = 1 -- t["x"] = 1
    t[3] = f(x) -- 3rd exp
    t[30] = 23
    t[4] = 45 -- 4th exp
    a = t
end
```

Ha a lista utolsó mezőjének exp formulája van és a kifejezés egy függvényhívás vagy vararg kifejezés, akkor ezeknek a kifejezéseknek a visszatérési értékei folyamatosan beépülnek ebbe a listába (lásd §2.5.8). Ez megelőzhető azzal, ha a függvényhívást (vagy a vararg kifejezést) zárójelbe téve hívjuk meg (lásd §2.5).

A mezőlistának lehet egy opcionális elválasztója, kényelmi funkcióként a gép által generált kód számára.

2.5.8 - Függvényhívások

A függvényhívásnak a Lua nyelvben a következő a szintaktikája:

```
functioncall ::= prefixexp args
```

A függvény hívásakor először az előtag (prefixexp) és az argumentumok lesznek kiértékelve. Ha az előtag típusa *function*, akkor a függvény a megadott argumentumokkal meghívódik. Ellenkező esetben az előtag "call" metaeljárása lesz meghívva, melynek első paramétere az előtag értéke, amelyet az eredeti argumentumok követnek (lásd §2.8).

A következő formula:

```
functioncall ::= prefixexp `:´ Name args
```

eljárások meghívására használható. A `v:name(args)` hívás ugyanaz, mint a `v.name(v,args)`, azzal a különbséggel, hogy a `v` csak egyszer lesz kiértékelve.

Az argumentumok szintaktikája a következő:

```
args ::= `(´ [explist1] `)´
args ::= tableconstructor
args ::= String
```

A hívás előtt minden argumentum kifejezés kiértékelődik. Az `f{fields}` formátumú hívás megfelel az `f({fields})` kifejezésnek; ebben az esetben az argumentumok listája egy új tömbnek felel meg. Az `f'string'` (vagy `f"string"` vagy `f[[string]]`) formátumú hívás megegyezik az `f('string')` hívással; ebben az esetben az argumentumok listája egy literális karakterláncnak felel meg.

Egyetlen kivétel a háromféle Lua szintakszis alól, hogy nem használható újsor karakter a '`(`' jel előtt függvényhívásnál. Ez a korlátozás megelőz néhány félreértést a nyelvben. A következő kódrészlet

```
a = f
(g).x(a)
```

a Lua számára egyetlen utasítást fog jelenteni, `a = f(g).x(a)`. Így, ha két utasításként szeretnénk a fenti kódot felhasználni, pontosvesszővel kell elválasztani őket. Ha az `f` függvényt közvetlenül szeretnénk meghívni, el kell távolítani az újsor karaktert a `(g)` elől.

A `return` *függvényhívás* formátum neve *véghívás*. A Lua *tökéletes véghívást* (vagy *tökéletes végújrahívást*) valósít meg: egy véghívásban a hívott függvény újra felhasználja a hívó függvény verem bemenetét. Emiatt nincs korlátozás a program által végrehajtható, egymásba ágyazott véghívások számát illetően. Azonban a véghívás minden hibakeresési információt töröl a hívó függvényről. A véghívás csak egyéni szintakszis esetén fordulhat elő, ahol a **return** argumentuma csak egy függvényhívás; ez a szintakszis arra készteti a hívó függvényt, hogy a hívott

függvény visszatérési értékeivel térjen vissza. A következő példában véghívások szerepelnek:

```
return (f(x)) -- az eredmények száma 1-re korlátozva
return 2 * f(x)
return x, f(x) -- további eredmények
f(x); return -- a visszatérési értékek figyelmen kívül hagyva
return x or f(x) -- az eredmények száma 1-re korlátozva
```

2.5.9 - Függvénydefiníciók

A függvénydefiníció szintakszisa a következő:

```
function ::= function funcbody
funcbody ::= `(´ [parlist1] `)´ block end
```

A következő szintakszis leegyszerűsíti a függvénydefiníciókat:

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {`.´ Name} [`:´ Name]
```

A következő kifejezés:

```
function f () body end
```

a következőképpen fordítható:

```
f = function () body end
```

A következő kifejezés:

```
function t.a.b.c.f () body end
```

pedig a következőképpen:

```
t.a.b.c.f = function () body end
```

A következő kifejezés

```
local function f () body end
```

megfelel a következőnek:

```
local f; f = function () body end
```

és nem pedig ennek:

```
local f = function () body end
```

(Ez a különbség csak akkor számít, ha a függvénytest `f`-re hivatkozik.)

A függvénydefiníció egy végrehajtható kifejezés, amely értékének típusa *function*. Amikor a Lua elő-fordít egy csonkot, akkor annak az összes függvényteste előfordításra kerül. Így, amikor a Lua végrehajt egy függvénydefiníciót, a függvény *véglegesítődik* (vagy *lezáródik*). Ez a függvényhivatkozás (vagy zárlat) a kifejezés végleges értéke lesz. Azonos függvények különböző hivatkozásai különböző külső lokális változókat is elérhetnek, valamint különböző környezeti tömbjeik is lehetnek.

A paraméterek lokális változókként viselkednek, amelyek az argumentum értékeivel jönnek létre:

```
parlist1 ::= namelist [`,´ `...´] | `...´
```

Egy függvény meghívásakor az argumentumok listája a paraméterek listájának számához igazodik, kivéve ha a függvény variadikus vagy vararg függvény, amelyet a paraméterlista végén három pont jelöl ('. . .'). A vararg függvény nem korlátozza az argumentumok listáját, éppen ellenkezőleg, összegyűjti az összes extra argumentumot és a három pont formában leírt *vararg kifejezésen* keresztül a függvényhez rendeli ezeket. Ennek a kifejezésnek az értéke az aktuális extra argumentumok listája, a több visszatérési értékkel rendelkező függvényhez hasonlóan. Ha a vararg kifejezés egy másik kifejezésen belül szerepel, vagy a kifejezések közben, akkor a visszatérési lista egy elemre korlátozódik. Ha a kifejezés a lista utolsó eleme, akkor nem lesz korlátozás végrehajtva (kivéve ha a hívás zárójelbe téve történik).

Példaként itt szerepel néhány ilyen definíció:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Ezek után a következő argumentumok hozzárendelése a paraméterekhez és a vararg kifejezéshez a következőket eredményezi:

```
PARAMÉTEREK HÍVÁSA

f(3) a=3, b=nil
f(3, 4) a=3, b=4
f(3, 4, 5) a=3, b=4
f(r(), 10) a=1, b=10
f(r()) a=1, b=2
g(3) a=3, b=nil, ... --> (semmi)
g(3, 4) a=3, b=4, ... --> (semmi)
g(3, 4, 5, 8) a=3, b=4, ... --> 5 8
g(5, r()) a=5, b=1, ... --> 2 3
```

Az eredmények a **return** utasítás használatával tértek vissza (lásd §2.4.4). Ha a vezérlés úgy éri el a függvény végét, hogy közben nem talál **return** utasítást, akkor a függvénynek nincs visszatérési értéke.

A *kettőspont* jellel *eljárások* határozhatóak meg, ebben az esetben a függvénynek van egy implicit extra paramétere, a `self`. Így a következő implicit utasítás

```
function t.a.b.c:f (params) body end
```

megfelel a következőnek:

```
t.a.b.c.f = function (self, params) body end
```

2.6 - Láthatósági szabályok

A Lua egy lexikális hatókörű nyelv. A változók hatóköre a deklarációk *utáni* első utasítással kezdődik, és a deklarációt tartalmazó legbelső blokk végéig tart. Következzen egy példa:

```
x = 10 -- globális változó
do -- új blokk
    local x = x -- új 'x', értéke 10
    print(x) --> 10
    x = x+1
    do -- újabb blokk
        local x = x+1 -- egy újabb 'x'
        print(x) --> 12
    end
    print(x) --> 11
end
print(x) --> 10 (a globális)
```

Jegyezzük meg, hogy a `local x = x` típusú deklarációk esetén az új `x` még nem lesz deklarálva ebben a hatókörben, és így a második `x` a külső változóra hivatkozik.

A lexikális láthatósági szabályok miatt a lokális változókat az őket deklaráló függvények szabadon elérhetik. Egy belső függvény által használt lokális változó neve *upvalue* (felső érték), vagy *külső lokális változó*, a belső függvényen belül.

Ne felejtsük, hogy a **local** utasítás minden egyes végrehajtásakor új lokális változók jönnek létre. Nézzük a következő példát:

```
a = {}
local x = 20
for i=1,10 do
    local y = 0
    a[i] = function () y=y+1; return x+y end
end
```

A ciklus tízszer hajtódik végre (azaz, tíz alkalommal fut le a névtelen függvény). Minden egyes lépés egy különböző `y` változót használ, míg az `x` minden lépésben ugyanaz lesz.

2.7 - Hibakezelés

Mivel a Lua egy beágyazott kiterjesztett nyelv, minden Lua művelet a host program C kódjából indul, amely meghívja a Lua eljáráskönyvtár egyik függvényét (lásd `lua_pcall`). Amikor a Lua hibát észlel a fordítás vagy a futtatás közben, a vezérlés visszatér a C-hez, ami megteheti a szükséges intézkedéseket (például kiírja a hibaüzenetet).

A Lua kód közvetlenül is generálhat hibát az `error` függvény meghívásával. Ha a hibákat a Lua-n belül szeretnénk lekezelni, használjuk a `pcall` függvényt.

2.8 - Metatömbök

A Lua-ban minden értéknek lehet *metatömbje*. Ez a *metatömb* egy átlagos Lua tömb, amely megváltoztatja az eredeti érték viselkedését egyes meghatározott műveletek közben. A metatömb megfelelő mezőinek beállításával egyes műveletek jó néhány tulajdonsága megváltoztatható. Például amikor egy összeadás hajtódik végre egy nem szám típusú értéken, a Lua ellenőrzi a változó metatömbjének `"__add"` mezőjét. Ha a Lua talál ilyet, akkor meghívja ezt a függvényt, hogy hajtsa végre az összeadást.

A metatömbök kulcsait *eseménynek*, az értékeit *metaeljárásnak* nevezzük. Az előző példában az esemény az `"add"`, a metaeljárás pedig a függvény, amely végrehajtja az összeadást.

A `getmetatable` függvény segítségével a metatömbök bármikor megtekinthetőek.

A `setmetatable` függvény használatával a metatömbök lecserélhetőek. Más típusok metatömjbei nem megváltoztathatóak a Lua-ból (kivéve a debug eljáráskönyvtárból); erre a C API-t kell használni.

A tömböknek és az userdata típusoknak egyéni metatömbjeik vannak (noha több tömb és userdata is osztozhat ugyanazon a metatömbön); az összes többi értéknek típusonként csak egy metatömbje lehet. Így csak egy metatömbje lehet az összes karakterláncnak, egy az összes számnak, stb.

Egy metatömb befolyásolhatja egy objektum viselkedését a számtani műveletek, a rendező összehasonlítások, az összefűzések, a hossz operátor használata és az indexelés közben. Egy metatömb azt is meghatározhatja, hogy egy függvény lefusson, amikor a szemétgyűjtő algoritmus lefut egy userdata típuson. A Lua ezekhez a műveletekhez egy speciális kulcsot rendel, amit *eseménynek* nevezünk. Amikor a Lua végrehajt egy ilyen műveletet egy értéken, ellenőrzi, hogy az érték metatömbjében szerepel -e a megfelelő esemény. Ha igen, akkor a kulcshoz rendelt érték (a metaeljárás) szabja meg, hogy a Lua hogyan hajtsa végre a műveletet.

A metatömbök a lenti listában szereplő műveleteket tudják befolyásolni. Minden műveletet a megfelelő név azonosít. A műveletekhez tartozó kulcs egy kettős aláhúzással ('__') kezdődő karakterlánc; például az "add" (összeadás) művelethez az `"__add"` kulcs tartozik. Ezeknek a műveleteknek jelentéstartalma jobban értelmezhető egy Lua függvényen keresztül, amelyik megadja az értelmezőnek a művelet végrehajtásának menetét.

Az itt szereplő Lua kódrészletek csak illusztrációk; a valódi viselkedés az értelmezőben bonyolultabb kódokkal lett definiálva és az sokkal hatásosabb mint ez a szimuláció. Minden, ebben a leírásban szereplő függvény (`rawget`, `tonumber`, stb.) részletesen le van írva az §5.1 fejezetben. Egy adott objektum metaeljárásának lekérésére a következő kifejezés használható:

```
metatable(obj)[event]
```

Amely így olvasandó:

```
rawget(getmetatable(obj) or {}, event)
```

Tehát a metaeljárások lekérése nem hív meg másik metaeljárást, és a metatömb nélküli objektumok elérése sem lesz sikertelen (az eredmény egyszerűen **nil** lesz).

- **"add":** a `+` művelet.

  Az alább szereplő `getbinhandler` függvény határozza meg, hogy a Lua hogyan választja ki a bináris művelet kezelőjét. Először a Lua megvizsgálja az első operandust. Ha ennek típusa nem határoz meg kezelőt a művelethez, akkor a Lua a második operandust vizsgálja meg.

  ```
  function getbinhandler (op1, op2, event)
      return metatable(op1)[event] or metatable(op2)[event]
  end
  ```

  Ennek a függvénynek a használatával az `op1 + op2` művelet viselkedése a következő lesz:

  ```
  function add_event (op1, op2)
      local o1, o2 = tonumber(op1), tonumber(op2)
      if o1 and o2 then -- mindkét operandus szám?
          return o1 + o2 -- a '+' itt egyszerű összeadást jelent
      else -- legalább az egyik operandus nem szám
          local h = getbinhandler(op1, op2, "__add")
          if h then
              -- a kezelő meghívása mindkét operandushoz
              return h(op1, op2)
          else -- nincs elérhető kezelő: alapértelmezett viselkedés
              error(˘˘˘)
          end
      end
  end
  ```

- **"sub":** a `-` művelet. A viselkedése hasonló az "add" (összeadás) művelethez.
- **"mul":** a `*` művelet. A viselkedése hasonló az "add" (összeadás) művelethez.
- **"div":** a `/` művelet. A viselkedése hasonló az "add" (összeadás) művelethez.
- **"mod":** a `%` művelet. A viselkedése hasonló az "add" (összeadás) művelethez, amely megfelel az `o1 - floor(o1/o2)*o2` primitív műveletnek.
- **"pow":** az `^` (exponencialitás) művelet. A viselkedése hasonló az "add" (összeadás) művelethez, amely megfelel a `pow` (a C math programkönyvtárából) primitív műveletnek.
- **"unm":** az unáris `-` művelet.

  ```
  function unm_event (op)
      local o = tonumber(op)
      if o then -- az operandus szám?
          return -o -- '-' a primitív 'unm'
      else -- az operandus nem szám.
          -- kezelő keresése az operandushoz
  ```

```
        local h = metatable(op).__unm
        if h then
            -- kezelő meghívása az operandussal
            return h(op)
        else -- nincs elérhető kezelő: alapértelmezett viselkedés
            error(˘˘˘)
        end
    end

end
```

- **"concat":** a `..` (összefűzés) művelet.

```
function concat_event (op1, op2)
    if (type(op1) == "string" or type(op1) == "number") and
   (type(op2) == "string" or type(op2) == "number") then
        return op1 .. op2 -- primitív karakterlánc-összefűzés
    else
        local h = getbinhandler(op1, op2, "__concat")
        if h then
            return h(op1, op2)
        else
            error(˘˘˘)
        end
    end
end
```

- **"len":** a `#` művelet.

```
function len_event (op)
    if type(op) == "string" then
        return strlen(op) -- primitív karakterlánc hossz
    elseif type(op) == "table" then
        return #op -- primitív tömb hossz
    else
        local h = metatable(op).__len
        if h then
            -- kezelő meghívása az operandussal
            return h(op)
        else -- nincs elérhető kezelő: alapértelmezett viselkedés
            error(˘˘˘)
        end
    end
end
```

A tömb hosszának bővebb leírása a §2.5.5 fejezetben található.

- **"eq":** az `==` művelet. A `getcomphandler` függvény határozza meg, hogy válasszon a Lua az összehasonlítási operátorokhoz metaeljárást. A metaeljárás csak akkor használható, ha mindkét összehasonlítandó objektumnak azonos a típusa, valamint ugyanazzal a metaeljárással rendelkeznek a kiválasztott művelethez.

```
function getcomphandler (op1, op2, event)
    if type(op1) ~= type(op2) then return nil end
    local mm1 = metatable(op1)[event]
    local mm2 = metatable(op2)[event]
    if mm1 == mm2 then return mm1 else return nil end
```

```
end
```

Az "eq" esemény definiálása a következő:

```
function eq_event (op1, op2)
    if type(op1) ~= type(op2) then -- különböző típusúak?
        return false -- különböző objektumok
    end
    if op1 == op2 then -- primitív egyenlőség?
        return true -- az objektumok ugyanazok
    end
    -- metaeljárás keresése
    local h = getcomphandler(op1, op2, "__eq")
    if h then
        return h(op1, op2)
    else
        return false
    end
end
```

Az `a ~= b` kifejezés ugyanaz, mint `not (a == b)`.

- **"lt":** a `<` művelet.

```
function lt_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
        return op1 < op2 -- számbeli összehasonlítás
    elseif type(op1) == "string" and type(op2) == "string" then
        return op1 < op2 -- sorfolytonos összehasonlítás
    else
        local h = getcomphandler(op1, op2, "__lt")
        if h then
            return h(op1, op2)
        else
            error(˘˘˘);
        end
    end
end
```

Az `a > b` kifejezés ugyanaz, mint a `b < a`.

- **"le":** a `<=` művelet.

```
function le_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
        return op1 <= op2 -- számbeli összehasonlítás
    elseif type(op1) == "string" and type(op2) == "string" then
        return op1 <= op2 -- sorfolytonos összehasonlítás
    else
        local h = getcomphandler(op1, op2, "__le")
        if h then
            return h(op1, op2)
        else
            h = getcomphandler(op1, op2, "__lt")
            if h then
                return not h(op2, op1)
            else
                error(˘˘˘);
```

```
            end
        end
    end
end
```

Az `a >= b` kifejezés ugyanaz, mint a `b <= a`. Megjegyzés: a "le" metaeljárás hiányában a Lua megpróbálja az "lt" metaeljárást, azzal a feltétellel, hogy az `a <= b` kifejezés ugyanaz, mint a `not (b < a)`.

- **"index":** A `table[key]` formátumú tömbelérés.

```
function gettable_event (table, key)
    local h
    if type(table) == "table" then
        local v = rawget(table, key)
        if v ~= nil then return v end
        h = metatable(table).__index
        if h == nil then return nil end
    else
        h = metatable(table).__index
        if h == nil then
            error(˘˘˘);
        end
    end
    if type(h) == "function" then
        return h(table, key) -- kezelő hívása
    else return h[key] -- vagy a művelet megismétlése
    end
end
```

- **"newindex":** A `table[key] = value` formátumú tömbmező értékadás.

```
function settable_event (table, key, value)
    local h
    if type(table) == "table" then
        local v = rawget(table, key)
        if v ~= nil then rawset(table, key, value); return end
        h = metatable(table).__newindex
        if h == nil then rawset(table, key, value); return end
    else
        h = metatable(table).__newindex
        if h == nil then
            error(˘˘˘);
        end
    end
    if type(h) == "function" then
        return h(table, key,value) -- kezelő hívása
    else h[key] = value -- vagy a művelet megismétlése
    end
end
```

- **"call":** akkor hajtódik végre, amikor a lua meghív egy értéket.

```
function function_event (func, ...)
    if type(func) == "function" then
        return func(...) -- primitív hívás
    else
        local h = metatable(func).__call
```

```
            if h then
                return h(func, ...)
            else
                error(˘˘˘)
            end
        end
    end
```

2.9 - Környezetek

A thread, function és userdata típusú objektumok a metatömbök mellett rendelkeznek még egy hozzájuk rendelt tömbbel, amit a *környezetüknek* nevezünk. A metatömbökhöz hasonlóan a környezetek is átlagos Lua tömbök, valamint több objektum is osztozhat ugyanazon a környezeten.

A userdata típushoz rendelt környezeteknek nincs jelentősége a Lua számára. Ez csak egy kényelmi funkció a programozók számára, hogy egy tömböt egy userdata típushoz társítsanak.

A szálakhoz rendelt környezetek neve *globális környezet*. Ezek az alapértelmezett környezetek a szál által létrehozott további szálak és nem-beágyazott függvények számára (a `loadfile`, `loadstring` vagy `load` hívásokon keresztül), valamint a C kód által közvetlenül is elérhetőek (lásd §3.3).

A C függvényekhez társított környezetek a C kódból közvetlenül elérhetőek (lásd §3.3). Ezek az alapértelmezett környezetei a függvény által létrehozott további C függvényeknek.

A Lua függvényekhez társított környezetek arra használatosak, hogy a függvényben felhasznált globális változók elérése feloldható legyen (lásd §2.3). Ezek az alapértelmezett környezetei a függvény által létrehozott további Lua függvényeknek.

Egy Lua függvény vagy egy futó szál környezete a `setfenv` hívással megváltoztatható, illetve a `getfenv` hívással lekérhető. A többi objektum környezetének módosításához (userdata, C függvény, más szálak) a C API-t kell használni.

2.10 - Szemétgyűjtés

A Lua automatikus memóriakezeléssel rendelkezik. Ez azt jelenti, hogy programozónak nem kell foglalkoznia sem a memória-lefoglalással, sem annak felszabadításával, ha az objektumra már nincs többé szükség. A Lua memóriakezelése úgy zajlik, hogy időről-időre lefuttatja a *szemétgyűjtőt*, amely összegyűjti az *élettelen* (nem használt) *objektumokat* (tehát ezek az objektumok többé nem lesznek elérhetőek a Lua-ból). A Lua-ban minden objektumra vonatkozik az automatikus kezelés: a tömbökre, a userdata típusokra, a függvényekre, a szálakra és a karakterláncokra is.

A Lua ún. *megjelöl-és-seper* gyűjtést végez. Két számot használ a szemétgyűjtő-körök szabályozására: a *szemétgyűjtő szünetet* és a *szemétgyűjtő lépésszorzót*.

A szemétgyűjtő szünet szabályozza, hogy mekkora szünetet tartson az egyes szemétgyűjtő körök indítása között. Magasabb értékek mellett a gyűjtés nem lesz annyira erőteljes. 1-nél kisebb érték esetén a gyűjtő nem vár az egyes körök indítása között. 2-es értéknél a gyűjtő addig vár, amíg az összmemória-használat a kör indítása előtti érték duplája lesz.

A lépésszorzó szabályozza a gyűjtő relatív memóriafelszabadítási sebességét. Magasabb érték mellett a gyűjtő erőteljesebb, de minden lépés méretét növeli. 1-nél kisebb érték a gyűjtőt túl lassúvá teszi, és azt eredményezheti, hogy a gyűjtő soha nem fejezi be a kört. Az alapértelmezett érték 2 jelenti azt, hogy a gyűjtő sebessége kétszerese a memória-lefoglalásnak.

Ezek a számok megváltoztathatóak a C kódból a `lua_gc` hívással, illetve a Lua kódból a `collectgarbage` hívással. Mindkettő százalékpontot kap paraméterként (így a 100-as argumentum valódi értéke 1). Ezekkel a függvényekkel a gyűjtő is közvetlenül szabályozható (pl. megállítható és újraindítható).

2.10.1 - Szemétgyűjtési metaeljárások

A C API használatával szemétgyűjtési metaeljárások rendelhetőek a userdata típusú objektumokhoz (lásd §2.8). Ezek a metaeljárások *véglegesítő* néven is ismertek. Ezek segítségével a Lua szemétgyűjtője összehangolható külső erőforrás-kezelőkkel is (például fájlok bezárása, hálózati vagy adatbázis kapcsolatok lezárása, vagy a saját memória felszabadítása).

Azok a felszabadításra jelölt userdata típusú objektumok, amelyeknek a metatömbjében szerepel a `__gc` mező, nem kerülnek egyből törlésre, hanem a Lua egy tömbbe helyezi őket. A gyűjtőkör lefutása után a Lua a következő függvénnyel egyenértékű műveletet hajt végre minden egyes tömbben lévő userdata objektummal:

```
function gc_event (udata)
    local h = metatable(udata).__gc
    if h then
         h(udata)
    end
end
```

Minden egyes szemétgyűjtő-kör befejezésekor a userdata objektumok véglegesítői a létrehozásukhoz képest *fordított* sorrendben vannak meghívva, beleértve azokat is, amelyek abban a körben lettek összegyűjtve. Így tehát az első véglegesítő meghívásakor a hozzárendelt userdata objektum az, amelyik a programban az utolsóként lett létrehozva.

2.10.2 - Gyenge tömbök

A *gyenge tömb* egy olyan tömb, amelynek elemei *gyenge hivatkozások*. Ezeket a szemétgyűjtő figyelmen kívül hagyja. Másképpen kifejezve, ha egy objektumhoz csak gyenge hivatkozások tartoznak, akkor a szemétgyűjtő törli ezt az objektumot.

Egy gyenge tömbnek lehet gyenge kulcsa, gyenge értéke, vagy mindkettő egyszerre. A gyenge kulcsokkal rendelkező tömb esetén ezek a kulcsok törlésre kerülnek, azonban az értékeik nem. Ha a tömbben mind a kulcs, mind az érték gyenge, akkor sem a kulcs, sem az érték nem kerül törlésre. Minden egyéb esetben, ha a kulcs vagy az érték eltávolításra kerül, az egész pár törölve lesz a tömbből. Egy tömb gyengeségét a metatömbjében lévő `__mode` mező szabályozza. Ha a `__mode` mező értéke egy olyan karakterlánc, amely tartalmazza a '`k`' karaktert, a tömb kulcsai, ha a '`v`' karaktert, akkor pedig az értékei gyengék.

Miután egy tömb metatömbként lesz használva, nem tanácsos a `__mode` mező módosítása, különben azon gyenge tömbök viselkedése, amelyeket ez a metatömb szabályoz, kiszámíthatatlan lesz.

2.11 - Korutinok

A Lua támogatja a korutinokat, másnéven az *együttműködésen alapuló többszálú működést*. A korutin a Lua nyelvben egy végrehajtás különálló szálát képviseli. A több szálat támogató rendszerek szálaitól eltérően a korutin csak szünetelteti a végrehajtást a yield függvény hívásával.

Egy korutin a `coroutine.create` hívással hozható létre, amelynek egyetlen paramétere a korutin fő függvénye. A `create` függvény csak létrehozza az új korutint, és visszatér a kezelőjével (egy *thread* típusú objektummal), de nem indítja el annak végrehajtását.

A `coroutine.resume` függvény első hívásakor, amelynek első argumentuma a `coroutine.create` hívásból származó szál, a korutin megkezdi a fő függvény első sorának végrehajtását. A fő függvény megkapja a `coroutine.resume` extra argumentumait. A korutin az indítás után addig fut, amíg be nem fejeződik, vagy *szüneteltetve* nem lesz.

Egy korutin kétféle módon fejeződhet be: normál módon, amikor a fő függvény visszatér (explicit vagy implicit módon, az utolsó utasítás után); és abnormálisan, egy lekezeletlen hiba esetén. Az első esetben a `coroutine.resume` visszatérési értéke **true**, plusz a korutin fő függvénye által visszatérő értékek. Hiba esetén a visszatérési érték **false** plusz egy hibaüzenet.

Egy korutin a `coroutine.yield` hívással szüneteltethető. Ekkor a megfelelő `coroutine.resume` azonnal visszatér, még akkor is, ha a szüneteltetés egy beágyazott függvényhívásból származik (tehát nem a fő függvényben, hanem a főfüggvényből közvetve vagy közvetlenül hívott másik függvényből). Szüneteltetéskor a `coroutine.resume` szintén **true** értékkel tér vissza, plusz a `coroutine.yield` argumentumaival. A korutin következő folytatásakor a végrehajtás a szüneteltetés helyétől folytatódik, a `coroutine.yield` megkapja a `coroutine.resume` extra paramétereit.

A `coroutine.wrap` függvény is egy korutint hoz létre, hasonlóan, mint a `coroutine.create`, de a visszatérési értéke nem a korutin maga, hanem egy olyan függvény, amely minden hívásakor folytatja a korutint. Minden egyes argumentumot,

ami ehhez a függvényhez van társítva, megkap a `coroutine.resume` hívás. A `coroutine.wrap` visszatérési értéke megegyezik a `coroutine.resume` visszatérési értékével, kivéve az elsőt (a boolean típusú hibakódot). A `coroutine.resume` függvénytől eltérően a `coroutine.wrap` nem kezeli le a hibákat; minden hiba a hívónak lesz továbbítva.

Következzen egy példa:

```
function foo (a)
    print("foo", a)
    return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
    print("co-body", a, b)
    local r = foo(a+1)
    print("co-body", r)
    local r, s = coroutine.yield(a+b, a-b)
    print("co-body", r, s)
    return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

Futtatáskor ez a kód a következő kimenetet fogja eredményezni:

```
co-body 1 10
foo 2
main true 4
co-body r
main true 11 -9
co-body x y
main true 10 end
main false cannot resume dead coroutine
```

3 - Az Alkalmazás Programozási Interfész

Ez a rész a Lua C API-ját tárgyalja, azaz a host program számára elérhető C függvényeket, amelyek segítségével a host kommunikálhat a Lua-val. Minden API függvény és a kapcsolódó típusok és konstansok a `lua.h` fejlécfájlban vannak deklarálva.

Sok esetben akkor is a "függvény" kifejezést használjuk, ha néha az az API-ban makroutasításként érhető el. Minden ilyen makró minden egyes argumentumát csak egyszer használja fel (kivéve az elsőt, amely mindig egy Lua állapot), és így nem hajt végre semmilyen rejtett mellékműveletet.

A legtöbb C eljáráskönyvtárhoz hasonlóan a LUA függvények sem ellenőrzik az argumentumaik érvényességét vagy állapotát. Viszont ez a tulajdonság

megváltoztatható, ha a Lua-ban a `luaconf.h` fájlban található `luai_apicheck` makrót tökéletes definícióval látjuk el.

3.1 - A verem

A Lua az értékek C-beli átadásához és átvételéhez *virtuális vermet* használ. Ebben a veremben minden érték egy Lua értéket képvisel (**nil**, szám, karakterlánc, stb).

Amikor a Lua C hívást hajt végre, a hívott függvény egy új vermet kap, amely különbözik mind az előző vermektől, mind az aktív C függvényekétől. Ez a verem alapértelmezés szerint a C függvény argumentumait tartalmazza, valamint ebbe kerülnek azok az eredmények, amelyek visszatérési értékek lesznek (lásd `lua_CFunction`).

Kényelmi okokból a API-ban a legtöbb lekérdező művelet nem követi a pontos veremszabályokat. Ehelyett *indexek* használatával bármely elem elérhető: a pozitív index egy *abszolút* verempozíciót jelöl (1-től indulva), míg a negatív index a verem tetejétől való *távolságot* jelöli. Pontosabban, egy *n* elemű verem esetén az 1-es index jelenti az első elemet (azaz azt, amelyik elsőként került a verembe), az *n* index pedig az utolsó elemet; a -1 is az utolsó elemet (tehát azt, amelyik a verem tetején van), a -*n* index pedig az elsőt. Az index akkor tekinthető *érvényesnek*, ha az értéke 1 és a verem teteje között fekszik (tehát ha igaz rá, hogy `1 ≤ abs(index) ≤ top`).

3.2 - Verem méret

A Lua API-val történő kapcsolat közben a programozó felelőssége az egyenletesség biztosítása. Azaz, *a programozó feladata a veremtúlcsordulás lekezelése*. A `lua_checkstack` függvény használatával növelhető a verem mérete.

Amikor a Lua meghívja a C-t, megbizonyosodik arról, hogy elérhető -e legalább `LUA_MINSTACK` verempozíció. A `LUA_MINSTACK` értéke 20, így a legtöbb esetben nem kell aggódni a verem helyek miatt, ha csak nem a kód ciklusok használatával helyez el elemeket a veremben.

A legtöbb lekérdező függvény bármilyen index értékeket elfogad az elérhető veremméreten belül, tehát akkora értékig, amekkora a `lua_checkstack` értéknek be lett állítva. Ezeknek az indexeknek a neve az *elfogadható indexek*. Szabályszerűbben, az *elfogadható indexet* az alábbi formában definiálhatjuk:

```
(index < 0 && abs(index) <= top) ||
(index > 0 && index <= stackspace)
```

A 0 soha nem minősül elfogadható indexnek.

3.3 - Pszeudo-indexek

Ha nincs másképp feltüntetve, bármely függvény, amely érvényes indexeket elfogad, egyben pszeudo-indexeknek is nevezzük, ami olyan Lua értéket képvisel, ami elérhető a C kód számára, de nincs a veremben. A pszeudo-indexek a szál

környezetek, függvény környezetek, a registry és a C függvények felsőértékei számára vannak fenntartva (lásd §3.4).

A szál környezete (ahol a globális változók vannak) mindig a `LUA_GLOBALSINDEX`, a futó C függvény környezete pedig mindig a `LUA_ENVIRONINDEX` pszeudoindexnél található.

A globális változók eléréséhez és megváltoztatásához szabványos tömbműveletek használatosak a környezeti tömbön. Például, egy globális változó értékének elérésére a következő kód használható:

```
lua_getfield(L, LUA_GLOBALSINDEX, varname);
```

3.4 - C zárványok

Amikor egy C függvény létrejön, néhány érték társítható hozzá, ami így egy *C zárványt* eredményez; ezek az értékek a *felsőértékek*, és a függvények számára bármikor elérhetőek, amikor meghívódnak (lásd `lua_pushcclosure`).

Amikor egy C függvény meghívódik, a felsőértékek egy meghatározott pszeudo-indexen helyezkednek el. Ezeket a pszeudoindexeket a `lua_upvalueindex` makró hozza létre. Az első függvényhez társított érték a `lua_upvalueindex(1)` pozícióban található, és így tovább. Bármely `lua_upvalueindex(`*n*`)` elérés esetén, ahol *n* nagyobb, mint a pillanatnyi függvény felsőértékeinek a száma, egy elfogadható (de nem érvényes) indexet eredményez.

3.5 - Registry

A Lua biztosít egy *registryt* is, amely egy előre definiált tömb, ahol a C kód bármilyen szükséges Lua értéket tárolhat. Ez a tömb mindig a `LUA_REGISTRYINDEX` pszeudoindex pozícióban található. Bármilyen C eljáráskönyvtár tárolhat adatot ebben a tömbben, csak arra kell ügyelni, hogy ne legyen névütközés, azaz hogy más kulcsokat használjon, mint a többi eljáráskönyvtár. Kulcsként ajánlatos az eljáráskönyvtár nevét használni karakterlánc formában, vagy egy könnyű userdata típust, amely a kódban a C objektum címzése.

Az egész típusú kulcsokat a registryben a hivatkozási mechanizmus használja, amelyet a kisegítő eljáráskönyvtár hoz létre, így más célra nem használható.

3.6 - Hibakezelés C-ben

Belsőleg a Lua a C `longjmp` -ot használja a hibakezelésre. (Kivételek is használhatóak a C++ kódban; lásd a `luaconf.h` fájlt.) Amikor a Lua valamilyen hibába ütközik (mint például memória-lefoglalási hibák, típushibák, szintaktikai hibák és futási hibák), egy hibához *ér*, azaz egy hosszú ugrást hajt végre. A *védett környezet* a `setjmp` -ot használja visszaállítási pontok létrehozásához, bármilyen hiba a legutolsó aktív visszaállítási ponthoz ugrik.

Majdnem az összes API függvény okozhat hibát, például egy memória-lefoglalási hibán keresztül. A következő függvények védett módban futnak (azaz futásukkor egy védett környezetet hoznak létre), így ezek soha nem okoznak hibát: `lua_newstate`, `lua_close`, `lua_load`, `lua_pcall`, és `lua_cpcall`.

Egy C függvényen szándékosan is lehet hibát okozni a `lua_error` meghívásával.

3.7 - Függvények és típusok

A következőkben szerepelnek a C API függvényei és típusai, ABC sorrendben.

---

lua_Alloc

```
typedef void * (*lua_Alloc) (void *ud,
 void *ptr,
 size_t osize,
 size_t nsize);
```

A memória-lefoglalási függvény típusa, amit a Lua állapotok használnak. A lefoglaló függvénynek hasonlóan kell működnie, mint a `realloc`, de nem pont ugyanúgy. Az argumentumai az `ud`, amely a `lua_newstate` nem átlátszó mutatója; `ptr`, egy mutató, amely a lefoglalandó/újrafoglalandó/felszabadítandó memóriablokkra mutat, `osize`, a blokk eredeti mérete; `nsize`, a blokk új mérete. A `ptr` értéke akkor, és csakis akkor `NULL` ha az `osize` értéke zéró. Amikor az `nsize` értéke zéró, a lefoglaló visszatérési értéke `NULL`; ha `osize` nem zéró, akkor fel kell szabadítania a `ptr` által mutatott blokkot. Amikor `nsize` nem zéró, a lefoglaló akkor, és csakis akkor tér vissza `NULL` értékkel, ha nem tudta végrehajtani a kérést. Ha `nsize` nem zéró és `osize` zéró, a lefoglalónak úgy kell működnie, mint a `malloc`-nak. Amikor sem az `nsize` sem az `osize` nem zéró, a lefoglaló úgy működik, mint a `realloc`. A Lua feltételezi, hogy a lefoglaló soha nem hibázik, ha `osize >= nsize`.

A következőkben a lefoglaló függvény egyszerű alkalmazása szerepel. Ezt a kiegészítő eljáráskönyvtárban a `lua_newstate` használja.

```
static void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
(void)ud; /* not used */
(void)osize; /* not used */
    if (nsize == 0) {
        free(ptr); /* ANSI requires that free(NULL) has no effect */
        return NULL;
    } else
    /* ANSI requires that realloc(NULL, size) == malloc(size) */
    return realloc(ptr, nsize);
}
```

---

lua_atpanic

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

Beállít egy új pánik függvényt, és visszatér a régivel.

Ha a védett környezeten kívül hiba keletkezik, a Lua először a *pánik függvényt*, majd az `exit(EXIT_FAILURE)` függvényt hívja meg, így kilép a host alkalmazásba. A pánik függvény használatával megelőzhető ez a kilépés, ha soha nem tér vissza (pl. csinál egy hosszú ugrást).

A pánik függvény elérheti a hibaüzenetet is, ami a verem tetején található.

---

lua_call

```
void lua_call (lua_State *L, int nargs, int nresults);
```

Meghív egy függvényt.

Egy függvény meghívásához a következő szabályokat kell betartani: először, a hívandó függvényt a verembe kell helyezni; utána az argumentumokat közvetlen sorrendben, azaz az elsőt először. Végül meghívható a függvény a `lua_call` segítségével; `nargs` a verembe tett argumentumok száma. Minden argumentum és a függvény értéke a függvény hívásakor a veremből lesz kiemelve. A függvény eredmények a függvény visszatérésekor a veremben lesznek elhelyezve. Az eredmények számát a `nresults` szabja meg; ennek hiányában a `nresults` értéke `LUA_MULTRET`. Ebben az esetben a függvény *minden* eredménye verembe lesz téve. A Lua gondoskodik arról, hogy a visszatért értékek elférjenek a veremben. A függvény eredményei közvetlen sorrendben lesznek elhelyezve a veremben (azaz a legelső eredmény kerül bele először),így a hívás után a verem tetején a legutolsó eredmény lesz.

A hívott függvény minden hibája felfelé lesz továbbítva (egy `longjmp` használatával).

A következő minta megmutatja, hogyan hívható meg a Lua kód a host programból:

```
 a = f("how", t.x, 14)
```

Ugyanez C-ben:

```
 lua_getfield(L, LUA_GLOBALSINDEX, "f"); /* hívandó függvény */
 lua_pushstring(L, "how"); /* első argumentum */
 lua_getfield(L, LUA_GLOBALSINDEX, "t"); /* Az indexelendő tömb */
 lua_getfield(L, -1, "x"); /* t.x eredményének elhelyezése (második arg) */
 lua_remove(L, -2); /* 't' eltávolítása a veremből */
 lua_pushinteger(L, 14); /* harmadik argumentum */
 lua_call(L, 3, 1); /* függvény hívása három argumentummal és egy
eredménnyel */
 lua_setfield(L, LUA_GLOBALSINDEX, "a"); /* az 'a' globális változó
beállítása */
```

A fenti kód "kiegyensúlyozott": a végére a verem az eredeti állapotába tér vissza. Ez egy fontos tanács programozóknak.

---

lua_CFunction

```
typedef int (*lua_CFunction) (lua_State *L);
```

A C függvények típusa.

A Lua-val történő tökéletes kommunikáció érdekében a C függvénynek a következő szabályokat kell követnie (ezek szabályozzák a paraméterek és az eredmények átadását): A C függvény megkapja az argumentumokat a Lua-tól annak vermében, követlen sorrendben (az első arumentum szerepel az első helyen). Így a függvény indulásakor a `lua_gettop(L)` a függvény argumentumainak számával tér vissza. Az első argumentum (ha van), az 1-es indexen, míg az utolsó a `lua_gettop(L)` indexen helyezkedik el. Az eredmények visszatéréséhez a Lua-ba a C függvény a verembe helyezi azokat közvetlen sorrendben (tehát az elsőt helyezi bele először), és az eredmények számával tér vissza. A veremben található további értékeket a Lua figyelmen kívül hagyja. Mint egy Lua függvény, a Lua által hívott C függvénynek is számos visszatérési értéke lehet.

A következő példafüggvény a változó számú szám argumentumok átlagával és összegével tér vissza:

```
 static int foo (lua_State *L) {
     int n = lua_gettop(L); /* argumentumok száma */
     lua_Number sum = 0;
     int i;
     for (i = 1; i <= n; i++) {
         if (!lua_isnumber(L, i)) {
             lua_pushstring(L, "incorrect argument to function 'average'");
             lua_error(L);
         }
         sum += lua_tonumber(L, i);
     }
     lua_pushnumber(L, sum/n); /* első eredmény */
     lua_pushnumber(L, sum); /* második eredmény */
     return 2; /* eredmények száma */
 }
```

---

lua_checkstack

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

Megbizonyosodik arról, hogy van legalább `extra` szabad veremhely a veremben. false értékkel tér vissza, ha nem tudta megnövelni a verem méretét a szükségesre. Ez a függvény soha nem kicsinyíti a verem méretét; ha a verem mérete nagyobb, mint az új méret, úgy hagyja.

---

lua_close

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

A megadott Lua állapot összes objektumát megsemmisíti (a megfelelő szemétgyűjtő metaeljárás meghívásával, ha van) és az állapot által használt összes dinamikus memóriát felszabadítja. Néhány platformon nem szükséges meghívni ezt a függvényt, mivel minden erőforrás fel lesz szabadítva, mikor a host program futása véget ér. Viszont a hosszú futású programok esetén, mint például egy webszerver, amint nincs rá szükség, azonnal fel kell szabadítani ezeket, hogy ne nőhessenek túl nagyra.

---

lua_concat

```
void lua_concat (lua_State *L, int n);
```

Összefűzi a verem tetején lévő `n` értéket, kiveszi őket, majd az eredményt a verem tetején helyezi el. Ha `n` értéke 1, az eredmény a veremben lévő egyetlen karakterlánc (tehát nem tesz semmit); ha `n` 0, az eredmény egy üres karakterlánc. Az összefűzés a Lua szabályait követi (lásd §2.5.4).

---

lua_cpcall

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

A `func` C függvényt védett módban futtatja. A `func` vermében csak egyetlen elem van, egy könnyű userdata típusú, melynek tartalma `ud`. Hiba esetén a `lua_cpcall` ugyanazzal a hibakóddal tér vissza, mint a `lua_pcall`, plusz a hibaobjektummal a verem tetején, egyébként a visszatérési értéke zéró, és nem változtatja meg a verem tartalmát. A `func` minden visszatérési értéke figyelmen kívül lesz hagyva.

---

lua_createtable

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

Egy üres tömböt készít, majd a verem tetejére helyezi. Az új tömbnek `narr` számú előre lefoglalt helye van tömb elemek és `nrec` helye nem-tömb elemek számára. Az elő-lefoglalás akkor lehet hasznos, ha pontosan tudható, mennyi eleme lesz a tömbnek. Egyéb esetben a `lua_newtable` függvény használandó.

---

lua_dump

```
int lua_dump (lua_State *L, lua_Writer writer, void *data);
```

A megadott függvényt bináris csonkká alakítja át. Egy Lua függvényt kap a verem tetején, és ennek a bináris reprezentációját készíti el, ha ez újra be lesz töltve, ugyanazt a függvényt adja vissza, mint ami az átalakítás előtt volt. A csonk részeinek

létrehozása közben a lua_dump meghívja a writer függvényt is (lásd lua_Writer) az adott data paraméterrel, írásra.

A visszatérési érték egy hibakód, amely az író által visszaadott utolsó érték; 0 esetén nem történt hiba.

Ez a függvény nem emeli ki a Lua függvényt a veremből.

---

lua_equal

```
int lua_equal (lua_State *L, int index1, int index2);
```

Visszatérési értéke 1, ha az index1 és index2 elfogadható indexeknél található értékek megegyeznek, a Lua == operátorának szemantikáját követve (így meghívhat metaeljárásokat is). Egyéb esetben a visszatérési értéke 0. Szintén 0 a visszatérési értéke, ha valamelyik index nem érvényes.

---

lua_error

```
int lua_error (lua_State *L);
```

Egy Lua hibát generál. A hibaüzenetnek (amely bármilyen Lua típus érték lehet) a verem tetején kell elhelyezkednie. A függvény hosszú ugrást hajt végre, így soha nem rendelkezik visszatérési értékkel. (lásd luaL_error).

---

lua_gc

```
int lua_gc (lua_State *L, int what, int data);
```

A szemétgyűjtőt szabályozza.

Ez a függvény többféle folyamatot is végrehajthat, a what paraméter értékétől függően:

- **LUA_GCSTOP:** megállítja a szemétgyűjtőt.
- **LUA_GCRESTART:** újraindítja a szemétgyűjtőt.
- **LUA_GCCOLLECT:** elindít egy teljes szemétgyűjtő kört.
- **LUA_GCCOUNT:** a Lua által használt memória pillanatnyi értékét adja vissza (Kbyte-okban).
- **LUA_GCCOUNTB:** visszatérési értéke a Lua által felhasznált memória értéke (byte-okban) és 1024 osztásából származó maradék.
- **LUA_GCSTEP:** végrehajt egy növekményes szemétgyűjtő-lépést. A lépés "méretét" a data határozza meg (nagyobb érték mellett több lépés) meghatározatlan módon. Ha a lépés méretét szabályozni akarod, a data

értékével kell kísérletezni. Visszatérési értéke 1, ha a lépéssel befejeződött egy szemétgyűjtő-kör.
- **LUA_GCSETPAUSE:** A gyűjtő *szünetét* állítja be `data`/100 értékre (lásd §2.10). Visszatérési értéke a szünet előző értéke.
- **LUA_GCSETSTEPMUL:** A gyűjtő *lépésszorzóját* állítja be `arg`/100 értékre (lásd §2.10). Visszatérési értéke a lépésszorzó előző értéke.

---

lua_getallocf

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

Visszatérési értéke a megadott állapot memória-lefoglaló függvénye. Ha az `ud` nem `NULL`, a Lua a `lua_newstate` számára átadott nem átlátszó mutatót az `*ud` -ben tárolja.

---

lua_getfenv

```
void lua_getfenv (lua_State *L, int index);
```

A verembe helyezi a megadott indexhez tartozó érték környezeti tömbjét.

---

lua_getfield

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

A verembe helyezi a `t[k]` értékét, ahol `t` az érvényes `index` index értéke. Mint a Lua nyelvben, ez a függvény is kiválthatja az "index" metaeljárást (lásd §2.8).

---

lua_getglobal

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

A verembe helyezi a globális `name` változó értékét. Ez makróként definiált:

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

---

lua_getmetatable

```
int lua_getmetatable (lua_State *L, int index);
```

A verembe helyezi a megadott elfogadható indexen található érték metatömbjét. Ha az index nem érvényes, vagy az értéknek nincs metatömbje, akkor a visszatérési érték 0, és semmi nem kerül a verembe.

---

lua_gettable

```
void lua_gettable (lua_State *L, int index);
```

A verembe helyezi a `t[k]` értékét, ahol `t` az érvényes `index` index értéke és `k` az értéke a verem tetején.

Ez a függvény kiemeli a veremből az adott kulcsot (és az eredményét rakja be a helyére). Mint a Lua nyelvben, ez a függvény is kiválthatja az "index" metaeljárást (lásd §2.8).

---

lua_gettop

```
int lua_gettop (lua_State *L);
```

Visszatérési értéke a verem legfelső elemének indexe. Mivel az indexek 1-től indulnak, ez a szám megegyezik a veremben lévő elemek számával (és így a 0 azt jelenti, hogy a verem üres).

---

lua_insert

```
void lua_insert (lua_State *L, int index);
```

A legfelső elemet a megadott érvényes indexre mozgatja, és a fentebbi elemeket felfelé, szabad helyre csúsztatja. Nem hívható pszeudo-indexszel, mivel egy pszeudo-index nem az aktuális verempozíciót adja vissza.

---

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

Ezt a típust a Lua API használja egész típusú értékek létrehozásához.

Alapértelmezés szerint ez egy `ptrdiff_t`, amely rendszerint a legnagyobb integrált típus, amelyet a gép "kényelmesen" kezelni tud.

---

lua_isboolean

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

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték típusa boolean, egyéb esetben 0.

---

lua_iscfunction

```
int lua_iscfunction (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték egy C függvény, egyéb esetben 0.

---

lua_isfunction

```
int lua_isfunction (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték függvény (akár C, akár Lua), egyéb esetben 0.

---

lua_islightuserdata

```
int lua_islightuserdata (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték könnyű userdata, egyéb esetben 0.

---

lua_isnil

```
int lua_isnil (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték **nil**, egyéb esetben 0.

---

lua_isnumber

```
int lua_isnumber (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték szám, vagy számmá alakítható karakterlánc, egyéb esetben 0.

---

lua_isstring

```
int lua_isstring (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték karakterlánc vagy szám (amely mindig karakterlánccá alakítható), egyéb esetben 0.

---

lua_istable

```
int lua_istable (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték tömb, egyéb esetben 0.

---

lua_isthread

```
int lua_isthread (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték szál, egyéb esetben 0.

---

lua_isuserdata

```
int lua_isuserdata (lua_State *L, int index);
```

Visszatérési értéke 1, ha a megadott elfogadható indexen található érték userdata (akár teljes, akár könnyű), egyéb esetben 0.

---

lua_lessthan

```
int lua_lessthan (lua_State *L, int index1, int index2);
```

Visszatérési értéke 1, ha a megadott elfogadható `index1` indexen található érték kisebb, mint az `index2` elfogadható indexen található érték, a Lua `<` operátor szemantikájának megfelelően (így metaeljárások is meghívódhatnak). Egyéb esetben a visszatérési érték 0. Szintén 0 a visszatérési érték, ha valamelyik index nem érvényes.

---

lua_load

```
int lua_load (lua_State *L,
 lua_Reader reader,
 void *data,
 const char *chunkname);
```

Betölt egy Lua csonkot. Ha eközben nem történik hiba, a `lua_load` a Lua függvényként lefordított csonkot helyezi a verem tetejére. Egyéb esetben egy hibaüzenetet helyez oda. A `lua_load` visszatérési értékei:

- **0:** nincs hiba;
- **`LUA_ERRSYNTAX`:** szintaktikai hiba az előfordítás közben;
- **`LUA_ERRMEM`:** memória lefoglalási hiba.

A függvény csak betölti a csonkot, de nem futtatja le azt.

A `lua_load` automatikusan érzékeli, hogy a csonk szöveges vagy bináris, és ennek megfelelően tölti be azt (lásd a `luac` programot).

A `lua_load` egy felhasználó által betöltött `reader` függvényt használ a csonk olvasására (lásd `lua_Reader`). A `data` argumentum az olvasó függvénynek átadott, nem átlátszó érték.

A `chunkname` argumentum nevet ad a csonknak, amely a hibaüzenetekben, valamint hibakereséskor van használatban (lásd §3.8).

`lua_newstate`

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

Létrehoz egy új, független állapotot. Visszatérési értéke `NULL`, ha az állapot nem létrehozható (memóriahiány miatt). Az `f` argumentum a lefoglaló függvény; a Lua a teljes memórialefoglalást ezen a függvényen keresztül végzi. A második, `ud`, egy nem átlátszó mutató, amelyet a Lua a lefoglaló függvénynek ad át minden híváskor.

`lua_newtable`

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

Egy új, üres tömböt hoz létre, és a verem tetejére helyezi azt. Megegyezik a `lua_createtable(L, 0, 0)` hívással.

`lua_newthread`

```
lua_State *lua_newthread (lua_State *L);
```

Létrehoz egy új szálat, a verembe helyezi, és annak a `lua_State` értéknek a mutatójával tér vissza, amelyik ezt a szálat adja vissza. Az ebből a függvényből visszatért új állapot osztozik az eredeti állapot globális objektumaival (például tömbjeivel), de rendelkezik egy független végrehajtási veremmel is.

Egy szál lezárására vagy megsemmisítésére nincs kifejezett függvény. A szálakat a szemétgyűjtő algoritmus semmisíti meg, mint minden Lua objektumot.

---

lua_newuserdata

```
void *lua_newuserdata (lua_State *L, size_t size);
```

A függvény felszabadítja a megadott méretű memóriaterületet, és a verembe helyezi az új teljes userdata elemet a blokk címével, és ezzel a címmel tér vissza.

A userdata C értékeket hoz létre a Lua-ban. Egy *teljes userdata* egy memóriablokkot reprezentál. Ez egy objektum (mint egy tömb): létre kell hozni, lehet saját metatömbje, és művelet hajtható végre, amikor megsemmisítésre kerül. Egy teljes userdata csak önmagával egyenlő (raw egyenlőség esetén).

Amikor a Lua összegyűjt egy olyan teljes userdata elemet, amelynek van `gc` metaeljárása, a Lua meghívja azt, majd a userdata elemet véglegesítettnek jelöli. Amikor ez a userdata elem újra összegyűjtésre kerül, a Lua felszabadítja a megfelelő memóriát.

---

lua_next

```
int lua_next (lua_State *L, int index);
```

Kiemel egy kulcsot a veremből, majd a megadott indexen található kulcs-érték párt helyezi bele (a megadott kulcs utáni "következő" párt). Ha nincs több elem a tömbben, a `lua_next` visszatérési értéke 0 (és nem helyez a verembe semmit).

Egy tipikus bejárás így néz ki:

```
 /* A tömb a veremben a 't' indexnél helyezkedik el */
 lua_pushnil(L); /* első kulcs */
 while (lua_next(L, t) != 0) {
     /* 'kulcs' a -2-es indexnél van, az 'érték' -1 indexnél */
     printf("%s - %s\n",
     lua_typename(L, lua_type(L, -2)), lua_typename(L, lua_type(L, -1)));
     lua_pop(L, 1); /* eltávolítja az 'értéket'; megtartja a 'kulcsot' a
következő iterációhoz */
 }
```

Egy tömb bejárása közben ne használjuk közvetlenül a `lua_tolstring` hívást egy kulcson, hacsak nem biztos, hogy az aktuális kulcs karakterlánc. A `lua_tolstring` újrahívása *megváltoztatja* a megadott indexen található értéket, ami összezavarja a következő `lua_next` hívást.

---

lua_Number

```
typedef double lua_Number;
```

A Lua által használt számok típusa. Alapértelmezés szerint ez dupla (double), de megváltoztatható a `luaconf.h` fájlban.

A konfigurációs fájl segítségével a Lua más szám-típusokkal is végezhet műveleteket (pl. lebegőpontos (float) vagy hosszú (long)).

---

lua_objlen

```
size_t lua_objlen (lua_State *L, int index);
```

Visszatérési értéke a megadott elfogadható indexen található érték "hossza": karakterlánc esetén annak hossza, tömbök esetén a hossz operátor eredménye ('`#`'); userdata típus esetén a számára lefoglalt memóriablokk mérete, egyéb értékek esetén 0.

---

lua_pcall

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

Védett módban hív meg egy függvényt.

Mind a `nargs`, mind a `nresults` jelentése ugyanaz, mint a `lua_call` esetén. Ha nem történik hiba a hívás közben, `lua_pcall` pontosan ugyanúgy viselkedik, mint a `lua_call`. Azonban ha hiba történik, a `lua_pcall` lekezeli azt, egyetlen értéket helyez a verem tetejére (a hibaüzenetet), és a hibakóddal tér vissza. A `lua_call`-hoz hasonlóan, a `lua_pcall` is mindig eltávolítja a veremből a függvényt, valamint annak argumentumait.

Ha az `errfunc` értéke 0, akkor a veremben visszatérő hibaüzenet megegyezik az eredeti hibaüzenettel. Egyéb esetben az `errfunc` egy *hiba kezelő függvény* veremindexe. (A jelenlegi implementációban ez az index nem lehet pszeudo-index.) Futtatási hiba esetén ez a függvény a hibaüzenettel lesz meghívva, és a visszatérési értéke a `lua_pcall` által a verembe helyezett hibaüzenet lesz.

A hibakezelő függvény leginkább arra szolgál, hogy további hibakeresési információkat adhassunk a hibaüzenethez, mint például a verem visszavezetés (stack traceback). Ezek az információk a `lua_pcall` visszatérése után már nem gyűjthetőek össze, mivel akkorra a verem már kiürül.

A `lua_pcall` függvény visszatérési értéke 0 siker esetén, egyéb esetben a következő hibakódok szerepelhetnek (a `lua.h` fájlban vannak definiálva):

- **LUA_ERRRUN:** futtatási hiba (runtime error).

- **LUA_ERRMEM:** memória lefoglalási hiba. Ilyen hiba esetén a Lua nem hívja meg a hibakezelő függvényt.
- **LUA_ERRERR:** Hiba a hibakezelő függvény futtatása közben.

---

lua_pop

```
void lua_pop (lua_State *L, int n);
```

Kiemel `n` elemet a veremből.

---

lua_pushboolean

```
void lua_pushboolean (lua_State *L, int b);
```

A verembe helyez egy boolean értéket `b` értékkel.

---

lua_pushcclosure

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

Egy új C zárványt helyez a verembe.

Amikor egy C függvény létrejön, társítható hozzá néhány érték, ami így egy C zárványt hoz létre (lásd §3.4); ezután ezek az értékek bármikor elérhetőek lesznek, amikor a függvény meg lesz hívva. Értékek C függvényhez társításához először ezeket az értékeket a verembe kell helyezni (több érték esetén az első érték legyen először elhelyezve). Ezután a `lua_pushcclosure` lesz meghívva, hogy hozza létre a C függvényt és helyezze a verembe. Az `n` argumentum adja meg, hogy mennyi érték van a függvényhez társítva. A `lua_pushcclosure` kiemeli az értékeket a veremből.

---

lua_pushcfunction

```
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

A verembe helyez egy C függvényt. A függvény egy C függvényre mutató mutatót kap, és a verembe helyezi a `function` Lua típusát, így amikor meg lesz hívva, ehhez a megfelelő C függvényt hívja segítségül.

Bármilyen, Lua-ban regisztrálandó függvénynek pontosan követnie kell a protokollt, hogy megkaphassa a paramétereit, és visszatérhessen az eredményekkel (lásd `lua_CFunction`).

A `lua_pushcfunction(L, f)` megegyezik a következővel: `lua_pushcclosure(L, f, 0)`.

---

lua_pushfstring

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

A verembe helyez egy formázott karakterláncot, és a karakterláncra mutató pointerrel tér vissza. Hasonlít a `sprintf` C függvényhez, de van néhány fontos különbség:

- Nem kell helyet felszabadítani az eredménynek: a visszatérési érték egy Lua karakterlánc, és a Lua gondoskodik a memória-lefoglalásról (és felszabadításról, a szemétgyűjtésen keresztül).
- Az átalakítási vezérlők korlátozottabbak. Nincsenek jelzők (flagek), szélességek (widths) és pontosságok. Az átalakítási vezérlők a következőek lehetnek: '`%%`' (Egy '`%`' jelet helyez a karakterláncba), '`%s`' (egy zéró-végű karakterláncot helyez el, méretbeli korlátozások nélkül), '`%f`' (egy `lua_Number`-t helyez el), '`%p`' (egy mutatót helyez el hexadecimális számként), '`%d`' (egy `int` egész számot helyez el), és '`%c`' (egy `int` egész számot helyez el karakterként).

---

lua_pushinteger

```
void lua_pushinteger (lua_State *L, lua_Integer n);
```

Egy `n` értékű egész számot helyez a verembe.

---

lua_pushlightuserdata

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

Egy könnyű userdata-t helyet a verembe.

A userdata C értékeket jelent Lua-ban. Egy *könnyű userdata* egy mutatót képvisel. Ez egy érték (mint egy szám): nem kell létrehozni, nincs egyedi metatömbje és nem lesz összegyűjtve (mivel soha nem is volt létrehozva). Egy könnyű userdata megegyezik "bármelyik" könnyű userdata-val, amelynek ugyanaz a C címzése.

---

lua_pushlstring

```
void lua_pushlstring (lua_State *L, const char *s, size_t len);
```

A verembe helyezi az `s` által mutatott `len` méretű karakterláncot. A Lua egy belső másolatot készít (vagy újra felhasznál) az adott karakterláncról, így az `s` memóriaterülete közvetlenül a függvény visszatérése után felszabadítható vagy újra felhasználható. A karakterlánc tartalmazhat beágyazott zérókat.

---

lua_pushnil

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

Egy nil értéket helyez el a verem tetején.

---

lua_pushnumber

```
void lua_pushnumber (lua_State *L, lua_Number n);
```

Egy `n` értékű számot helyez a verembe.

---

lua_pushstring

```
void lua_pushstring (lua_State *L, const char *s);
```

A verembe helyezi az `s` által mutatott `len` méretű, zéró végű karakterláncot. A Lua egy belső másolatot készít (vagy újra felhasznál) az adott karakterláncról, így az `s` memóriaterülete közvetlenül a függvény visszatérése után felszabadítható vagy újra felhasználható. A karakterlánc nem tartalmazhat beágyazott zérókat; az első ilyen a karakterlánc végét fogja jelenteni.

---

lua_pushthread

```
int lua_pushthread (lua_State *L);
```

A verembe helyezi az `L` által képviselt szálat. Visszatérési értéke 1, ha ez a szál az adott állapot főszála.

---

lua_pushvalue

```
void lua_pushvalue (lua_State *L, int index);
```

A verembe helyezi a megadott érvényes indexen található érték másolatát.

---

lua_pushvfstring

```
const char *lua_pushvfstring (lua_State *L,
 const char *fmt,
 va_list argp);
```

Megegyezik a lua_pushfstring hívással, kivéve, hogy ez egy va_list paramétert kap az argumentumok száma helyett.

---

lua_rawequal

```
int lua_rawequal (lua_State *L, int index1, int index2);
```

Visszatérési értéke 1, ha az index1 és index2 elfogadható indexen található értékek primitíven megegyeznek (azaz bármilyen metaeljárás meghívása nélkül). Egyéb esetben a visszatérési értéke 0. Szintén 0-val tér vissza, ha a megadott indexek valamelyike nem érvényes.

---

lua_rawget

```
void lua_rawget (lua_State *L, int index);
```

A lua_gettable híváshoz hasonlóan működik, de raw elérést hajt végre (tehát metaeljárások nélkül).

---

lua_rawgeti

```
void lua_rawgeti (lua_State *L, int index, int n);
```

A verembe helyezi a t[n] értékét, ahol t az érvényes index index értéke. Az elérés raw formátumú, így nem hajt végre metaeljárásokat.

---

lua_rawset

```
void lua_rawset (lua_State *L, int index);
```

A lua_settable híváshoz hasonlóan működik, de raw értékadást hajt végre (tehát metaeljárások nélkül).

---

lua_rawseti

```
void lua_rawseti (lua_State *L, int index, int n);
```

Ugyanazt hajtja végre, mint a `t[n] = v`, ahol `t` az érvényes `index` index értéke, és `v` a verem tetején lévő érték.

Ez a függvény kiemeli a verem tetején lévő értéket. Az értékadás raw formátumú, így nem hajt végre metaeljárásokat.

---

lua_Reader

```
typedef const char * (*lua_Reader) (lua_State *L,
 void *data,
 size_t *size);
```

Az olvasó függvényt a `lua_load` használja. Minden alkalommal, amikor szüksége van egy csonk újabb részére, a `lua_load` meghívja az olvasót, végigmenve a `data` paraméteren. Az olvasónak a csonk új részletét tartalmazó memóriablokk mutatójával kell visszatérnie, és a `size` értékét a blokk méretére kell állítania. A blokknak addig kell léteznie, amíg az olvasó függvény újra meg lesz hívva. A csonk végét az jelenti, ha az olvasó visszatérési értéke `NULL`. Az olvasó bármekkora méretű darabbal visszatérhet, ami nagyobb, mint nulla.

---

lua_register

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

A megadott `f` C függvényt állítja be a globális `name` új értékeként. Ez egy makróként van definiálva:

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

---

lua_remove

```
void lua_remove (lua_State *L, int index);
```

Eltávolítja a megadott érvényes indexen található értéket, majd az efölött lévő indexeket lefelé csúsztatva betölti a hézagokat. Ne hívható meg pszeudo-indexszel, mivel egy pszeudo-index nem az aktuális verempozíció.

---

lua_replace

```
void lua_replace (lua_State *L, int index);
```

A legfelső elemet a megadott pozícióra mozgatja (és kiemeli azt), a többi elem csúsztatása nélkül (így áthelyezi a megadott pozícióban lévő értéket).

---

lua_resume

```
int lua_resume (lua_State *L, int narg);
```

Elindítja és folytatja a korutint a megadott szálban.

Egy korutin indításához először létre kell hozni egy új szálat (lásd `lua_newthread`); majd a vermébe kell helyezni fő függvényét és annak argumentumait; ezután hívható a `lua_resume`, ahol a `narg` az argumentumok számát adja meg. Ez a hívás akkor tér vissza, ha a korutin szünetel, vagy befejezi a futását. Amikor visszatér, a verem tartalmazza a `lua_yield`-nek átadott összes értéket, vagy az összes, főfüggvény által visszatért értéket. A `lua_resume` visszatérési értéke `LUA_YIELD`, ha a korutin szünetel, 0, ha a korutin hiba nélkül befejezi futását, vagy hiba esetén egy hibakód (lásd `lua_pcall`). Hiba esetén a verem nem ürül ki, így használható rajta a hibakereső API. A hibaüzenet a verem tetején található. A korutin újraindításához annak vermébe kell helyezni a `yield` visszatérési értékeit, majd utána meghívni a `lua_resume`-t.

---

lua_setallocf

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

Megváltoztatja egy megadott állapot lefoglaló függvényét `f`-re, `ud` felhasználói adattal.

---

lua_setfenv

```
int lua_setfenv (lua_State *L, int index);
```

Kiemel egy tömböt a veremből, és a megadott indexen található érték új környezeteként állítja azt be. Ha a megadott indexen lévő érték nem függvény, vagy nem userdata, a `lua_setfenv` visszatérési értéke 0. Egyéb esetben 1.

---

lua_setfield

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

Ugyanazt hajtja végre, mint a `t[k] = v`, ahol `t` az érvényes `index` index értéke, és `v` a verem tetején lévő érték.

Ez a függvény kiemeli a verem tetején lévő értéket. Mint a Lua, ez a függvény is végrehajthatja a "newindex" metaeljárást (lásd §2.8).

---

lua_setglobal

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

Kiemel egy értéket a veremből, és beállítja a globális `name` új értékeként. Ez egy makróként van definiálva:

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

---

lua_setmetatable

```
int lua_setmetatable (lua_State *L, int index);
```

Kiemel egy tömböt a veremből, és a megadott elfogadható indexen található érték új metatömbjeként állítja azt be.

---

lua_settable

```
void lua_settable (lua_State *L, int index);
```

Ugyanazt hajtja végre, mint a `t[k] = v`, ahol `t` az érvényes `index` index értéke, `v` a verem tetején lévő érték és `k` a verem tetejétől számított második érték.

Ez a függvény mind a kulcsot, mind az értéket kiemeli a veremből. Mint a Lua, ez a függvény is végrehajthatja a "newindex" metaeljárást (lásd §2.8).

---

lua_settop

```
void lua_settop (lua_State *L, int index);
```

Bármilyen elfogadható indexet elfogad, vagy 0-t, és a verem tetejét ehhez az elemhez állítja be. Ha az új tetőpont magasabb, mint a régebbi, az újabb elemek **nil** értéket vesznek fel. Ha az `index` értéke 0, akkor az összes elem el lesz távolítva a veremből.
.

---

lua_State

```
typedef struct lua_State lua_State;
```

Egy olyan nem átlátszó struktúra, amely az egész Lua értelmező állapotát tárolja. A Lua eljáráskönyvtár teljesen újrakezdhető: nincsen globális változója. Egy állapot minden információja ebben a struktúrában található.

Az eljáráskönyvtár minden függvényének első paramétere egy olyan mutató, amely erre az állapotra mutat, kivéve a `lua_newstate` esetén, amely a semmiből hozza létre a Lua állapotot.

---

lua_status

```
int lua_status (lua_State *L);
```

Visszatérési értéke az `L` szál állapota.

Az állapot 0 normál szál esetén, hibakód, ha a szál hibával fejezte be a futását, vagy `LUA_YIELD`, ha a szál szüneteltetve van.

---

lua_toboolean

```
int lua_toboolean (lua_State *L, int index);
```

A megadott elfogadható indexen található Lua értéket boolean (0 vagy 1) értékké alakítja át. Mint minden teszt a Lua-ban, a `lua_toboolean` visszatérési értéke 1 bármely Lua érték esetén, amely különbözik **false**-tól és **nil**-től; egyébként 0-val tér vissza. Szintén 0 a visszatérési érték, ha nem érvényes indexszel van meghívva. (Ha aktuálisan csak boolean értékeket akarsz elfogadni, használd a `lua_isboolean` hívást, hogy teszteld az érték típusát.)

---

lua_tocfunction

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

A megadott elfogadható indexen található értéket C függvénnyé alakítja át. Az érték csak C függvény lehet, egyéb esetben a visszatérési érték `NULL`.

---

lua_tointeger

```
lua_Integer lua_tointeger (lua_State *L, int idx);
```

A megadott elfogadható indexen található értéket előjeles integrált `lua_Integer` típussá alakítja át. A Lua értéknek számnak, vagy számmá alakítható karakterláncnak kell lennie (lásd §2.2.1); egyébként a `lua_tointeger` visszatérési értéke 0.

Ha a szám nem egész típusú, akkor nem-meghatározott módon átalakításra kerül.

---

lua_tolstring

```
const char *lua_tolstring (lua_State *L, int index, size_t *len);
```

A megadott elfogadható indexen található Lua értéket karakterlánccá alakítja át (`const char*`). Ha a `len` nem `NULL`, akkor a `*len` értékét a karakterlánc hosszának megfelelő értékre állítja. A Lua értéknek számnak vagy karakterláncnak kell lennie, egyébként a függvény visszatérési értéke `NULL`. Ha az érték szám, a `lua_tolstring` *a jelenlegi értékét karakterlánccá alakítja a veremben*. (Ez a változtatás összezavarhatja a `lua_next`-et, amikor a `lua_tolstring` egy kulcson van végrehajtva, tömb-bejárás alatt.)

A `lua_tolstring` visszatérési értéke egy teljesen sorbarendezett mutató, amely a Lua állapoton belül a karakterláncra mutat. Ez a karakterlánc mindig zéróval végződik ('`\0`') az utolsó karakter után (mint a C-ben), de több beágyazott zérót is tartalmazhat. Mivel a Lua szemétgyűjtéssel is rendelkezik, nincs garancia arra, hogy a `lua_tolstring` hívásból visszatérő mutató érvényes, miután a megfelelő érték el lesz távolítva a veremből.

---

lua_tonumber

```
lua_Number lua_tonumber (lua_State *L, int index);
```

A megadott elfogadható indexen található Lua értéket számmá alakítja át (lásd `lua_Number`). A Lua értéknek számnak, vagy számmá alakítható karakterláncnak kell lennie (lásd §2.2.1); egyéb esetben a `lua_tonumber` visszatérési értéke 0.

---

lua_topointer

```
const void *lua_topointer (lua_State *L, int index);
```

A megadott elfogadható indexen található értéket általános C mutatóvá (`void*`) alakítja át. Az érték lehet userdata, tömb, szál vagy függvény, egyéb esetben a `lua_topointer` visszatérési értéke `NULL`. Különböző objektumok különböző mutatókat eredményeznek. Egy mutató eredeti értékére történő visszaalakítására nincs lehetőség.

Ez a függvény általában hibakereső információk gyűjtésekor van használatban.

---

lua_tostring

```
const char *lua_tostring (lua_State *L, int index);
```

Megyegyezik a `lua_tolstring` függvénnyel, de a `len` értéke `NULL`.

---

lua_tothread

```
lua_State *lua_tothread (lua_State *L, int index);
```

A megadott elfogadható indexen található értéket Lua szállá alakítja át (amit a `lua_State*` képvisel). Az értéknek szálnak kell lennie, egyébként a függvény visszatérési értéke `NULL`.

---

lua_touserdata

```
void *lua_touserdata (lua_State *L, int index);
```

Ha a megadott elfogadható indexen található érték teljes userdata, visszatérési értéke a blokk címe. Ha az érték könnyű userdata, visszatérési értéke a mutatója. Egyéb esetben a visszatérési érték `NULL`.

---

lua_type

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

Visszatérési értéke a megadott elfogadható indexen található érték típusa, vagy `LUA_TNONE` érvénytelen index esetén (tehát olyan index, ami "üres" verempozícióra mutat). A `lua_type` által visszatérő típuskódok a `lua.h` fájlban vannak definiálva: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, és `LUA_TLIGHTUSERDATA`.

---

lua_typename

```
const char *lua_typename (lua_State *L, int tp);
```

Visszatérési értéke a típus neve, amelyet a `tp` értéke határoz meg, ami mindig a `lua_type` függvény egyik visszatérési értéke.

---

lua_Writer

```
typedef int (*lua_Writer) (lua_State *L,
 const void* p,
 size_t sz,
 void* ud);
```

Az író függvényt a `lua_dump` használja. Minden alkalommal, amikor egy csonk újabb darabját elkészíti, a `lua_dump` meghívja az írót, végigmenve az írandó (`p`) bufferen, az (`sz`) méretén, és a `data` `lua_dump` által átadott paraméteren.

Az író hibakóddal tér vissza: 0 esetén nem történt hiba; bármilyen más érték hibát jelent, és megállítja a `lua_dump` függvényt, hogy ne hívja újra az írót.

---

lua_xmove

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

Értékeket cserél ki *azonos* globális állapoton belül lévő különböző szálak között.

A függvény kiemel `n` értéket a veremből `from` pozíciótól kezdve, és a `to` verembe helyezi őket.

---

lua_yield

```
int lua_yield (lua_State *L, int nresults);
```

Szüneteltet egy korutint.

Ez a függvény csak egy C függvény visszatérési értékeként hívható meg, a következők szerint:

```
 return lua_yield (L, nresults);
```

Amikor a C függvény ezen a módon hívja meg a `lua_yield`-et, a futó korutin szünetelteti a futását, valamint az a függvény, amelyik kiadta a `lua_resume` hívást, visszatér. A `nresults` paraméter a `lua_resume` függvénynek átadott paraméterek száma a veremben.

3.8 - A Debug Interfész

A Lua nyelvben nincs beépített hibakeresési szolgáltatás. Ezzel szemben egy speciális felületet biztosít függvények és *hurkok* segítségével. Ez a felület biztosítja, hogy debuggerek, profilozók és egyéb olyan eszközök is létrehozhatóak legyenek, amelyek belső információkat adnak a feldolgozóról.

---

lua_Debug

```
typedef struct lua_Debug {
 int event;
 const char *name; /* (n) */
 const char *namewhat; /* (n) */
 const char *what; /* (S) */
 const char *source; /* (S) */
 int currentline; /* (l) */
 int nups; /* (u) number of upvalues */
 int linedefined; /* (S) */
 int lastlinedefined; /* (S) */
 char short_src[LUA_IDSIZE]; /* (S) */
 /* private part */
 other fields
```

```
} lua_Debug;
```

A struktúra az aktív függvény különböző információit tartalmazza. A `lua_getstack` ennek a struktúrának csak a privát részeit tölti fel, későbbi használatra. A `lua_Debug` többi mezőjének hasznos információkkal való feltöltésére a `lua_getinfo` használható.

A `lua_Debug` mezőinek jelentése:

- **`source`:** Ha a függvény karakterláncként lett definiálva, akkor a `source` értéke az adott karakterlánc. Ha a függvény egy fájlban van definiálva, akkor a `source` egy '`@`'jellel kezdődik, és a fájl nevével folytatódik.
- **`short_src`:** A `source` "nyomtatható" változata, hibaüzenetekben van használatban.
- **`linedefined`:** A sor száma, ahol a függvény definíciója kezdődik.
- **`lastlinedefined`:** A sor száma, ahol a függvény definíciója befejeződik.
- **`what`:** Értéke a `"Lua"` karakterlánc, ha az adott függvény egy Lua függvény, `"C"`, ha C függvény, `"main"`, ha egy csonk fő része, és `"tail"`, ha egy olyan függvény volt, ami véghívást hajtott végre. Utóbbi esetben, a Lua-nak nincs további információja a függvényről.
- **`currentline`:** Az adott függvény végrehajtódó sora. Ha nincs információ a sorról, a `currentline` értéke -1.
- **`name`:** Az adott függvény neve elfogadható formában. Mivel a Lua-ban a függvények első-osztályú értékek, nincs állandó nevük: néhány függvény több globális változó értéke is lehet, míg mások csak tömbmezőként tárolódhatnak. A `lua_getinfo` függvény ellenőrzi, hogy a függvény hogyan lett meghívva, hogy megfelelő nevet találjon neki. Ha nem talál ilyet, akkor a `name` értéke `NULL`.
- **`namewhat`:** Megmagyarázza a `name` mezőt. A `namewhat` értéke lehet `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"`, vagy `""` (üres karakterlánc), attól függően, hogy a függvény hogyan lett meghívva. (a Lua üres karakterláncot használ, amikor más opció nem tűnik elfogadhatónak.)
- **`nups`:** A függény upvalue értékeinek száma.

`lua_gethook`

```
lua_Hook lua_gethook (lua_State *L);
```

Visszatérési értéke a jelenlegi hurok függvény.

`lua_gethookcount`

```
int lua_gethookcount (lua_State *L);
```

Visszatérési értéke a jelenlegi hurkok száma.

---

lua_gethookmask

```
int lua_gethookmask (lua_State *L);
```

Visszatérési értéke a jelenlegi hurok maszk.

---

lua_getinfo

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

Információkkal tér vissza a megadott függvényről vagy függvényhívásról.

Hogy információt kapjunk egy függvényhívásról, az `ar` paraméternek valós aktivizáló rekordnak kell lennie, amelyet előzőleg a `lua_getstack` töltött fel, vagy a hurok argumentumaként lett átadva (lásd `lua_Hook`).

Hogy információt nyerhessünk egy függvényről, először a verembe kell helyezni, és a `what` karakterláncnak a '`>`' karakterrel kell kezdődnie. (Ebben az esetben, a `lua_getinfo` kiemeli a függvény a veremből.) Például, hogy megtudhassuk, az `f` függvény melyik sorban lett definiálva, a következő kód használható:

```
lua_Debug ar;
lua_getfield(L, LUA_GLOBALSINDEX, "f"); /* globális 'f' lekérése */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

A `what` karakterláncban minden egyes karakter az `ar` struktúra egy bizonyos mezőjét tölti ki, vagy egy érték, amely a verembe lesz helyezve:

- **'`n`':** Kitölti a `name` és `namewhat` mezőket;
- **'`s`':** Kitölti a `source`, `linedefined`, `lastlinedefined`, `what`, és `short_src` mezőket;
- **'`l`':** Kitölti a `currentline` mezőt;
- **'`u`':** Kitölti a `nups` mezőt;
- **'`f`':** A verembe helyezi a megadott szinten futó függvényt;
- **'`L`':** A verembe helyezi azt a tömböt, amelynek indexei a függvény érvényes sorainak száma. (Egy *érvényes sor* olyan sort jelent, amelyhez kód társul, így olyan sor, ahol töréspont helyezhető el. A nem érvényes sorokhoz tartoznak az üres sorok és a megjegyzések.)

A függvény visszatérési értéke 0 hiba esetén (például ha a `what` értéke érvénytelen).

---

lua_getlocal

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
```

Információt ad a megadott aktivizáló rekordon található lokális változóról. Az `ar` paraméternek valós aktivizáló rekordnak kell lennie, amelyet előzőleg a `lua_getstack` töltött fel, vagy a hurok argumentumaként lett átadva (lásd `lua_Hook`). Az `n` index választja ki a megvizsgálandó lokális változót (1 az első paraméter vagy aktív lokális változó, és így tovább, egészen az utolsó aktív lokális változóig.) A `lua_getlocal` a változók értékét a verembe helyezi, és a nevükkel tér vissza.

Az olyan változónevek, amelyek '`(`' jellel (nyitó zárójellel) kezdődnek, belső változókat képviselnek (ciklusvezérlő változók, ideiglenes változók és C függvények lokális változói).

Visszatérési értéke `NULL` (és nem helyez a verembe semmit), ha az index nagyobb, mint az aktív lokális változók száma.

---

lua_getstack

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

Információt ad a feldolgozó futási veméről.

Ez a függvény részlegesen feltölti a `lua_Debug` struktúrát a megadott szinten futó függvény *aktivációs rekordjának* azonosítójával. A 0. szint a jelenlegi függvény, az *n+1*. szint pedig a jelenlegi függvény által hívott *n*. szint. Ha nem történik hiba, a `lua_getstack` visszatérési értéke 1; ha magasabb szinttel lesz meghívva, mint az aktuális veremméret, akkor a visszatérési érték 0.

---

lua_getupvalue

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
```

Lekéri egy zárvány felsőértékeit. (Lua függvények esetén a felsőértékek olyan külső lokális változók, amelyeket a függvény használ, és így következésképpen a zárványa is tartalmazza). A `lua_getupvalue` az *n*. szintű felsőértéket kéri le, majd ennek értékét a verembe helyezi, és a nevével tér vissza. `funcindex` a zárvány verembeli helyére mutat. (A felső értékeknek nincs egyéni rendezettségük, mivel az egész függvényen keresztül aktívak. Emiatt tetszőleges sorrendben lesznek számozva.)

Visszatérési értéke `NULL` (és nem helyez a verembe semmit) ha az index nagyobb, mint a felsőértékek száma. C függvények esetén ez a függvény az üres karakterláncot (`""`) használja a felsőértékek neveiként.

---

lua_Hook

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

A hibakeresési hurokfüggvények típusa.

Amikor egy hurok meg lesz hívva, az `ar` argumentumnak van egy `event` mezője, amely azt a megadott eseményt jelöli, amely elindította a hurkot. A Lua a következő konstansokkal azonosítja ezeket az eseményeket: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILRET`, `LUA_HOOKLINE`, és `LUA_HOOKCOUNT`. Ezen felül a sor eseményeknél a `currentline` mező is be lesz állítva. Az `ar` egyéb mezőinek lekéréséhez a huroknak meg kell hívnia a `lua_getinfo` függvényt. A visszatérési események esetén az `event` lehet `LUA_HOOKRET`, az alapértelmezett érték, vagy `LUA_HOOKTAILRET`. A második esetben a Lua egy visszatérést szimulál a függvényből, ami véghívást hajtott végre; ebben az esetben nem szükséges a `lua_getinfo` hívása.

Amíg a Lua futtatja a hurkot, letiltja a hurok egyéb hívását. Tehát ha egy hurok meghívja a Lua-t, hogy hajtson végre egy függvényt vagy csonkot, a futtatás hurkok meghívása nélkül fog lezajlani.

---

lua_sethook

```
int lua_sethook (lua_State *L, lua_Hook func, int mask, int count);
```

Beállítja a hibakeresési hurokfüggvényt.

A `func` a hurokfüggvény. A `mask` értéke adja meg, hogy milyen eseményeknél legyen meghívva a hurok: a kialakítása a konstansok bitenkénti 'or'-al lett létrehozva `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, és `LUA_MASKCOUNT`. A `count` argumentum csak akkor van használatban, ha a maszk tartalmazza a `LUA_MASKCOUNT` konstanst. Egyes események esetén a hurok az alábbiak szerint lesz meghívva:

- **A hívó hurok:** Akkor lesz meghívva, amikor a feldolgozó meghív egy függvényt. A hurok akkor lesz meghívva, amikor a Lua belép az új függvénybe, de mielőtt a függvény megkapná az argumentumait.
- **A visszatérési hurok:** Akkor lesz meghívva, amikor a feldolgozó visszatér egy függvényből. A hurok akkor lesz meghívva, mielőtt a lua elhagyná a függvényt. A függvény visszatérési értékei nem elérhetőek innen.
- **A sor hurok:** Akkor lesz meghívva, amikor a feldolgozó egy újabb sor végrehajtását megkezdi, vagy visszafelé ugrik a kódban (még ha azonos sorhoz is). (Ez az esemény csak akkor történhet meg, amikor a Lua egy Lua függvényt hajt végre.)
- **A számláló hurok:** Akkor lesz meghívva, amikor a feldolgozó végrehajt minden `count` utasítást. (Ez az esemény csak akkor történhet meg, amikor a Lua egy Lua függvényt hajt végre.)

A hurok a `mask` zéró értékével kapcsolható ki.

---

lua_setlocal

```
const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
```

A megadott aktivációs rekord lokális változóját a megadott értékkel látja el. Az `ar` és `n` paraméterek ugyanazok, mint a `lua_getlocal` függvény esetén (lásd `lua_getlocal`). A `lua_setlocal` a verem tetején lévő értéket adja a változónak, és a nevével tér vissza. Az értéket kiemeli a veremből.

Visszatérési értéke `NULL` (és nem emel ki semmit a veremből) ha az index nagyobb, mint az aktív lokális változók száma.

`lua_setupvalue`

```
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

A megadott zárvány felsőértékének értékét állítja be. A `funcindex` és `n` paraméterek ugyanazok, mint a `lua_getupvalue` függvény esetén (lásd `lua_getupvalue`). A verem tetején lévő értéket adja a felsőértéknek, és a nevével tér vissza. Az értéket kiemeli a veremből.

Visszatérési értéke `NULL` (és nem emel ki semmit a veremből) ha az index nagyobb, mint a felsőértékek száma.

4 - A segédkönyvtár

A *segédkönyvtár* több kényelmi függvényt is biztosít a C és a Lua között. Míg az alap API primitív függvények segítségével biztosítja a párbeszédet a C és a Lua között, addig a segédkönyvtár magasabb szintű függvényeket biztosít néhány általános feladat ellátásához.

A segédkönyvtár összes függvénye a `lauxlib.h` fejlécfájlban van definiálva, és a `luaL_` prefixszel rendelkezik.

A segédkönyvtár minden függvénye az alap API alapján épül fel, így semmi olyat nem tartalmaz, amit nem lehetne megoldani ezen API használatával.

A segédkönyvtár több függvénye is ellenőrzi a C függvények argumentumait. Ezeknek a neve mindig `luaL_check*` vagy `luaL_opt*` kifejezéssel kezdődik. Ezek mindig hibát érnek el, ha az ellenőrzés sikertelen. Mivel a hibaüzenetek az argumentumoknak megfelelően vannak formázva (pl., "`bad argument #1`"), ezek a függvények nem használhatóak más verem értékek esetén.

4.1 - Függvények és típusok

A következőkben szerepelnek a segédkönyvtár függvényei és típusai, ABC sorrendben.

---

luaL_addchar

```
void luaL_addchar (luaL_Buffer *B, char c);
```

A `c` karaktert a `B` bufferhez adja (lásd `luaL_Buffer`).

---

luaL_addlstring

```
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
```

Az `s` által mutatott, `l` hosszúságú karakterláncot a `B` bufferhez adja (lásd `luaL_Buffer`). A karakterlánc tartalmazhat beágyazott zérókat.

---

luaL_addsize

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

Egy előzőleg a buffer területre másolt (lásd `luaL_prepbuffer`) `n` hosszúságú karakterláncot a `B` bufferhez adja (lásd `luaL_Buffer`).

---

luaL_addstring

```
void luaL_addstring (luaL_Buffer *B, const char *s);
```

Az `s` által mutatott zéróvégű karakterláncot a `B` buffehez adja (lásd `luaL_Buffer`). A karakterlánc nem tartalmazhat beágyazott zérókat.

---

luaL_addvalue

```
void luaL_addvalue (luaL_Buffer *B);
```

A verem tetején lévő értéket a `B` bufferhez adja (lásd `luaL_Buffer`). Az értéket kiemeli a veremből.

Ez az egyetlen olyan karakterlánc-buffer művelet, amely hívásakor extra elemet vár a veremben, méghozzá a bufferhez adandó értéket.

---

luaL_argcheck

```
void luaL_argcheck (lua_State *L,
 int cond,
 int numarg,
```

```
 const char *extramsg);
```

Ellenőrzi, hogy a megadott `cond` feltétel igaz -e. Ha nem, egy hibát ér el a következő hibaüzenettel, ahol `func` a hívó veremből származik:

```
 bad argument #<numarg> to <func> (<extramsg>)
```

---

luaL_argerror

```
int luaL_argerror (lua_State *L, int numarg, const char *extramsg);
```

Egy hibát ér el a következő üzenettel, ahol `func` a hívó veremből származik:

```
 bad argument #<numarg> to <func> (<extramsg>)
```

A függvény soha nem tér vissza, mivel ez egy olyan kifejezés, amely C függvényekben a következőként használható: `return luaL_argerror(`*`args`*`)`.

---

luaL_Buffer

```
typedef struct luaL_Buffer luaL_Buffer;
```

A *karakterlánc buffer* típusa.

Egy karakterlánc buffer lehetővé teszi a C nyelvben Lua karakterláncok létrehozását. Használatára a következők vonatkoznak:

- Először deklarálni kell `luaL_Buffer` típusú *b* változót.
- Ezután elő kell készíteni azt a `luaL_buffinit(L, &b)` hívással.
- Ezután a karakterlánc darabjait hozzá kell adni a bufferhez valamelyik `luaL_add*` függvénnyel.
- Végül meg kell hívni a `luaL_pushresult(&b)` függvényt. Ez a hívás a verem tetején hagyja a kész karakterláncot.

A normál műveletek közben a karakterlánc buffer változó számú verem helyet használ. Így, amíg egy buffer használatban van, nem lehet tudni, pontosan hol van a verem teteje. A verem a sikeres bufferművelet-hívások között használható, feltéve, ha azok kiegyensúlyozottak; azaz egy bufferművelet hívásakor a verem azonos szinten van, mint az előző bufferművelet előtt. (az egyetlen kivétel ez alól a szabály alól a `luaL_addvalue`). A `luaL_pushresult` hívása után a verem visszatér az előkészítés előtti szintre, plusz a verem tetején lesz a kész karakterlánc.

---

luaL_buffinit

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
```

Előkészíti a `B` buffert. A függvény nem foglal le helyet; a buffert változóként kell deklarálni (lásd `luaL_Buffer`).

---

luaL_callmeta

```
int luaL_callmeta (lua_State *L, int obj, const char *e);
```

Meghív egy metaeljárást.

Ha az `obj` indexnél lévő objektumnak van metatömbje, és ennek van `e` mezője, ez a függvény meghívja ezt a mezőt és az objektumot adja egyetlen argumentumául. Ebben az esetben a függvény visszatérési értéke 1 és a verembe helyezi a hívásból visszatérő értéket. Ha nincs metatömbje, vagy nincs ilyen metaeljárása, a függvény visszatérési értéke 0 (és semmit nem helyez el a veremben).

---

luaL_checkany

```
void luaL_checkany (lua_State *L, int narg);
```

Ellenőrzi, hogy a függvénynek van -e valamelyik típusú argumentuma (beleértve a **nil**-t is) `narg` pozícióban.

---

luaL_checkint

```
int luaL_checkint (lua_State *L, int narg);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum szám -e, és visszatérési értéke ez a szám `int` formában.

---

luaL_checkinteger

```
lua_Integer luaL_checkinteger (lua_State *L, int narg);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum szám -e, és visszatérési értéke ez a szám `lua_Integer` formában.

---

luaL_checklong

```
long luaL_checklong (lua_State *L, int narg);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum szám -e, és visszatérési értéke ez a szám `long` formában.

---

luaL_checklstring

```
const char *luaL_checklstring (lua_State *L, int narg, size_t *l);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum karakterlánc -e, és ezzel tér vissza; ha `l` nem `NULL`, `*l` értéke a karakterlánc hossza lesz.

---

luaL_checknumber

```
lua_Number luaL_checknumber (lua_State *L, int narg);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum szám -e, és ezzel tér vissza.

---

luaL_checkoption

```
int luaL_checkoption (lua_State *L,
 int narg,
 const char *def,
 const char *const lst[]);
```

Ellenőrzi, hogy a `narg` argumentum karakterlánc -e, és megkeresi ezt a `lst` tömbben (amelynek NULL-végződésűnek kell lennie). Visszatérési értéke az az index, ahol a karakterlánc található a tömbben. Egy hibát ér el, ha az argumentum nem karakterlánc, vagy a karakterlánc nem található.

Ha a `def` nem `NULL`, a függvény a `def` értékét használja alapértelmezett értékként, ha nincs `narg` argumentum, vagy annak értéke **nil**.

Ez főleg akkor hasznos, ha karakterláncokat akarunk megfeleltetni C felsorolásnak (enum). (a Lua függvénykönyvtárakban megállapodás, hogy karakterláncokat kell használni számok helyett a lehetőségek kiválasztásakor.)

---

luaL_checkstack

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

A verem méretét `top + sz` eleműre bővíti, hibát ér el, ha a verem nem növelhető akkorára. Az `msg` egy kiegészítő üzenet, amely a hibaüzenetben fog szerepelni.

---

luaL_checkstring

```
const char *luaL_checkstring (lua_State *L, int narg);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum karakterlánc -e, és ezzel tér vissza.

---

luaL_checktype

```
void luaL_checktype (lua_State *L, int narg, int t);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum `t` típusú -e.

---

luaL_checkudata

```
void *luaL_checkudata (lua_State *L, int narg, const char *tname);
```

Ellenőrzi, hogy a `narg` pozícióban lévő argumentum `tname` típusú userdata -e (lásd `luaL_newmetatable`).

---

luaL_dofile

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

Betölti és futtatja a megadott fájlt. Ez a következő makróként van definiálva:

```
 (luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Visszatérési értéke 0, ha nem volt hiba, illetve 1 valamilyen hiba esetén.

---

luaL_dostring

```
int luaL_dostring (lua_State *L, const char *str);
```

Betölti és futtatja a megadott karakterláncot. Ez a következő makróként van definiálva:

```
 (luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
```

Visszatérési értéke 0, ha nem volt hiba, illetve 1 valamilyen hiba esetén.

---

luaL_error

```
int luaL_error (lua_State *L, const char *fmt, ...);
```

Elér egy hibát. A hibaüzenet formátumát a `fmt` és annak extra argumentumai szablyák meg, a `lua_pushfstring` szabályainak megfelelően. A hibaüzenet elejére kerül a fájlnév és a sor száma, ahol a hiba történt, ha ez az információ elérhető.

A függvény soha nem tér vissza, mivel ez egy olyan kifejezés, amely C függvényekben a következőként használható: `return luaL_error(`*`args`*`)`.

---

luaL_getmetafield

```
int luaL_getmetafield (lua_State *L, int obj, const char *e);
```

Az `obj` indexen lévő objektum metatömbjének `e` mezőjét helyezi a verembe. Ha az objektumnak nincs metatömbje, vagy annak nincs ilyen mezője, visszatérési értéke 0, és nem helyez semmit a verembe.

---

luaL_getmetatable

```
void luaL_getmetatable (lua_State *L, const char *tname);
```

A verembe helyezi a registryben a `tname` névhez társított metatömböt (lásd `luaL_newmetatable`).

---

luaL_gsub

```
const char *luaL_gsub (lua_State *L,
 const char *s,
 const char *p,
 const char *r);
```

Elkészíti az `s` karakterlánc másolatát, amelyben a `p` karakterlánc összes előfordulását `r` karakterláncra cseréli. A verembe helyezi a kész karakterláncot, majd azzal tér vissza.

---

luaL_loadbuffer

```
int luaL_loadbuffer (lua_State *L,
 const char *buff,
 size_t sz,
 const char *name);
```

A megadott buffert lua csonkként tölti be. A függvény a `lua_load`-ot használja a `buff` által mutatott `sz` méretű buffer csonkba töltéséhez.

A függvény ugyanazokkal az eredményekkel tér vissza, mint a `lua_load`. A `name` a csonk neve, amely hibakereső információknál ls hibaüzeneteknél van használatban.

---

luaL_loadfile

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

A megadott fájlt Lua csonkként tölti be. A függvény a `lua_load`-ot használja a `filename` nevű fájl csonkba töltéséhez. Ha a `filename` értéke `NULL`, akkor az alapértelmezett bemenetből tölt be. Az első sor nem lesz figyelembe véve, ha az `#` jellel kezdődik.

A függvény ugyanazokkal az eredményekkel tér vissza, mint a `lua_load`, azonban van egy extra hibakódja is, `LUA_ERRFILE`, ha fájl nem nyitható meg, vagy nem olvasható.

A `lua_load`-hoz hasonlóan, ez a függvény csak betölti a csonkot, de nem futtatja azt.

---

luaL_loadstring

```
int luaL_loadstring (lua_State *L, const char *s);
```

A megadott karakterláncot Lua csonkként tölti be. A függvény a `lua_load`-ot használja a zéró-végződésű `s` karakterlánc csonkba töltéséhez.

A függvény ugyanazokkal az eredményekkel tér vissza, mint a `lua_load`.

A `lua_load`-hoz hasonlóan, ez a függvény csak betölti a csonkot, de nem futtatja azt.

---

luaL_newmetatable

```
int luaL_newmetatable (lua_State *L, const char *tname);
```

Ha a registry már tartalmaz `tname` kulcsot, 0-val tér vissza. Egyéb esetben egy új tömböt készít, amely a userdata metatömbje lesz, a registryhez adja `tname` kulccsal, és 1-el tér vissza.

Mindkét esetben a verembe helyezi a registryben a `tname` kulcshoz rendelt értéket.

---

luaL_newstate

```
lua_State *luaL_newstate (void);
```

Egy új Lua állapotot készít, amelyhez a `lua_newstate`-t egy lefoglaló függvénnyel hívja meg, amely a szabványos C `realloc` függvényen alapszik, valamint beállít egy pánik függvényt (lásd `lua_atpanic`), amely az alapértelmezett hibakimeneten jeleníti meg a hibaüzenetet fatális hiba esetén.

Visszatérési értéke az új állapot, vagy `NULL`, ha memória-lefoglalási hiba történt.

---

luaL_openlibs

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

Megnyitja az összes szabványos Lua eljáráskönyvtárat a megadott állapot számára.

---

luaL_optint

```
int luaL_optint (lua_State *L, int narg, int d);
```

Ha a `narg` pozícióban lévő argumentum szám, visszatérési értéke ez a szám `int` formában. Ha ez az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

---

luaL_optinteger

```
lua_Integer luaL_optinteger (lua_State *L, int narg, lua_Integer d);
```

Ha a `narg` pozícióban lévő argumentum szám, visszatérési értéke ez a szám `lua_Integer` formában. Ha ez az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

---

luaL_optlong

```
long luaL_optlong (lua_State *L, int narg, long d);
```

Ha a `narg` pozícióban lévő argumentum szám, visszatérési értéke ez a szám `long` formában. Ha ez az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

---

luaL_optlstring

```
const char *luaL_optlstring (lua_State *L,
 int narg,
 const char *d,
```

```
 size_t *l);
```

Ha a `narg` pozícióban lévő argumentum karakterlánc, azzal tér vissza. Ha ez az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

Ha `l` értéke nem `NULL`, az `*l` pozíciót az eredmény hosszával megegyező értékre állítja.

---

luaL_optnumber

```
lua_Number luaL_optnumber (lua_State *L, int narg, lua_Number d);
```

Ha a `narg` pozícióban lévő argumentum szám, visszatérési értéke ez a szám. Ha ez az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

---

luaL_optstring

```
const char *luaL_optstring (lua_State *L, int narg, const char *d);
```

Ha a `narg` pozícióban lévő argumentum karakterlánc, azzal tér vissza. Ha az argumentum hiányzik, vagy **nil**, visszatérési értéke `d`. Egyéb esetben hibát okoz.

---

luaL_prepbuffer

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

Visszatérési értéke `LUAL_BUFFERSIZE` méret helyének címzése, ahová egy `B` bufferhez adandó karakterlánc másolható (lásd `luaL_Buffer`). Miután a karakterlánc erre a helyre lett másolva, meg kell hívni a `luaL_addsize` függvényt a karakterlánc méretével, hogy ténylegesen is a bufferbe kerüljön.

---

luaL_pushresult

```
void luaL_pushresult (luaL_Buffer *B);
```

Befejezi a `B` buffer használatát, a kész karakterláncot a verem tetején hagyva.

---

luaL_ref

```
int luaL_ref (lua_State *L, int t);
```

Elkészít egy *hivatkozást* a `t` indexnél található tömbhöz, a verem tetején lévő objektumhoz, és visszatér vele (kiemeli az objektumot a veremből).

A hivatkozás egy egyedi egész típusú kulcs. Amíg manuálisan nem kerülnek a `t` tömbbe egész típusú kulcsok, a `luaL_ref` megbizonyosodik a kulcsok egyedülállóságáról visszatérés előtt. Egy `r` által hivatkozott objektum lekérhető a `lua_rawgeti(L, t, r)` hívással. A `luaL_unref` függvény felszabadítja a hivatkozást, valamint a hozzá társított objektumot is.

Ha a verem tetején található objektum **nil**, a `luaL_ref` visszatérési értéke a `LUA_REFNIL` konstans. A `LUA_NOREF` konstans garantátan eltérő bármilyen `luaL_ref` által visszatérő hivatkozástól.

---

luaL_Reg

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

A `luaL_register` által regisztrálandó függvénytömbök típusa. A `name` a függvény neve, a `func` pedig egy mutató a függvényhez. A `luaL_Reg` minden tömbjének egy jelzős bejegyzéssel kell végződnie, ahol mind a `name`, mind a `func` értéke `NULL`.

---

luaL_register

```
void luaL_register (lua_State *L,
 const char *libname,
 const luaL_Reg *l);
```

Megnyit egy eljáráskönyvtárat.

Ha a `libname` értéke `NULL`, az összes függvényt regisztrálja az `l` listából (lásd `luaL_Reg`) a verem tetején lévő tömbe.

Ha nem-null `libname` argumentummal lesz meghívva, egy új `t` tömböt készít, amelyet a globális `libname` változó értékeként, valamint a `package.loaded[libname]` értékeként állítja be, és regisztrálja az összes függvényét az `l` listában. Ha van ilyen tömb a `package.loaded[libname]` címen, vagy a globális `libname` változóban, újra felhasználja ezt a tömböt ahelyett, hogy egy újat készítene.

A függvény minden esetben a verem tetején hagyja a tömböt.

---

luaL_typename

```
const char *luaL_typename (lua_State *L, int idx);
```

A megadott `idx` indexen található érték típusának nevével tér vissza.

---

luaL_typerror

```
int luaL_typerror (lua_State *L, int narg, const char *tname);
```

A következőhöz hasonló hibaüzenetet állít elő:

```
 <location>: bad argument <narg> to <function> (<tname> expected, got
<realt>)
```

ahol `<location>` a `luaL_where`-ből származik, a `<function>` a jelenlegi függvény neve, és a `<realt>` az aktuális argumentum típusának neve.

---

luaL_unref

```
void luaL_unref (lua_State *L, int t, int ref);
```

Törli a `ref` hivatkozást a `t` indexen található tömbből (lásd `luaL_ref`). A bejegyzés el lesz távolítva a tömbből, így a hivatkozott objektum is összegyűjthető. A `ref` hivatkozás szintén felszabadul, és újra felhasználható.

Ha a `ref` értéke `LUA_NOREF` vagy `LUA_REFNIL`, a `luaL_unref` nem csinál semmit.

---

luaL_where

```
void luaL_where (lua_State *L, int lvl);
```

A verembe helyez egy karakterláncot, amely azonosítja a vezérlés jelenlegi pozícióját `lvl` szinten a hívó veremben. Jellemzően a karakterlánc formátuma a következő: `<chunkname>:<currentline>:`. A 0. szint a futó függvény, 1., amelyik hívta a futó függvényt, és így tovább.

Ez a függvény a hibaüzenetek előtagjának előállításakor van használatban.

5 - Szabványos függvénykönyvtárak

A Lua szabványos függvénykönyvtárai hasznos függvényeket biztosítanak, amelyek közvetlenül a C API-n keresztül vannak megvalósítva. Ezek közül a függvények közül néhány a nyelvben nélkülözhetetlen szolgáltatást biztosít (pl., `type` és `getmetatable`); mások "külső" hozzáféréseket (pl., I/O); míg megint mások a Lua-ban magában

vannak megvalósítva, de ezek vagy különösen hasznosak, vagy olyan kritikus teljesítményigényük van, hogy érdemes volt megvalósítani C-ben (pl., `sort`).

Az összes függvénykönyvtár a hivatalos C API-n keresztül, különböző C modulokként lett megvalósítva. Jelenleg a következő szabványos függvénykönyvtárak léteznek a Lua nyelvben:

- alap függvénykönyvtár;
- csomag függvénykönyvtár;
- karakterlánc műveletek;
- tömb műveletek;
- matematikai függvények (sin, log, stb.);
- bemenet és kimenet;
- operációs rendszer szolgáltatások;
- hibakereső szolgáltatások.

Az alap és a csomag függvénykönyvtárakat leszámítva minden egyes függvénykönyvtár függvényei egy globális tömb mezőiként, vagy az objektumának eljárásaiként érhetőek el.

Ezeknek a függvénykönyvtáraknak az eléréséhez a C kliensprogramnak meg kell hívnia a `luaL_openlibs` függvényt, amely megnyitja az összes alapértelmezett függvénykönyvtárat. Lehetőség van arra is, hogy ezeket egyenként nyissuk meg a következő hívásokkal: `luaopen_base` (az alap függvénykönyvtárhoz), `luaopen_package` (a csomag függvénykönyvtárhoz), `luaopen_string` (a karakterlánc függvénykönyvtárhoz), `luaopen_table` (a tömb függvénykönyvtárhoz), `luaopen_math` (a matematikai függvénykönyvtárhoz), `luaopen_io` (az I/O és operációs rendszer függvénykönyvtárakhoz), és `luaopen_debug` (a hibakereső függvénykönyvtárhoz). Ezek a függvények a `lualib.h` fájlban vannak deklarálva, és közvetlenül nem hívhatóak meg: a többi Lua C függvényhez hasonlóan működnek, pl., a `lua_call` használatával hívhatóak meg.

5.1 - Alap függvények

Az alap függvénykönyvtár néhány alapvető fontosságú függvényt biztosít a Lua számára. Ha ez az eljáráskönyvtár nincs betöltve a programban, különösen óvatosan kell eljárni, hogy szükséges -e annak valamilyen szolgáltatásait megvalósítani.

`assert (v [, message])`

Egy hibát okoz, amikor a `v` argumentum hamis (tehát **nil** vagy **false**); egyébként az argumentumokkal tér vissza. A `message` egy hibaüzenet, ha nincs megadva, az alapértelmezett üzenet lesz használva: "assertion failed!"

`collectgarbage (opt [, arg])`

A függvény egy általános felületet hoz létre a szemétgyűjtőhöz. Az első, `opt` függvénytől függően különböző függvényeket hajthat végre:

- **"stop":** megállítja a szemétgyűjtőt.
- **"restart":** újraindtja a szemétgyűjtőt.
- **"collect":** elindít egy teljes szemétgyűjtő kört.
- **"count":** a Lua által használt memória pillanatnyi értékét adja vissza (Kbyte-okban).
- **"step":** végrehajt egy növekményes szemétgyűjtő-lépést. A lépés méretét az `arg` határozza meg (nagyobb érték mellett több lépés) meghatározatlan módon. Ha a lépés méretét szabályozni akarod, az `arg` értékével kell kísérletezni. Visszatérési értéke **true**, ha a lépéssel befejeződütt egy szemétgyűjtő kör.
- **"setpause":** A gyűjtő *szünetét* állítja be `arg`/100 értékre (lásd §2.10).
- **"setstepmul":** A gyűjtő *lépésszorzóját* állítja be `arg`/100 értékre (lásd §2.10).

`dofile (filename)`

Megnyitja az adott fájlnevet, majd annak tartalmát Lua csonkként végrehajtja. Ha argumentumok nélkül van meghívva, az alapértelmezett bemenet (`stdin`) tartalmát hajtja végre. A csonk összes visszatérési értékével tér vissza. Hiba esetén a `dofile` továbbítja a hibát a hívónak (tehát a `dofile` nem futtatható védett módban).

`error (message [, level])`

Leállítja az utoljára hívott védett függvényt, és a `message` argumentummal, mint hibaüzenettel tér vissza. Az `error` függvény soha nem tér vissza.

Legtöbb esetben az `error` a hibaüzenet elejére fűz valamilyen információt a hiba helyéről. A `level` argumentum adja meg, hogyan legyen lekérve a hiba helye. Az 1-es érték (alapértelmezett érték) esetén a hiba helye az, ahonnan az `error` függvény meg lett hívva. 2-es érték esetén az, ahonnan az `error`-t meghívó függvény meg lett hívva, és így tovább. 0 érték esetén nem lesz kiegészítő információ adva a hibaüzenethez.

`_G`

Egy olyan globális változó (nem függvény), amely a globális környezetet tartalmazza (tehát, `_G._G = _G`). A Lua önmagában nem használja ezt a változót; az értékeinek megváltoztatása nem befolyásolja a környezetet, és fordítva (A környezetek megváltoztatására a `setfenv` használható.)

---

getfenv (f)

A megadott függvény által használt környezettel tér vissza. Az `f` lehet egy Lua függvény, vagy egy szám, amely a függvény veremben lévő pozícióját adja meg: az 1-es szint az a függvény, amely meghívta a `getfenv` függvényt. Ha a megadott függvény nem Lua függvény, vagy `f` értéke 0, a `getfenv` visszatérési értéke a globális környezet. Az `f` alapértelmezett értéke 1.

---

getmetatable (object)

Ha a megadott `object` objektumnak nincs metatömbje, visszatérési értéke **nil**. Egyébként, ha az objektum metatömbjének van `"__metatable"` mezője, akkor az ahhoz társított értékkel tér vissza. Egyéb esetben a megadott objektum metatömbjével tér vissza.

---

ipairs (t)

Három értékkel tér vissza: egy iterátor függvénnyel, a `t` tömbbel és 0-val, így a következő szerkezet:

```
for i,v in ipairs(t) do body end
```

a (`1,t[1]`), (`2,t[2]`), ˇˇˇ, párokon fog végigiterálni, az első nem létező egész típusú kulcsig.

A bejárás közbeni értékváltoztatás veszélyeiről lásd a `next` függvény leírását.

---

load (func [, chunkname])

Betölt egy csonkot a `func` függvény segítségével, amely megadja annak darabjait. Minden egyes `func` hívásnak olyan karakterlánccal kell visszatérnie, amely összefűzi az előző eredményeket. **nil** (vagy hiányzó érték) visszatérése jelzi a csonk végét.

Ha nem történik hiba, visszatérési értéke a lefordított csonk függvényként; egyéb esetben **nil** és a hibaüzenet. A visszatérő függvény környezete a globális környezet.

A `chunkname` argumentum nevet ad a csonknak, amely a hibaüzenetekben, valamint hibakereséskor van használatban.

---

loadfile ([filename])

A `load`-hoz hasonlóan működik, de a csonkot a megadott `filename` fájlból, vagy ennek hiányában az alapértelmezett bementről tölti be.

---

loadstring (string [, chunkname])

A `load`-hoz hasonlóan működik, de a csonkot a megadott karakterláncból tölti be.

A megadott kifejezés betöltésére és végrehajtására a következő kifejezés használható:

```
assert(loadstring(s))()
```

---

next (table [, index])

Egy tömb összes mezőjének bejárására használható. Az első argumentuma a tömb, a második pedig a tömb egy indexe. A `next` visszatérési értéke a tömb következő indexe, és a hozzá tartozó érték. Ha második argumentuma **nil**, a visszatérési érték a kezdő index, és az ahhoz társított érték. Ha az utolsó indexszel van meghívva, vagy **nil**-el egy üres tömbön, a visszatérési érték is **nil**. Ha a második argumentum hiányzik, akkor az **nil**-nek lesz kiértékelve. A `next(t)` kifejezés arra is használható, hogy meggyőződjünk arról, hogy egy tömb üres -e.

Az indexek számlálásának sorrendje nincs meghatározva, *még szám-indexek esetén sem*. (Egy tömb sorrendben történő bejárására a numerikus **for** ciklus vagy az `ipairs` függvény használható.)

A `next` viselkedése *meghatározatlanná válik*, ha a bejárás közben egy nem létező mezőhöz érték adódik. A létező mezők azonban módosíthatóak, sőt törölhetőek is.

---

pairs (t)

Három visszatérési értéke van: a `next` függvény, a `t` tömb, és **nil**, így a következő szerkezet:

```
for k,v in pairs(t) do body end
```

a `t` tömb összes kulcs-érték párján végigiterál.

A bejárás közbeni értékváltoztatás veszélyeiről lásd a `next` függvény leírását.

---

pcall (f, arg1, ˇˇˇ)

Védett módban hívja meg az `f` függvényt a megadott argumentumokkal. Ez azt jelenti, hogy a függvényen belüli hibák nem lesznek továbbítva; ehelyett a `pcall`

elkapja a hibát, és egy állapotkóddal tér vissza. Az első visszatérési érték egy boolean típusú állapotkód, amely igaz, ha a hívás hiba nélkül fejeződik be. Ebben az esetben a hívás visszatérési értékei is visszatérnek az első után. Hiba esetén a visszatérési érték **false**, valamint a hibaüzenet.

`print (˘˘˘)`

Bármennyi argumentumot kaphat, és az értékeiket írja ki a `stdout`-ra. Az argumentumokat a `tostring` függvény alakítja karakterlácokká. A `print` nem formázott kimenet készítésére szánták, csak egy olyan gyors lehetőség, amellyel bizonyos értékek gyorsan megtekinthetőek, főleg hibakeresési céllal. Formázott kimenethez a `string.format` használható.

`rawequal (v1, v2)`

Bármilyen metaeljárás meghívása nélkül ellenőrzi, hogy `v1` és `v2` értéke megegyezik -e. Visszatérési értéke boolean.

`rawget (table, index)`

Bármilyen metaeljárás meghívása nélkül lekéri a `table[index]` valódi értékét. A `table` csak tömb lehet, míg az `index` bármilyen érték.

`rawset (table, index, value)`

Bármilyen metaeljárás meghívása nélkül `value` értékre állítja a `table[index]` valódi értékét. A `table` csak tömb lehet, `index` bármilyen **nil**-től különböző érték, `value` pedig bármilyen Lua érték.

A függvény visszatérési értéke a `table` tömb.

`select (index, ˘˘˘)`

Ha az `index` argumentum egy szám, a visszatérési érték az `index` pozíciójú argumentumok utáni összes argumentum. Egyébként az `index` értéke csak a `"#"` karakterlánc lehet: ekkor a `select` a megkapott extra argumentumok számával tér vissza.

`setfenv (f, table)`

A megadott függvény által használt környezetet cseréli le. Az `f` lehet egy Lua függvény, vagy egy szám, amely a függvény veremben lévő pozícióját adja meg: az 1-es szint az a függvény, amely meghívta a `setfenv` függvényt. Visszatérési értéke a megadott függvény

Abban az esetben, amikor `f` értéke 0, a `setfenv` a futó szál környezetét változtatja meg. Ebben az esetben nincs visszatérési érték.

---

setmetatable (table, metatable)

A megadott tábla metatömbjét állítja be. (A többi típus metatömbje nem változtatható meg a Lua-ból, csak a C-ből.) Ha a `metatable` értéke **nil**, törli a megadott tömb metatömbjét. Ha az eredeti táblának van `"__metatable"` mezője, hiba keletkezik.

A függvény visszatérési értéke a `table` tömb.

---

tonumber (e [, base])

Megpróbálja a megadott argumentumot számmá alakítani. Ha az argumentum már szám, vagy olyan karakterlánc, amely számmá alakítható, akkor visszatérési érték ez a szám, egyéb esetben **nil**.

Az opcionális paraméter adja meg a szám feldoglozásának számrendszerét. Ez 2 és 36 közötti egész szám lehet. A 10 feletti számrendszereknél az '`A`' (kis- vagy nagybetűs formában is) 10-et jelent, '`B`' 11-et, és így tovább, míg '`Z`' 35-öt. A 10-es számrendszernél (az alapértelmezett értéknél), a számnak lehet törtrésze valamint hatványkitevője is (lásd §2.1). A többi számrendszer csak előjel nélküli egész típust fogad el.

---

tostring (e)

Bármilyen típusú argumentumot kaphat, és ezt konvertálja elfogadható formátumú karakterlánccá. A számok konvertálásának teljes leírásához lásd: `string.format`.

Ha az `e` metatömbjének van `"__tostring"` mezője, akkor a `tostring` a hozzátartozó értéket hívja meg, `e` argumentummal, és az eredménye lesz ennek a hívásnak az eredménye.

---

type (v)

Visszatérési értéke az egyetlen argumentumának típusa, karakterlánc formában. A függvény lehetséges eredményei: "`nil`" (karakterlánc, és nem **nil** érték), "`number`", "`string`", "`boolean`", "`table`", "`function`", "`thread`", és "`userdata`".

---

unpack (list [, i [, j]])

Visszatérési értékei a megadott tömb elemei. A függvény megegyezik a következő kifejezéssel:

```
return list[i], list[i+1], ···, list[j]
```

kivéve, hogy a fenti kód csak meghatározott számú elem esetén használható. Alapértelmezés szerint `i` értéke 1 és `j` értéke a hossz operátor által meghatározott listahossz (lásd §2.5.5).

_VERSION

Egy globális változó (nem függvény), amely a jelenlegi feldolgozó verziószámát tartalmazza. A változó jelenlegi tartalma: "`Lua 5.1`".

---

xpcall (f, err)

A függvény hasonló a `pcall`-hoz, kivéve, hogy itt egy új hibakezelő is beállítható.

Az `xpcall` az `f` függvényt védett módban hívja meg, az `err` függvényt használva hibakezelőként. Az `f` függvényen belüli hibák nem lesznek továbbítva, ehelyett az `xpcall` elkapja a hibát, meghívja az `err` függvényt az eredeti hibaobjektummal, és egy állapotkóddal tér vissza. Az első visszatérési érték egy boolean típus, amelyik igaz, ha a hívás hiba nélkül befejeződik. Ebben az esetben a hívás visszatérési értékei is visszatérnek az első után. Hiba esetén a visszatérési érték **false**, valamint az `err` függvény eredménye.

5.2 - Korutin kezelés

A korutinokhoz kapcsolódó műveletek az alap függvénykönyvtár alkönyvtárához, valamint a `coroutine` tömbhöz tartoznak. A korutinok általános ismertetése a §2.11 fejezetben található.

---

coroutine.create (f)

Egy új korutint hoz létre, `f` testtel. Az `f` csak Lua függvény lehet. Visszatérési értéke a `"thread"` típusú új korutin.

---

coroutine.resume (co [, val1, ˇˇˇ])

Elkezdi, vagy folytatja a megadott `co` korutin futtatását. A korutin első folytatásakor annak testét futtatja. A `val1`, ˇˇˇ értékek a test-függvény argumentumaiként szerepelnek. Ha a korutin szüneteltetve lett, a `resume` újraindítja azt; a `val1`, ˇˇˇ értékek a szüneteltetés eredményei.

Ha a korutin hiba nélkül lefut, a `resume` visszatérési értéke **true**, valamint a `yield`-nek átadott értékek (ha a korutin szüneteltetve van) vagy bármilyen értékek, amelyek a test-függvény visszatérési értékei (ha a korutin befejeződik). Hiba esetén a `resume` visszatérési értéke **false** valamint a hibaüzenet.

---

coroutine.running ()

Visszatérési értéke a futó korutin, vagy **nil** ha a fő szálból lett meghívva.

---

coroutine.status (co)

Visszatérési értéke a `co` korutin állapota, karakterlánc alakban: `"running"`, ha a korutin fut (azaz, az hívta meg a `status`-t); `"suspended"`, ha a korutin a `yield` hívással fel lett függesztve, vagy még nem lett elindítva; `"normal"`, ha a korutin aktív, de nem fut (azaz, egy másik korutin folytatatta); valamint `"dead"`, ha a korutin befejezte a test-függvényt, vagy hibába ütközött.

---

coroutine.wrap (f)

Egy új korutint hoz létre, `f` testtel. Az `f` csak Lua függvény lehet. Visszatérési értéke egy függvény, amely minden egyes híváskor folytatja a korutint. A függvényhez társított argumentumok a `resume` extra argumentumai lesznek. Visszatérési értékei megegyeznek a `resume` hívásával, kivéve az első booleant. Hiba esetén továbbítja a hibát.

---

coroutine.yield (ˇˇˇ)

Felfüggeszti a hívó korutin futtatását. A korutin nem futtathat C függvényt, metaeljárást vagy iterátort. A `yield` argumentumai a `resume` extra visszatérési értékei lesznek.

5.3 - Modulok

A csomag függvénykönyvtár biztosítja modulok betöltését és létrehozását a Lua-ban. Két függvénye közvetlenül a globális környezetből érhető el: a `require` és a `module`. Minden egyéb a `package` tömbben található.

`module (name [, ˇˇˇ])`

Egy modult készít. Ha a `package.loaded[name]` tömb már létezik, ez a tömb lesz a modul. Egyébként ha létezik a megadott nevű globális `t` tömb, az a tömb lesz a modul. Egyéb esetben egy üres `t` tömböt hoz létre, és a globális `name` és a `package.loaded[name]` értékeként állítja be azt. A függvény szintén létrehozza a `t._NAME` mezőt a megadott névvel, a `t._M` mezőt a modullal (a `t`-vel magával), és a `t._PACKAGE` mezőt a csomag nevével (a teljes modulnévvel, mínusz az utolsó komponens, lásd lentebb). Végül, a `module` beállítja a `t`-t, mint a jelenlegi függvény új környezetét, és a `package.loaded[name]` új értékeként, így a `require` visszatérési értéke `t`.

Ha a `name` egy vegyes név (tehát egyenként pontokkal elválasztva), a `module` tömböket készít (vagy újra felhasznál, ha már léteznek) az egyes komponensek számára. Például, ha a `name` értéke `a.b.c`, akkkor a `module` a modult a globális `a` tömb `b` mezőjének `c` mezőjében tárolja.

A függvény kaphat *options* argumentumokat a modul neve után, ahol minden egyes opció egy függvény, amelyet a modul használ fel.

`require (modname)`

Betölti a megadott modult. A függvény először a `package.loaded` tömböt vizsgálja meg, hogy a megadott `modname` modul be van -e már töltve.Ha igen, a `require` a `package.loaded[modname]` értékével tér vissza. Egyébként megpróbál egy *betöltőt* keresni a modulhoz.

Ha talál ilyet, a `require` végrehajtja a `package.preload[modname]` lekérést. Ha ennek van értéke, ez lesz a betöltő (ami egy függvény). Egyébként a `require` egy Lua betöltőt keres a `package.path` változóban megadott útvonalakon. Ha ez sikertelen, akkor egy C betöltőt, a `package.cpath` változóban megadott útvonalakon. Ha ez is sikertelen, akkor egy *minden-az-egyben* betöltőt próbál meg (lásd lentebb).

Egy C függvénykönyvtár betöltésekor a `require` egy dinamikusan kapcsolódó készséget használ az alkalmazás és a függvénykönyvtár összekapcsolására. Ezután megpróbál egy C függvényt keresni az eljáráskönyvtáron belül, amely betöltőként használható. Ennek a C függvénynek a neve "`luaopen_`" karakterlánccal kezdődik, és a modul nevével folytatódik, ahol minden pontot aláhúzás helyettesít. Ezen felül, ha a modul nevében van egy kötőjel, az előtte lévő rész (a kötőjellel együtt) el lesz távolítva. Így például az `a.v1-b.c` név esetén a függvény neve `luaopen_b_c` lesz.

Ha a `require` nem talál sem Lua, sem C függvénykönyvtárat a modulnak, akkor a *minden-az-egyben* betöltőt hívja meg. Ez a C útvonalat vizsgálja át a megadott modul tőalakú nevű eljáráskönyvtáráért. Így például az `a.b.c` név esetén a require az `a` nevű C függvénykönyvtárat fogja keresni. Ha talál ilyet, ebben keres egy almodult, esetünkban ez a `luaopen_a_b_c`. Ezzel a lehetőséggel a csomagkezelő több C almodult is egy függvénykönyvtárba csomagolhat, amelyekben megmarad az eredeti megnyitó függvény.

Ha van betöltő, a `require` meghívja az a `modname` argumentummal. Ha a betöltőnek van visszatérési értéke, a `require` a `package.loaded[modname]` értékeként állítja be azt. Ha nincs, és a `package.loaded[modname]` mezőnek nincs értéke, akkor a `require` a **true** értéket rendeli ehhez a bejegyzéshez. Bármilyen más esetben a `require` visszatérési értéke a `package.loaded[modname]` végső értéke.

Ha a modul futtatása közben hiba lép fel, vagy nem található betöltő a modulhoz, a `require` hibát jelez.

`package.cpath`

A `require` által használt C betöltő útvonalát határozza meg.

A Lua a `package.cpath` értékét a `package.path`-hoz hasonlóan határozza meg, a `LUA_CPATH` környezeti változó használatával (valamint a másik, `luaconf.h` fájlban megadott útvonal segítségével).

`package.loaded`

A `require` által használt tömb, a már betöltött modulok ellenőrzésére. A `modname` modul betöltésekor, ha a `package.loaded[modname]` nem hamis, a `require` visszatérési értéke az ott tárolt érték.

`package.loadlib (libname, funcname)`

Dinamikusan összekapcsolja a fő programot a `libname` nevű C függvénykönyvtárral. Ebben a függvénykönyvtárban megkeresi a `funcname` függvényt, és ezzel tér vissza, C függvényként. (Tehát a `funcname` függvénynek követnie kell a protokollt (lásd `lua_CFunction`)).

Ez egy alacsonyszintű függvény, amely teljesen eltér a csomag- és modulrendszertől. A `require`-től eltérően, nem hajt végre útvonal-kereséseket és nem vesz fel automatikusan kiterjesztéseket. A `libname` a C eljáráskönyvtár teljes neve kell hogy legyen, beleértve az útvonalat és a kiterjesztést is. A `funcname` a C eljáráskönyvtár által exportált pontos név (ez a használt C fordítótól és a linkelőtől is függ).

Ez a függvény ANSI C-ben nincs támogatva. Mint ilyen, csak néhány platformon elérhető (Windows, Linux, Mac OS X, Solaris, BSD, valamint az olyan Unix rendszerek, amelyek támogatják a `dlfcn` szabványt).

`package.path`

A `require` által használt Lua betöltő útvonalát határozza meg.

Indításkor a Lua a `LUA_PATH` környezeti változó értékét rendeli hozzá, vagy a `luaconf.h` fájlban megadott alapértelmezett útvonalat, ha a környezeti változó nincs megadva. A környezeti változó értékében minden egyes "`;;`" kifejezés az alapértelmezett útvonalra lesz cserélve.

Az útvonal *minták* sorozata, pontosvesszővel elválasztva. A `require` minden egyes mintában a kérdőjeleket a `filename`-ra cseréli le, ami a `modname` értéke, ahol minden egyes pont a "könyvtár elválasztóra" lesz cserélve (mint pl. a "`/`" Unix rendszereken); majd megpróbálja betölteni az így kapott fájlnevet. Így tehát, ha a Lua útvonal értéke a következő:

```
"./?.lua;./?.lc;/usr/local/?/init.lua"
```

akkor a `foo` modul Lua betöltője a következő fájlokat próbálja meg betölteni: `./foo.lua`, `./foo.lc`, és `/usr/local/foo/init.lua`, ebben a sorrendben.

`package.preload`

Egy tömb, amely a modulok betöltőit tartalmazza. (lásd `require`).

`package.seeall (module)`

A `module` metatömbjét állítja be, amelyben az `__index` mező a globális környezetre hivatkozik, így ez a modul megörökli a globális környezet változóit is. A `module` függvény beállításaként használható.

5.4 - Karakterlánc kezelés

Ez a függvénykönyvtár karakterláncok kezeléséhez biztosít általános függvényeket, úgy mint kisebb karakterláncok megtalálása és kivonása, valamint minta-illeszkedés. A Lua nyelvben a karakterláncok indexelése 1-től kezdődik (és nem 0-tól, mint C-ben). Az indexek lehetnek negatívak, ekkor hátulról indexelésként a karakterlánc végétől számítva lesz feldolgozva. Így tehát az utolsó karakter pozíciója -1, és így tovább.

A karakterlánc eljáráskönyvtár összes függvénye a `string` tömb mezőiként érhetőek el. A karakterláncoknak van metatömbje is, amelynek az `__index` mezője a `string` tömbre mutat. Így a karakterlánc-függvények objektum-orientált formában használhatóak. Például a `string.byte(s, i)` kifejezés felírható `s:byte(i)` formában is.

`string.byte (s [, i [, j]])`

Visszatérési értékei az `s[i]`, `s[i+1]`, ˇˇˇ, `s[j]` karakterek belső numerikus kódjai. Az `i` alapértelmezett értéke 1; a `j` értéke pedig `i`.

Jegyezzük meg, hogy a belső numerikus kódok nem feltétlenül ugyanazok az egyes platformokon.

`string.char (ˇˇˇ)`

Nulla vagy több egész típusú argumentumot kap. Az argumentumok számával megegyező hosszúságú karakterlánccal tér vissza, ahol minden egyes karakter belső numerikus kódja megegyezik a hozzá tartozó argumentummal.

Jegyezzük meg, hogy a belső numerikus kódok nem feltétlenül ugyanazok az egyes platformokon.

`string.dump (function)`

Visszatérési értéke a megadott függvény bináris karakterlánc formában, így később a `loadstring` hívás ezen a karakterláncon a függvény másolatát adja. A `function` csak felsőértékek nélküli Lua függvény lehet.

`string.find (s, pattern [, init [, plain]])`

Az `s` karakterláncban megkeresi a `pattern` első egyezését. Ha talál ilyet, a `find` az `s` indexeivel tér vissza, ahol az egyezés kezdődik, illetve végződik; egyéb esetben a visszatérési értéke **nil**. A harmadik, opcionális szám alakú `init` argumentum adja meg, hol kezdődjön a keresés; az alapértelmezett értéke 1, és negatív is lehet. A negyedik, opcionális `plain` argumentum **true** értéke kikapcsolja a minta illeszkedési lehetőségeket, azaz a függvény egy sima "al-karakterlánc keresést" hajt végre, ahol a `pattern`-ban egy karakter sem minősül "mágikusnak". Ha a `plain` meg van adva, akkor az `init`-nek is értéket kell adni.

Ha a minta értékeket is kiemel, akkor ezek is visszatérnek, a két index után.

---

string.format (formatstring, ˇˇˇ)

Visszatérési értéke az első (karakterlánc) argumentumban megadott formába átalakított változó számú argumentumok. A formátum a szabványos C függvények `printf` családjának szabályait követi. Az egyetlen különbség, hogy a szabályozók/módosítók (`*`, `l`, `L`, `n`, `p`, és `h`) nem támogatottak, valamint van egy extra opció, a `q`. Ez használható arra, hogy a karakterláncok biztonságosan visszaolvashatóak legyenek a Lua feldolgozó számára: a karakterláncok dupla idézőjelek közé kerülnek, és a karakterláncban minden dupla idézőjel, újsor karakter, beágyazott nulla és backslash karakterek biztonságos formátumúra lesznek alakítva. Például a

```
string.format('%q', 'a string with "quotes" and \n new line')
```

hívás a következő karakterláncot eredményezi:

```
"a string with \"quotes\" and \
new line"
```

A `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, és `x` opciók mind számot várnak paraméterként, míg a `q` és `s` karakterláncokat.

A függvény nem fogad el olyan karakterláncot, amely beágyazott zérót tartalmaz.

---

string.gmatch (s, pattern)

Visszatérési értéke egy olyan iterációs függvény, amely minden hívásakor az `s` karakterlánc egy újabb, `pattern` mintára illeszkedő részével tér vissza.

Ha a `pattern` nem emel ki értékeket, akkor minden híváskor az egész egyezés lesz az eredmény.

Például a következő ciklus:

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
    print(w)
end
```

végigiterál az `s` karakterlánc szavain, majd soronként kiírja azokat. A következő példa a megadott karakterlánc összes `key=value` párját egy tömbbe gyűjti össze:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
    t[k] = v
end
```

---

string.gsub (s, pattern, repl [, n])

Visszatérési értéke az `s` karakterlánc másolata, amelyben a `pattern` minta összes illeszkedése le lett cserélve a megadott `repl` értékre, ami lehet karakterlánc, tömb vagy függvény. A `gsub` második visszatérési értéke az elvégzett cserék száma.

Ha a `repl` karakterlánc, akkor annak értéke lesz a cserekifejezés. A `%` karakter vezérlőkarakterként működik: a `repl`-ben szereplő minden `%`*n* formátumú kifejezés, ahol *n* 1 és 9 közötti érték, az *n*. számú kiemelt értéket adja vissza (lásd lentebb). A `%0` kifejezés az egész egyezést adja vissza. A `%%` kifejezés egy `%` karaktert jelent.

Ha a `repl` tömb, akkor a tömb minden egyezéskor le lesz kérdezve, ahol a minta lesz a kulcs; ha a minta nem emel ki értéket, akkor az egész egyezés lesz a kulcs.

Ha a `repl` függvény, akkor ez minden egyezéskor meg lesz hívva, argumentumai pedig sorban az illeszkedő al-karakterláncok lesznek. Ha a minta nem emel ki értéket, akkor az egész egyezés lesz az egyetlen argumentum.

Ha a tömb-lekérdezés értéke vagy a függvény visszatérési értéke karakterlánc vagy szám, az lesz a cserekifejezés, egyébként, ha az **false** vagy **nil**, nem történik csere (azaz, az eredeti egyezés a karakterláncban marad).

Az utolsó, opcionális `n` paraméter a maximális cserék számát szabályozza. Például, ha `n` értéke 1, csak a `pattern` minta első illeszkedése lesz lecserélve.

Következzen néhány példa:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
    return loadstring(s)()
end)

--> x="4+5 = 9"

local t = {name="lua", version="5.1"}

x = string.gsub("$name%-$version.tar.gz", "%$(%w+)", t)

--> x="lua-5.1.tar.gz"
```

---

string.len (s)

Karakterlánc paramétert kap, és a hosszával tér vissza. Az `""` üres karakterlánc hossza 0. A beágyazott zérók is beleszámítanak ebbe, így a `"a\000bc\000"` karakterlánc hossza 5.

`string.lower (s)`

Karakterláncot kap, és a másolatával tér vissza, amelyben az összes nagybetű kicsire lesz cserélve. Minden egyéb karakter változatlan marad. A nagybetűk definíciója a nyelvi beállításoktól függően eltérő lehet.

`string.match (s, pattern [, init])`

Az `s` karakterláncban megkeresi a `pattern` minta első *egyezését*. Ha talál ilyet, akkor a `match` a talált részekkel tér vissza; egyéb esetben a visszatérési érték **nil**. Ha a `pattern` minta nem emel ki értéket, az egész egyezés visszatér. A harmadik, opcionális szám argumentum, az `init` adja meg, hol kezdődjön a keresés; az alapértelmezett értéke 1, és negatív is lehet.

`string.rep (s, n)`

Visszatérési értéke a megadott `s` karakterlánc `n` számú összefűzött másolata.

`string.reverse (s)`

Visszatérési értéke az `s` karakterlánc fordítottja.

`string.sub (s, i [, j])`

Visszatérési értéke az `s` karakterlánc része, amely `i` indexnél kezdődik és `j`-ig tart; `i` és `j` is lehet negatív. Ha a `j` nincs megadva, -1-ként lesz értelmezve (ami megegyezik a karakterlánc hosszával). Általánosságban, a `string.sub(s,1,j)` hívás az `s` `j` hosszúságú előtagjával, míg a `string.sub(s, -i)` hívás az `s` `i` hosszúságú utótagjával tér vissza.

`string.upper (s)`

Karakterláncot kap, és a másolatával tér vissza, amelyben az összes kisbetű nagyra lesz cserélve. Minden egyéb karakter változatlan marad. A kisbetűk definíciója a nyelvi beállításoktól függően eltérő lehet.

5.4.1 - Minták

Karakterosztályok:

Egy *karakterosztály* karakterek sorozatát jelenti. A következő kombinációk karakterosztályokat írnak le:

- ***x*:** (ahol *x* nem *mágikus karakter:* `^$()%.[]*+-?`) az *x* karaktert jelenti.
- **`.`:** (egy pont) minden karakternek megfelel.
- **`%a`:** minden betűnek megfelel.
- **`%c`:** minden vezérlőkarakternek megfelel.
- **`%d`:** minden számjegynek megfelel.
- **`%l`:** minden kisbetűs betűnek megfelel.
- **`%p`:** minden írásjelnek megfelel.
- **`%s`:** minden szóköz karakternek megfelel.
- **`%u`:** minden nagybetűs betűnek megfelel.
- **`%w`:** minden alfanumerikus karakternek megfelel.
- **`%x`:** minden hexadecimális számjegynek megfelel.
- **`%z`:** 0-szor előforduló karakternek felel meg.
- **`%`*x*:** (ahol *x* bármilyen nem alfanumerikus karatker) az *x* karakternek felel meg. Ez a mágikus karakterek semlegesítésének alapértelmezett módja. Bármilyen írásjelet (még a nem mágikusakat is) megelőzhet a semlegesítő '`%`' karakter; ebben az esetben a mintában saját magát fogja jelenteni.
- **`[`*sorozat*`]`:** A megadott sorozat összes karakterének uniójával képez egy osztályt. Karakterek tartománya megadható a tartomány záró karakterének '`-`' jellel történő elválasztásával. Minden `%`*x* formátumú fentebb leírt osztály használható a *sorozat* elemeként. Az összes többi karakter saját magát jelenti. Például, a `[%w_]` (vagy a `[_%w]`) az összes alfanumerikus karaktert és az aláhúzást fogja jelenteni, a `[0-7]` oktális számjegyeket, a `[0-7%l%-]` az oktális számjegyeket, kisbetűs betűket, valamint a '`-`' karaktert jelenti.

  A tartományok és az osztályok között nincsen együttműködés, tehát a `[%a-z]` vagy `[a-%%]` alakú minta nem értelmezhető.

- **`[^`*sorozat*`]`:** A megadott sorozat komplementerét (kiegészítését) jelenti, ahol a *sorozat* a fentebb leírtak szerint van értelmezve.

Az összes osztályt, amit egy betű jelöl (`%a`, `%c`, stb.), a nagybetűs alakja az osztály komplemeterét (kiegészítését) jelenti. Például, az `%S` minden nem-szóköz karakternek megfelel.

A betű, szóköz és egyéb karaktercsoportok definíciója mindig a nyelvi beállításoktól függ. A gyakorlatban az `[a-z]` osztály nem minden esetben ugyanaz, mint a `%l`.

Minta elem:

Egy *minta elem* lehet

- egy karakterosztály, amely az osztály bármely egyetlen karakterével megegyezik;
- egy karakterosztály, amit egy '`*`' jel követ, amely az osztály karaktereinek 0 vagy több ismétlődésével egyezik meg. Ez az ismétlődés mindig a lehető leghosszabb sorozattal fog megegyezni;
- egy karakterosztály, amit egy '`+`' jel követ, amely az osztály karaktereinek 1 vagy több ismétlődésével egyezik meg. Ez az ismétlődés mindig a lehető leghosszabb sorozattal fog megegyezni.
- egy karakterosztály, amit egy '`-`' jel követ, amely az osztály karaktereinek szintén 0 vagy több ismétlődésével egyezik meg. Viszont a '`*`' jellel ellentétben, ez az elemismétlődés mindig a lehető *legrövidebb* sorozattal fog megegyezni;
- egy karakterosztály, amit egy '`?`' jel követ, amely az osztály karaktereinek 0 vagy 1 ismétlődésével egyezik meg;
- `%`*n*, ahol *n* 1 és 9 közötti szám; az ilyen elemek az *n*-dik számú kiemelt értéket adja vissza (lásd lentebb);
- `%b`*xy*, ahol *x* és *y* két különböző karakter; az ilyen elemek olyan karakterláncokkal egyeznek meg, amelyek *x*-el kezdődnek és *y*-al végződnek, és ahol *x* és *y kiegyensúlyozott*. Ez azt jelenti, hogy a karakterlánc balról jobbra lesz olvasva, és az *x*-ek esetén *+1*-el növekszik, *y*-ok esetén *-1*-el csökken egy számláló, a végső *y* az első olyan *y*, ahol a számláló eléri a 0-t. Például a `%b()` elem kiegyenlített zárójelekkel egyezik meg.

Minta:

A *minta* mintaelemek sorozata. A '`^`' jel a minta elején megszabja, hogy a minta csak a keresendő karakterlánc elejét, míg a '`$`' jel a minta végén azt, hogy a keresendő karakterlánc végét vizsgálja. Más pozíciókban a '`^`' és '`$`' jeleknek nincs speciális jelentésük, és saját magukat reprezentálják.

Kiemelt értékek:

A minták tartalmazhatnak olyan almintákat, amelyek zárójelek között vannak; ezek *kiemelt értékeket* jelentenek. Ha egy egyezés sikeres, a keresendő karakterlánc egyező al-karakterláncai eltároldónak (*kiemelődnek*) későbbi használatra. A kiemelt értékek a nyitó zárójelek sorrendje szerint vannak számolva. Például a `"(a*(.)%w(%s*))"` minta esetén a karakterláncban a `"a*(.)%w(%s*)"` mintára egyező rész az első kiemelt érték (és a száma 1); A "."-ra illeszkedő karakter 2-es számmal lesz kiemelve, és a "`%s*`"-ra illeszkedő rész a 3-as.

Egy speciális eset, amikor üres kiemelés `()` történik, ez esetben a visszatérési érték a jelenlegi karakterlánbeli pozíció (egy szám). Például, az `"()aa()"` minta a `"flaaap"` karakterláncon két eredményt hoz: 3 és 5.

A minta nem tartalmazhat beágyazott zérókat, helyette a `%z` használható.

5.5 - Tömb kezelés

Ez a függvénykönyvtár tömbök kezeléséhez biztosít általános függvényeket. A függvényei a `table` tömb mezőiként érhetőek el.

Ennek az eljáráskönyvtárnak a függvényei csak rendezett tömbökön és listákon működik. Ezeknél a függvényeknél, ha a tömb "hosszáról" van szó, akkor a hossz operátor eredménye értendő ezalatt.

---

table.concat (table [, sep [, i [, j]]])

Visszatérési értéke `table[i]..sep..table[i+1] ˇˇˇ sep..table[j]`. A `sep` alapértelemezett értéke az üres karakterlánc, az `i` értéke 1, és a `j` alapértelmezett értéke a tömb hossza. Ha az `i` értéke nagyobb, mint `j`, üres karakterlánccal tér vissza.

---

table.insert (table, [pos,] value)

A megadott `value` értéket elhelyezi a `table` tömbben, `pos` pozícióban, a többi elemet felfelé szabad helyre csúsztatva, ha szükséges. A `pos` alapértelmezett értéke `n+1`, ahol `n` értéke a tömb hossza (lásd §2.5.5), így a `table.insert(t,x)` hívás az `x` értéket a `t` tömb végére helyezi.

---

table.maxn (table)

Visszatérési értéke a tömb legmagasabb numerikus indexe, vagy zéró, ha a tömbnek nincsenek pozitív numerikus indexei. (a függvény ehhez lineáris bejárást hajt végre az egész táblán.)

---

table.remove (table [, pos])

A `table` tömbből eltávolítja a `pos` pozícióban lévő elemet, a többi elemet lefelé szabad helyre csúsztatva, ha szükséges. Visszatérési értéke az eltávolított elem értéke. A `pos` alapértelmezett értéke `n`, ahol `n` a tömb hossza, így a `table.remove(t)` hívás a `t` tömb utolsó elemét távolítja el.

---

table.sort (table [, comp])

A tömböt a megadott sorrend szerint rendezi, *belsőleg*, `table[1]`-től `table[n]`-ig, ahol `n` a tömb hossza. Ha a `comp` adott, akkor annak egy függvénynek kell lennie, ami két táblaelemet kap argumentumként, és true értékkel tér vissza, ha az első kisebb, mint

a második (így tehát a `not comp(a[i+1],a[i])` igazzá válik a rendezés után). Ha a `comp` nincs megadva, akkor az alapértelmezett `<` Lua operátor lesz használatban.

A rendezési algoritmus nem állandó, tehát még az egyenlőnek értékelt elemek relatív pozíciója is megváltozhat a rendezés közben.

5.6 - Matematikai függvények

Ez a függvénykönyvtár az alapértelmezett C matematikai függvénykönyvtárat valósítja meg. A függvényei a `math` tábla mezőiként érhetőek el.

`math.abs (x)`

Az `x` abszolútértékével tér vissza.

`math.acos (x)`

Az `x` arc cosinus-ával tér vissza (radiánokban).

`math.asin (x)`

Az `x` arc sinus-ával tér vissza (radiánokban).

`math.atan (x)`

Az `x` arc tangensével tér vissza (radiánokban).

`math.atan2 (x, y)`

Visszatérési értéke az `x/y` arc tangense (radiánokban), de mindkét paraméter előjelét használja az eredmény quadránsának meghatározásához. (Szintén megfelelően kezeli azt, ha az `y` értéke zéró.)

`math.ceil (x)`

Visszatérési értéke a legkisebb egész, amely nagyobb, vagy egyenlő, mint `x`.

---

math.cos (x)

Az `x` cosinus-ával tér vissza (a fok radiánban van megadva).

---

math.cosh (x)

Az `x` hiperbolikus cosinus-ával tér vissza.

---

math.deg (x)

A radiánokban megadott `x` értékével tér vissza fokokban.

---

math.exp (x)

Visszatérési értéke e^x.

---

math.floor (x)

Visszatérési értéke a legnagyobb egész, amely kisebb, vagy egyenlő, mint `x`.

---

math.fmod (x, y)

Visszatérési értéke az `x` `y`-al való osztásából származó maradék.

---

math.frexp (x)

Visszatérési értéke `m` és `e` a $x = m2^e$ kifejezésből, `e` egy egész, és az `m` abszolútértéke a *[0.5, 1)* tartományban (vagy zéró, ha `x` zéró).

---

math.huge

A `HUGE_VAL` értéke, egy érték, amely nagyobb vagy egyenlő, mint bármilyen más numerikus érték.

---

math.ldexp (m, e)

Visszatérési értéke $m2^e$ (`e` egész típusú).

`math.log (x)`

Visszatérési értéke `x` természetes logaritmusa.

`math.log10 (x)`

Visszatérési értke `x` 10-es alapú logaritmusa.

`math.max (x, ˇˇˇ)`

Az argumentumok legmagasabb értékével tér vissza.

`math.min (x, ˇˇˇ)`

Az argumentumok legalacsonyabb értékével tér vissza.

`math.modf (x)`

Két értékkel tér vissza, `x` egész és tört értékével.

`math.pi`

A PI értéke.

`math.pow (x, y)`

Visszatérési értéke x^y. (Az érték kiszámítására az `x^y` kifejezés is használható.)

`math.rad (x)`

Visszatérési értéke a fokokban megadott `x` szög értéke radiánban.

---

math.random ([m [, n]])

Ez a függvény egy látszólagos véletlenszám generátort valósít meg a `rand` ANSI C függvény segítségével. (A statisztikai tulajdonságaira nincs garancia.)

Argumentumok nélküli használat esetén a visszatérési érték egy véletlenszerű valós szám a *[0,1)* tartományból. Ha a numerikus `m` argumentum adott, a `math.random` visszatérési értéke egy véletlenszerű egész szám az *[1, m]* tartományból. Ha mindkét numerikus paraméter, `m` és `n` is adott, a `math.random` visszatérési értéke egy véletlenszerű egész szám az *[m, n]* tartományból.

---

math.randomseed (x)

Az `x`-et a pszeudo-véletlenszám generátor szórásaként állítja be: egyenlő szórás egyenlő sorozatú számokat állít elő.

---

math.sin (x)

Visszatérési értéke `x` sinus-a (a fok radiánban van megadva).

---

math.sinh (x)

Visszatérési értéke `x` hiperbolikus sinus-a.

---

math.sqrt (x)

Visszatérési értéke `x` négyzetgyöke. (Az érték kiszámítására az `x^0.5` kifejezés is használható.)

---

math.tan (x)

Visszatérési értéke `x` tangense (a fok radiánban van megadva).

---

math.tanh (x)

Visszatérési értéke `x` hiperbolikus tangense.

5.7 - Bemeneti és kimeneti lehetőségek

Az I/O függvénykönyvtár kétfajta fájlkezelést is biztosít. Az első implicit fájlleírókat használ, így beállítható alapértelmezett be- illetve kimeneti fájl; ezután minden művelet ezeken az alapértelmezett fájlokon lesz végrehajtva. A másik mód explicit fájlleírókat használ.

Implicit fájlleírók esetén minden művelet az `io` tömbből érhető el. Explicit fájlleírók esetén az `io.open` művelet egy fájlleíróval tér vissza, és a később minden művelet ennek a fájlleírónak eljárásaként érhető el.

Az `io` tömb három előre definiált fájlleírót is tartalmaz, a megszokott C jelentésük szerint: `io.stdin`, `io.stdout`, és `io.stderr`.

Ha nincs másként feltüntetve, minden I/O művelet **nil** értékkel tér vissza hiba esetén (valamint egy hibaüzenettel, második eredményként). Minden **nil**-től különböző érték sikeres végrehajtást jelent.

`io.close ([file])`

Megegyezik a `file:close()` hívással. `file` argumentum nélkül az alapértelmezett kimeneti fájlt zárja be.

`io.flush ()`

Megegyezik a `file:flush` művelettel, de az alapértelmezett kimeneti fájlon hajtja végre.

`io.input ([file])`

Ha a fájlnév meg van adva, megnyitja a nevezett fájl (szöveges módban), és az alapértelmezett bemeneti fájlként állítja be azt. Ha egy fájlkezelővel van meghívva, akkor egyszerűen ezt a kezelőt állítja be az alapértelmezett bemeneti fájlnak. Ha paraméterek nélkül van meghívva, visszatérési értéke a jelenlegi alapértelmezett bemeneti fájl.

Hiba esetén a függvény hibát ér el, és nem hibakóddal tér vissza.

`io.lines ([filename])`

Megnyitja a megadott fájlt olvasási módban, és annak iterációs függvényével tér vissza, ami minden híváskor a fájl egy újabb sorával tér vissza. Így a következő szerkezet

```
for line in io.lines(filename) do body end
```

végigiterál a fájl sorain. Amikor az iterátor függvény eléri a fájl végét, **nil** értékkel tér vissza (hogy befejezze a ciklust) és automatikusan bezárja a fájlt.

Az `io.lines()` hívás (fájlnév nélkül) megegyezik a `io.input():lines()` hívással, tehát az alapértelmezett bemeneti fájl sorain léptet végig. Ebben az esetben nem zárja be a fájlt, amikor a ciklus véget ér.

`io.open (filename [, mode])`

A függvény megnyit egy fájlt, a `mode` karakterlánc által megszabott módban. Visszatérési értéke egy új fájlkezelő, vagy hiba esetén **nil** és egy hibaüzenet.

A `mode` karakterlánc a következő értékeket veheti fel:

- **"r":** olvasási mód (alapértelmezett);
- **"w":** írási mód;
- **"a":** hozzáfűzési mód;
- **"r+":** frissítési mód, minden előzetes adat megmarad;
- **"w+":** frissítési mód, minden előzetes adat törölve lesz;
- **"a+":** frissítési mód, minden előzetes adat megmarad, írni csak a fájl végére lehet.

A `mode` karakterlánc végződhet '`b`' karakterrel, amely néhány operációs rendszeren szükséges, hogy a fájl bináris módban nyíljon meg. Ez a karakterlánc ugyanaz, mint a szabványos C függvényben használt `fopen`.

`io.output ([file])`

Az `io.input`-hoz hasonlít, de az alapértelmezett kimeneti fájlon hajtja végre a műveleteket.

`io.popen ([prog [, mode]])`

Elindítja a `prog` programot külön folyamatként, és egy fájlkezelővel tér vissza, amivel a programból adat olvasható ki (ha a `mode` értéke `"r"`, ami az alapértelmezett érték) vagy adat írható a program számára (ha a `mode` értéke `"w"`).

Ez a függvény a rendszertől függ, és nem érhető el minden platformon.

`io.read (ˇˇˇ)`

Megegyezik az `io.input():read` hívással.

---

io.tmpfile ()

Visszatérési értéke egy ideiglenes fájl kezelője. Ez a fájl frissítési módban lesz megnyitva, és automatikusan törölve lesz, amikor a program futása befejeződik.

---

io.type (obj)

Ellenőrzi, hogy a megadott `obj` érvényes fájlkezelő -e. Visszatérési értéke a `"file"` karakterlánc, ha az `obj` egy nyitott fájl, `"closed file"`, ha az `obj` egy bezárt fájl, és **nil**, ha az `obj` nem fájlkezelő.

---

io.write (˘˘˘)

Megegyezik az `io.output():write` hívással.

---

file:close ()

Bezárja a megadott `file` fájlkezelőt. Megjegyzendő, hogy a fájlok automatikusan bezáródnak, amikor a kezelőiket összegyűjti a szemétgyűjtő, de ez nem meghatározható időbe telik.

---

file:flush ()

A `file`-ba írt minden adatot elment.

---

file:lines ()

Egy iterációs függvényével tér vissza, ami minden híváskor a fájl egy újabb sorával tér vissza. Így a következő szerkezet

```
for line in file:lines() do body end
```

végigiterál a fájl sorain. (Az `io.lines`-tól eltérően, ez a függvény nem zárja be a fájlt a ciklus végén.)

---

file:read (˘˘˘)

A megadott formátumnak megfelelően olvas a `file` fájlból. Minden formátum esetén az olvasott karakterlánc (vagy szám) lesz a visszatérési érték, vagy **nil**, ha a megadott formátumban nem olvasható ki adat. Ha formátum nélkül van meghívva, az alapértelmezett formátum lesz használva, ami a következő egész sort olvassa (lásd lentebb).

A következő formátumok használhatóak:

- **"*n":** egy számot olvas; ez az egyetlen formátum, ami számértékkel tér vissza karakterlánc helyett.
- **"*a":** az egész fájlt beolvassa a jelenlegi pozíciótól. A fájl végén üres karakterlánccal tér vissza.
- **"*l":** a következő sort olvassa (a sorvéget átugorva), a fájl végén a visszatérési érték **nil**. Ez az alapértelmezett érték.
- ***szám*:** a megadott számú hosszúságú karakterláncot olvassa, a fájl végén a visszatérési érték **nil**. Ha a szám 0, nem olvas semmit, és a visszatérési érték egy üres karakterlánc, vagy a fájl végén **nil**.

---

file:seek ([whence] [, offset])

Lekéri a fájl pozícióját, a fájl elejétől mérve, vagy beállítja a megadott `offset` pozícióra, a megadott `whence` karakterlánctól függően, ami a következő értékek egyike lehet:

- **"set":** az alap a 0 pozíció (a fájl kezdete);
- **"cur":** az alap a jelenlegi pozíció;
- **"end":** az alap a fájl vége;

Siker esetén a `seek` függvény visszatérési értéke a végső pozíció, amelynek mértéke a fájl elejétől mérődik, byteokban. Ha a függvény sikertelenül ér véget, a visszatérési értéke **nil**, valamint egy hibaüzenet, ami a hibáról ad további információt.

A `whence` alapértelmezett értéke `"cur"`, az `offset` értéke pedig 0. Így a `file:seek()` hívás visszatérési értéke a jelenlegi fájl pozíciója, változtatás nélkül; a `file:seek("set")` hívás a fájl elejére állítja a pozíciót (és 0-val tér vissza); valamint a `file:seek("end")` hívás a fájl végére állítja a pozíciót, és a fájl méretével tér vissza.

---

file:setvbuf (mode [, size])

A kimeneti fájl bufferének módját állítja be. Háromféle mód elérhető:

- **"no":** nincs bufferelés; minden kimeneti művelet eredménye azonnal megjelenik.

- **"full":** teljes bufferelés; a kimeneti műveletek csak akkor hajtódnak végre, amikor a buffer megtelik (vagy a fájl explicit módon *ki lesz íratva* a fájlba (lásd `io.flush`)).
- **"line":** soros bufferelés; a kiemenet egy újsorig lesz bufferelve, vagy van valamilyen bemenet van speciális fájlokból (például egy terminális eszközről).

Az utolsó két esetben a `size` argumentum adja meg a buffer méretét, byteokban. Az alapértelmezett értéke egy elfogadható méret.

`file:write (ˇˇˇ)`

A megadott argumentumok értékeit a `file` fájlba írja. Az argumentumok karakterláncok és számok lehetnek. A többi érték kiírásához a `tostring` vagy `string.format` használandó a `write` hívás előtt.

5.8 - Operációs rendszer szolgáltatások

Ez a függvénykönyvtár a `os` tömbön kesztül érhető el.

`os.clock ()`

Visszatérési értéke a program által használt CPU idő megbecsült értéke.

`os.date ([format [, time]])`

Visszatérési értéke a dátumot és időt tartalmazó karakterlánc vagy tömb, a megadott `format` formátumban.

Ha a `time` argumentum meg van adva, ez az időpont lesz formázva (lásd az `os.time` függvényt ennek az értéknek a leírásáért). Egyébként a `date` a jelenlegi időt formázza.

Ha a `format` '`!`' jellel kezdődik, akkor a dátum az egységes egyetemes idő szerint lesz formázva. Ha ezután az opcionális karakter után a `format` értéke `*t`, akkor a `date` visszatérési értéke egy tömb, a következő mezőkkel: `year` (négy számjegy), `month` (1--12), `day` (1--31), `hour` (0--23), `min` (0--59), `sec` (0--61), `wday` (a hét napja, a vasárnap értéke 0), `yday` (az év napja), és `isdst` (szökőév jelző, boolean).

Ha a `format` értéke nem `*t`, akkor a `date` visszatérési értéke a jelenlegi dátum karakterláncként, melynek formázásának szabályai megegyeznek a `strftime` C függvényével.

Argumentumok nélküli hívásakor a visszatérési értéke a dátum és az idő elfogadható formátuma, amely a rendszertől és a jelenlegi nyelvi környezettől függ (tehát az `os.date()` hívás ugyanaz, mint az `os.date("%c")`).

`os.difftime (t2, t1)`

Visszatérési értéke a `t1` és `t2` idők közötti különbség, másodpercekben. POSIX, Windows, és néhány egyéb rendszeren ez ugyanaz, mint a `t2-t1`.

`os.execute ([command])`

Ez a függvény megegyezik a `system` C függvénnyel. A `command` parancs lesz végrehajtva az operációs rendszer parancsértelmezője által. Visszatérési értéke egy állapotkód, ami rendszerfüggő. Ha a `command` nincs megadva, visszatérési értéke egy nem-zéró érték, ha a parancsértelmező elérhető, ellenkező esetben 0.

`os.exit ([code])`

Meghívja az `exit` C függvényt, az opcionális `code` kóddal, a gazdaprogram befejezéséhez. A `code` alapértelmezett értéke a siker kódja.

`os.getenv (varname)`

Visszatérési értéke a folyamat `varname` nevű környezeti változójának értéke, vagy **nil**, ha a változó nincs deklarálva.

`os.remove (filename)`

A megadott nevű fájlt vagy könyvtárat törli. A törlendő könyvtár csak üres lehet. Ha a függvény sikertelen, visszatérési értéke **nil**, valamint egy hibaüzenet, amely a hibát írja le.

`os.rename (oldname, newname)`

Átnevezi a megadott `oldname` nevű fájlt vagy könyvtárat `newname` nevűre. Ha a függvény sikertelen, visszatérési értéke **nil**, valamint egy hibaüzenet, amely a hibát írja le.

---

os.setlocale (locale [, category])

A program jelenlegi nyelvi környezetét állítja be. A `locale` egy karakterlánc, ami megadja a nyelvi környezetet; a `category` egy opcionális karakterlánc, amely a megváltoztatandó kategóriát adja meg: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, vagy `"time"`; az alapértelmezett kategória az `"all"`. A függvény visszatérési értéke az új nyelvi környezet neve, vagy **nil**, ha a kérés nem végrehajtható.

Ha az első argumentum **nil**, a függvény nem tesz semmit, csak a megadott kategória jelenlegi nyelvi környezetének nevével tér vissza.

---

os.time ([table])

Visszatérési értéke a jelenlegi idő, ha argumentumok nélkül van meghívva, vagy a megadott tömb által meghatározott dátum és idő. Ennek a tömbnek a következő mezői kötelezőek: `year`, `month`, és `day`, a továbbiak csak opcionálisak: `hour`, `min`, `sec`, és `isdst` (ezeknek a mezőknek a magyarázata a `os.date` függvény leírásában szerepel).

A visszatérési érték egy szám, amelynek jelentése a rendszertől függ. POSIX, Windows, és néhány egyéb rendszeren ez a szám egy megadott kezdőponttól (a "korszaktól") eltelt másodpercek száma. Más rendszereken a jelentése nincs megadva, és csak a `date` és a `difftime` paramétereiként használható.

---

os.tmpname ()

Visszatérési értéke egy karakterlánc, amely egy ideiglenes fájl neveként használható. A fájlt explicit módon meg kell nyitni a használat előtt, valamint explicit módon el kell távolni, ha nincs rá tovább szükség.

5.9 - A hibakeresési függvénykönyvtár

Ez a függvénykönyvtár a hibakeresési felülethez biztosít elérést a Lua programok számára. Óvatosságra kell törekedni használatakor. Az itt található függvények kizárólag csak hibakeresési vagy hasonló célokra használhatóak, például sebesség optimalizálásra. Soha ne használjuk sima programozási eszközként, mivel nagyon lelassíthatja a programot. Sőt, néhány függvénye megkerüli a Lua kód bizonyos előfeltételezéseit (pl., egy függvény lokális változói kívülről nem elérhetőek, és az userdata típus metatömbjei sem változtathatóak meg a Lua kódból), és így átugorhat bizonyos biztonsági pontokon.

A függvénykönyvtár függvényei a `debug` tömb mezőiként érhetőek el. A szálakon műveleteket végző függvényeknek van egy első, opcionális paramétere, ami a szál maga, amin a művelet lesz végrehajtva. Az alapértelmezett érték mindig a jelenlegi szál.

---

debug.debug ()

Egy párbeszédes módba lép a felhasználóval, és minden egyes karakterláncot végrehajt, amit a felhasználó megad. Egyszerű parancsokkal, vagy más egyéb hibakereső lehetőségekkel, a felhasználó megvizsgálhat globális és lokális változókat, megváltoztathatja az értékeiket, kiértékelhet kifejezésket, stb. Ha a sor csak a `cont` szót tartalmazza, befejezi a függvényt, és a hívó folytatja a végrehajtást.

A `debug.debug` parancsai lexikálisan nem egymásba ágyazott a függvényekkel, így lokális változók közvetlen elérésére nincs lehetőség.

---

debug.getfenv (o)

Visszatérési értéke az `o` objektum környezete.

---

debug.gethook ([thread])

Visszatérési értéke a megadott szálhoz beállított hurok, három értékben: a jelenlegi hurokfüggvény, a jelenlegi hurok maszk, valamint a jelenlegi hurokszámláló (ami a `debug.sethook` függvénnyel állítható be).

---

debug.getinfo ([thread,] function [, what])

Visszatérési értéke egy tömb, amely a megadott függvény információit hordozza. A függvény közvetlenül is megadható, vagy számalakban is, ez esetben a megadott szál vermének szintjét jelenti ez a szám: a 0 a jelenlegi függvény (a `getinfo` maga); az 1 szint az a függvény, ami meghívta a `getinfo` függvényt, és így tovább. Ha a `function` szám, és nagyobb, mint az aktív függvények száma, a `getinfo` visszatérési értéke **nil**.

A visszatérő tömb mezői ugyanazok lehetnek, mint a `lua_getinfo` esetén, attól függően, hogy a `what` karakterlánc milyen mezőket tölt fel értékekkel. A `what` alapértelmezés szerint minden elérhető információt lekér, kivéve az érvényes sorok számának tömbjét. Ha az opcionális '`f`' paraméter is meg van adva, egy `func` nevű mező is létrejön, magávala a függvénnyel. Ha az '`L`' paraméter szerepel, akkor egy `activelines` mező jön létre, az érvényes sorok számának tömbjével.

Például, a `debug.getinfo(1,"n").name` kifejezés visszatérési értéke a jelenlegi függvény neve, ha található hozzá elfogadható név, valamint a `debug.getinfo(print)` visszatérési értéke egy tömb, amely a `print` függvény összes elérhető információját tartalmazza.

---

debug.getlocal ([thread,] level, local)

Ennek a függvénynek a visszatérési értéke a verem `level` szintjén lévő függvény `local` indexű változójának neve, valamint annak értéke. (Az első paraméter vagy lokális változó indexe 1, és így tovább, egészen az utolsó aktív lokális változóig.) A függvény visszatérési értéke **nil**, ha a megadott indexen nincs lokális változó, és hibát ér el abban az esetben, ha a `level` a tartmányon kívüli érték. (A `debug.getinfo` függvénnyel ellenőrizhető, hogy a szint érvényes -e.)

A '`(`' (nyitó zárójel) jellel kezdődő változók belső változókat írnak le (ciklus változók, ideiglenes tárolók és C függvények lokális változói).

---

debug.getmetatable (object)

Visszatérési értéke a megadott `object` metatömbje, vagy **nil**, ha nincs metatömbje.

---

debug.getregistry ()

Visszatérési értéke a registry tömb (lásd §3.5).

---

debug.getupvalue (func, up)

Visszatérési értéke a megadott `func` függvény `up` indexen lévő felsőértékének neve, valamint annak értéke. Visszatérési értéke **nil**, ha a megadott indexen nem található felsőérték.

---

debug.setfenv (object, table)

A megadott `object` környezetét cseréli le a megadott `table` tömbre. Visszatérési értéke az `object` objektum.

---

debug.sethook ([thread,] hook, mask [, count])

A megadott függvényt állítja be hurokként. A `mask` karakterlánc és a `count` szám adja meg, mikor lesz meghívva a függvény. A maszk a következő jelentéssel bíró karaktereket tartalmazhatja:

- **`"c"`:** A hurok minden alkalommal meg lesz hívva, amikor a Lua meghív egy függvényt;

- **`"r"`:** A hurok minden alkalommal meg lesz hívva, amikor a Lua visszatér egy függvényből;
- **`"l"`:** A hurok minden alkalommal meg lesz hívva, amikor a Lua egy újabb sor feldolgozásába kezd.

Nullától különböző `count` érték esetén a hurok minden `count` végrehajtott művelet után meg lesz hívva.

Ha argumentumok nélkül van meghívva, a `debug.sethook` kikapcsolja a hurkot.

Amikor a hurok meg lesz hívva, az első paramétere egy karakterlánc, ami megadja, mi váltotta ki az eseményt: `"call"`, `"return"` (vagy `"tail return"`), `"line"`, valamint `"count"`. A sor események esetén a hurok az új sor sorszámát kapja meg második paraméterként. A hurkon belül a `getinfo` 2-es szinten történő hívásával a jelenleg futó függvényről kérhető le bővebb információ (a 0. szint a `getinfo` függvény, az 1. a hurok függvény), kivéve, ha az esemény `"tail return"`. Ebben az esetben a Lua csak szimulálja a visszatérést, és a `getinfo` érvénytelen adatokkal fog visszatérni.

`debug.setlocal ([thread,] level, local, value)`

A függvény a verem `level` szintjén lévő függvény `local` indexű változójának a `value` értéket adja. A függvény visszatérési értéke **nil**, ha a megadott indexen nincs lokális változó, és hibát ér el abban az esetben, ha a `level` a tartmányon kívüli érték. (A `debug.getinfo` függvénnyel ellenőrizhető, hogy a szint érvényes -e.) Egyéb esetben a lokális változó nevével tér vissza.

`debug.setmetatable (object, table)`

A megadott `object` objektum metatömbjét cseréli le a `table` tömbre (ami **nil** is lehet).

`debug.setupvalue (func, up, value)`

A függvény a megadott `func` függvény `up` indexen lévő felsőértékének a `value` értéket adja. A függvény visszatérési értke **nil**, ha a megadott indexen nincs felsőérték. Egyéb esetben a felsőérték nevével tér vissza.

`debug.traceback ([thread,] [message])`

Visszatérési értéke egy karakterlánc, ami hívó verem visszavezetése. Az opcionális `message` karakterlánc a visszavezetés elejére lesz fűzve. A függvény főleg az `xpcall` függvény esetén van használatban, hogy jobb hibaüzeneteket lehessen előállítani.

6 - Lua feldolgozó (stand-alone)

Annak ellenére, hogy a Lua egy kiterjeszett nyelvként lett létrehozva, amelyet a C programba ágyazva lehet használni, gyakran használják egyedülálló nyelvként is. Az alap csomagban szerepel egy feldolgozó, melynek a neve egyszerűen csak `lua`. Ez tartalmazza az alapértelmezett függvénykönyvtárakat, beleértve a hibakeresőt is. A használata a következő:

```
lua [options] [script [args]]
```

Az opciók a következők lehetnek:

- **`-e` *`stat`*:** végrehajtja a *stat* kifejezést;
- **`-l` *`mod`*:** "betölti" a *mod* modult;
- **`-i`:** a *script* futtatása után interaktív módba vált;
- **`-v`:** kiírja a verzió információkat;
- **`--`:** kikapcsolja az opciók kezelését;
- **`-`:** az `stdin`-t fájlként hajtja végre, és kikapcsolja az opciók kezelését.

Az opciók kezelés után a `lua` futtatja a megadott *scriptet*, a megadott *args* karakterláncokat pedig argumentumként társítja hozzá. Ha argumentumok nélkül van meghívva, a `lua` viselkedése ugyanaz, mint a `lua -v -i` esetén, amikor az alapértelmezett bemenet (`stdin`) egy terminál, egyéb esetben úgy, mint a `lua -` .

Bármilyen argumentum futtatása előtt, a feldolgozó ellenőrzi a `LUA_INIT` környezeti változót. Ha a formátuma `@`*`filename`*, akkor a `lua` végrehajtja a fájlt. Egyébként a `lua` az ott megadott karakterláncot hajtja végre.

Minden opció sorrendben lesz kezelve, kivéve a `-i`. Például a következő indítás:

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

`a` értékét 1-re állítja, majd kiírja az `a` értékét (ami '`1`'), végül argumentumok nélkül futtatja a `script.lua` fájlt. (Itt a `$` jel a parancsértelmező készenléti jele. Ez rendszerenként eltérő lehet.)

A script végrehajtása előtt a `lua` a parancssor összes argumentumát egy `arg` nevű globális tömbbe gyűjti össze. A script neve a 0 indexen van tárolva, a script neve után első argumentum az 1 indexen, és így tovább. A script neve előtti argumentumok (tehát, a feldolgozó neve és az opciók után, de a script neve előtt), negatív indexeket kapnak. Így a következő hívás esetén:

```
$ lua -la b.lua t1 t2
```

a feldolgozó futtatja az `a.lua` fájlt, és létrehoz egy tömböt

```
arg = { [-2] = "lua", [-1] = "-la",
[0] = "b.lua",
[1] = "t1", [2] = "t2" }
```

és végül futtatja a `b.lua` fájlt. A script az `arg[1]`, `arg[2]`, ˇˇˇ argumentumokkal lesz meghívva; ezek az argumentumok elérhetőek a scriptből is a vararg kifejezés '...' használatával.

Interaktív módban, ha egy félkész állítás kerül bevitelre, a feldolgozó vár annak befejezésére, egy új készenléti jel megjelenítésével.

Ha a `_PROMPT` globális változó tartalmaz egy karakterláncot, az az érték lesz a készenléti jel. Hasonlóan, ha a `_PROMPT2` tartalmaz karakterláncot, annak értéke lesz a másodlagos készenléti jel (befejezetlen állítások esetén lesz megjelenítve). Tehát mindkét készenléti jel megváltoztatható közvetlenül a parancssorból is, például:

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(a külső idézőjelek a parancsétrelmező számára vannak, a belső pár pedig a Lua számára), vagy a Lua programból, a `_PROMPT` változónak történő értékadással. Ne feledjük el használni a `-i` kapcsolót, hogy interaktív módba lépjünk, ellenkező esetben a program csak 'csendben' véget érne, közvetlenül a `_PROMPT` változónak történő értékadás után.

Unix rendszereken a Lua beállítható, mint script-feldolgozó, mivel az nem veszi figyelembe az első sort, ha az `#` karakterrel kezdődik. Így a Lua scriptek végrehajtható programokká alakíthatóak át a `chmod +x` paranccsal, és a `#!` formulával:

```
#!/usr/local/bin/lua
```

(természetesen a Lua feldolgozója gépenként eltérő lehet. Ha a `lua` a `PATH` útvonalon van, akkor a

```
#!/usr/bin/env lua
```

egy sokkal jobb, átvihetőbb megoldás.)

7 - Eltérések az előző verzióhoz képest

A következőkben szerepelnek az eltérések, amelyeket a Lua 5.0 és Lua 5.1 közötti átalakítás során figyelembe kell venni. A legtöbb eltérés (inkompabilitás) megelőzhető, ha a Lua a megfelelő opciókkal van lefordítva (lásd a `luaconf.h` fájlt). Mindezek ellenére, ezek az opciók a Lua következő verziójában már nem fognak szerepelni.

7.1 - Változtatások a nyelvben

- A vararg rendszer az `arg` pszeudo-argumentumra lett megváltoztatva, ami egy tömb, ami az extra argumentumokat tartalmazza. (`LUA_COMPAT_VARARG` kapcsoló a `luaconf.h` fájlban.)
- Egy finomítás lépett életbe a **for** és **repeat** állítások implicit változóinak láthatóságánál.

- A hosszú karakterláncok / hosszá megjegyzések szintakszisa (`[[string]]`) nem engedélyezi az egymásba ágyazást. Az új szintakszis használható helyette (`[=[string]=]`) (`LUA_COMPAT_LSTR` kapcsoló a `luaconf.h` fájlban.)

7.2 - Változtatások a függvénykönyvtárakban

- A `string.gfind` függvény át lett nevezve, új neve `string.gmatch`. (`LUA_COMPAT_GFIND` kapcsoló)
- Ha a `string.gsub` harmadik argumentuma függvény, és annak visszatérési értéke **nil** vagy **false**, a cserekifejezés az egész egyezés, az üres karakterlánccal ellentétben.
- A `table.setn` el lett távolítva. A `table.getn` függvény helyett mostantól az új hossz operátor (`#`) használandó. (`LUA_COMPAT_GETN` kapcsoló)
- A `loadlib` függvény át lett nevezve, új neve `package.loadlib`. (`LUA_COMPAT_LOADLIB` kapcsoló)
- A `math.mod` függvény át lett nevezve, új neve `math.fmod`. (`LUA_COMPAT_MOD` kapcsoló)
- A `table.foreach` és `table.foreachi` függvények el lettek távolítva. Helyette for ciklus használandó `pairs` és `ipairs` kifejezésekkel.
- Nagy változások léptek életbe a `require` függvényben az új modulrendszer miatt. Annak ellenére, hogy a működése hasonló, és kompatibilis a régivel, a `require` az útvonalat a `package.path` változóból kéri le a `LUA_PATH` helyett.
- A `collectgarbage` függvény különböző argumentumokat kapott. A `gcinfo` függvény el lett távolítva, helyette a `collectgarbage("count")` használandó.

7.3 - Változtatások az API-ban

- A `luaopen_*` függvények (függvénykönyvtárak megnyitása) nem hívhatóak meg közvetlenül, mint egy sima C függvény. A Lua-n keresztül kell meghívni, mint egy Lua függvényt.
- A `lua_open` le lett cserélve a `lua_newstate` függvényre, hogy a felhasználó beállíthasson egy memória-lefoglaló függvényt. Az alapértelmezett függvénykönyvtár `luaL_newstate` függvényének használatával egy állapot készíthető, amely az alapértelmezett lefoglaló függvény lesz (ami a `realloc` függvényen alapszik).
- A `luaL_getn` és `luaL_setn` függvények (a segédkönyvtárban) el lettek távolítva. A `luaL_getn` helyett a `lua_objlen` használható, a `luaL_setn` helyett pedig semmi, véglegesen el lett távolítva.
- A `luaL_openlib` le lett cserélve a `luaL_register` függvényre.
- A `luaL_checkudata` függvény mostantól hibát dob, ha a megadott érték nem az elvárt típusú userdata típus. (5.0-ban a visszatérési értéke `NULL` volt.)

8 - A Lua teljes szintakszisa

A következőkben a Lua teljes szintakszisa szerepel, kiterjesztett BNF formátumban. (Ez nem írja le az operátorok precedenciáját.)

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::= varlist1 `=´ explist1 |

functioncall |

do block end |

while exp do block end |

repeat block until exp |

if exp then block {elseif exp then block} [else block] end |

for Name `=´ exp `,´ exp [`,´ exp] do block end |

for namelist in explist1 do block end |

function funcname funcbody |

local function Name funcbody |

local namelist [`=´ explist1]

laststat ::= return [explist1] | break

funcname ::= Name {`.´ Name} [`:´ Name]

varlist1 ::= var {`,´ var}

var ::= Name | prefixexp `[´ exp `]´ | prefixexp `.´ Name

namelist ::= Name {`,´ Name}

explist1 ::= {exp `,´} exp

exp ::= nil | false | true | Number | String | `...´ | function |

prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | `(´ exp `)´

functioncall ::= prefixexp args | prefixexp `:´ Name args

args ::= `(´ [explist1] `)´ | tableconstructor | String

function ::= function funcbody

funcbody ::= `(´ [parlist1] `)´ block end

parlist1 ::= namelist [`,´ `...´] | `...´

tableconstructor ::= `{´ [fieldlist] `}´

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
```

```
fieldsep ::= `,´ | `;´

binop ::= `+´ | `-´ | `*´ | `/´ | `^´ | `%´ | `..´ |

`<´ | `<=´ | `>´ | `>=´ | `==´ | `~=´ |

and | or

unop ::= `-´ | not | `#´
```

Utolsó módosítás: Mon Jun 5 17:05:27 BRT 2006
Fordítás dátuma: Sat Oct 25 10:13:20 CET 2008
Fordítási megjegyzések: néhány kifejezés csak nehezen ültethető át magyar nyelvre. Lehet hogy van rá szabványosított kifejezés, én azonban ilyenről - a fordítás befejezésekor - nem tudtam. A könnyebb megérthetőség érdekében most néhány kifejezés fel lesz tüntetve az eredeti angol megfelelőjével:

eljáráskönyvtár	library
értékkiemelés	captures
feldolgozó	interpreter
felső érték	upvalue
felsorolás	enum, enumeration
függvénykönyvtár	lásd eljáráskönyvtár
hibát ér el	raises an error
hurok	hook
kliens	host
rendezett tömb	array
szemétgyűjtő	garbage collector
tömb	table (bár a 'table' lehet hash is. lásd rendezett tömb)
véghívás	tail call
zárvány	closure

Hibás fordítás, elírás és ötletek jelenthetőek az e-mail címemen: ejjeliorjarat@gmail.com