What is Next for Lua?
A Personal Perspective

Roberto Ierusalimschy

What is Next?

- (No breakthroughs…)
- Libraries
- Unicode
- Integers

Libraries

- LPeg
- struct/pack

LPeg: What it is

- A library for pattern matching
- Goes from simple patterns to full grammars

```
"[a-z]+"
```

```
[[
Sexp <- atom / '(' sp Sexp* ')' sp
atom <- %w+ sp
sp   <- %s*
]]
```

LPeg: Pros

- A good balance of expressiveness and complexity
- It may become a real differential for Lua
- More “Unicode-friendly”
  - e.g., "—*" (zero or more em dashes)

LPeg: Cons

- Redundant with current pattern matching
  - ideally we should deprecate current implementation, but transition is not always easy
- Not so small
  - half the size of all current libraries together
- Not so mature

Struct: What it is

- A library for packing/unpacking binary data in strings

```
s = struct.pack("iic", -24, 13, "x")

print(struct.unpack("iic", s))
  --> -24    13    x    10
```

Struct: Pros

- Small and simple
- Common in several scripting languages
- Wide range of uses
  - binary data in sockets
  - packing of data inside Lua

Struct: Cons

- Conflict with future features
  - e.g., packing/unpacking of C data outside Lua, in the host program

Unicode

- What does it mean "support Unicode"?
- What encoding should Lua use?
- Do we need a new type for Unicode strings?
  - (NO!!!)

Unicode "Support"

- Lua has no intention of "supporting" Unicode
  - for any reasonable definition of "support"
- Unicode is too complex for Lua
  - too many tables, all huge
- But Lua can offer some very basic primitives to ease the coding of other libraries or simple tasks
- Mostly, operations to deal with the encoding

Encoding

- UTF-8 seems the clear winner
- UTF-16 has the same problems of UTF-8 plus some others
  - no easy access to i-th character
- UTF-8 can be smaller even for Asian languages
  - e.g.: front page of Wikipedia Japan: 83 kB in UTF-8, 144 kB in UTF-16

Encoding

Moreover, UTF-8 is much simpler for Lua :-)

Lua and UTF-8

- Lua strings work naturally with UTF-8
- Literal strings can contain UTF-8 characters
  - as long as text editor allows
- I/O works naturally with UTF-8
  - provided OS does not interfere
- File names and the like depend on the OS
- Many string-manipulation functions do not work properly with UTF-8
  - string.char, string.byte, string.upper, string.lower, string.reverse

Lua and UTF-8

- Some useful tricks with pattern matching

```
t = "ÃøÆËÐ"
print(#(string.gsub(t, "[\128-\191]", "")))
   --> 5

for c in string.gmatch(t, ".[\128-\191]*") do
  print(c)
end
  --> Ã
  --> ø
  --> Æ
  --> Ë
  --> Ð
```

A new UTF-8 Library

- utf8.char (num, num, ...)
  - returns a utf-8 string formed from the given code points
- utf8.codepoint (s, [i, [j]])
  - returns the code points of the string s:sub(i,j)
  - j defaults to i, but it always corrected to include a complete byte sequence
- utf8.len (s, [l])
  - number of code points in s up to byte l
  - nil if string is not properly formed

A new UTF-8 Library

- utf8.byteoffset (s, l, [i])
  - byte offset where l-th byte sequence starts (after position i)
  - l can be 0 (offset where current sequence starts) or negative, too
- utf8.gcodepoint (s, [i, [j]])
  - iterator for code points

Integers: What

- Add an integer type to Lua
- That type could be either a 32 or 64-bit signed integral type

Integers: Why

- 64 bits!
  - mainly for external entities
  - special algorithms
  - counting: is 2^{53} enough?
- Restricted systems: 32-bit integers + single precision floats
  - better performance on hardware with no FP support (or support only for single precision)

Integers: Alternatives

- Extra type for 64-bit values
  - e.g., userdata
- Extra type for floating-point numbers
  - on restricted systems using integers as numbers
- Both alternatives seem cumbersome
  - new operations in the API?
  - how they behave with arithmetic operations?
  - equality with numbers?

Integers: Alternatives

- Larger floating point
  - main reason to use doubles (long time ago)
  - too expensive (uses more memory)
  - does not solve related problem (small machines)

Integers: Pros

- Most programmers already expect an integer type.
- No need to explain about precision of floating arithmetic :)
- Make explicit an integer type already implicit in several libraries
- No need to change numbers to integers on restricted hardware.
  - easier support if difference is only int32 x int64 and float x double.

Integers: Cons

- Added complexity to the language
- Added complexity to the code
- Added frequency of occurrences of small variants
  - 32int x 64int and float x double
- For current 32-bit machines:
  - 32int + double offers few gains
  - 32int + float is incompatible
  - 64int + double slows down the interpreter

Integers: How

- Three options (at least)

1) Explicit new type
   - incompatible
   - too complex (?)

2) “Invisible”
   - representation depends only on the value
   - equal values mean equal representation
   - not really invisible; subtle rules
   - too complex (rules and implementation)

Integers: How

3) Subtype

- almost invisible

- It is possible to know whether a number is an integer or a float
  - `isfloat/isint` (?)
- 1 is integer; 1.0 is float
  - but 1 == 1.0

Integers: How

```
type(1) == "number"
type(1.0) == "number"
1 == 1.0
```

But:

```
1 + 2^60 > 2^60        -- assuming 64-bit integers
1.0 + 2^60 == 2^60   -- double
print(1)        --> 1
print(1.0)      --> 1.0    (?)
```

Integers: How

For all arithmetic operations except division and exponentiation:

- If both operands are integers, the operation is performed on integers and the result is an integer.

- Otherwise, operands are converted to float, the operation is performed on floats, and the result is a float.

Exponentiation is almost like other operators, but performed on floats if exponent is negative

Integers: How

- All operations except division give integer results when operands are integer
  - (other exception is x^-y)
- Therefore, they give the same results when performed either on integers or on floats, *except for overflows*
  - that includes comparisons
  - for overflows, floats lose precision
  - what should happen to integer overflow?

Integer Overflow

1) Convert to double

- best for compatibility
- few other uses (except for 32 x double configurations)

2) Raise an error

- more secure (there are no surprises)
- check may be expensive
- rule out some useful tricks

3) Wrap around

- dangerous, but has its uses
- cheap implementation

Integer Division

- Two different operations: float division and integer division
- Float divison: x/y, result is always float
- Integer division: x//y, result is always integer
  - notation borrowed from Python
  - floor of x/y
- x/1 converts to float, x//1 converts to integer

Integers and Tables

- When used as a key, a float with an "integer value" is always converted to an integer
  - "integer value" means that x == x//1
  - `if x == x//1 then x = x//1 end`
- Test is already present in current implementation
  - but invisible to the programmer

Integers: Equality

- For equality, adopting the same rules of arithmetic operations leads to some nasty properties
  - equality is not transitive: 2^60 == 2.0^60 and (2^60 + 1) == 2.0^60, but 2^60 ~= (2^60 + 1)
- Another definition: x == y iff (x/1 == y/1 **and** x//1 == y//1)
  - 2^60 == 2.0^60 but (2^60 + 1) ~= 2.0^60
  - more expensive implementation

Integers: Order

- Same rules as for arithmetic operations?
- Some nasty properties
  - order is not transitive: 2^60 <= 2.0^60 and (2^60 + 1) <= 2.0^60, but not 2^60 <= (2^60 + 1)
  - order is not strict: 2^60 <= 2.0^60 and 2.0^60 <= 2^60, but 2^60 ~= 2.0^60
- Equality rules have nasty properties, too
  - order is not total
  - more expensive implementation

Integers: C API

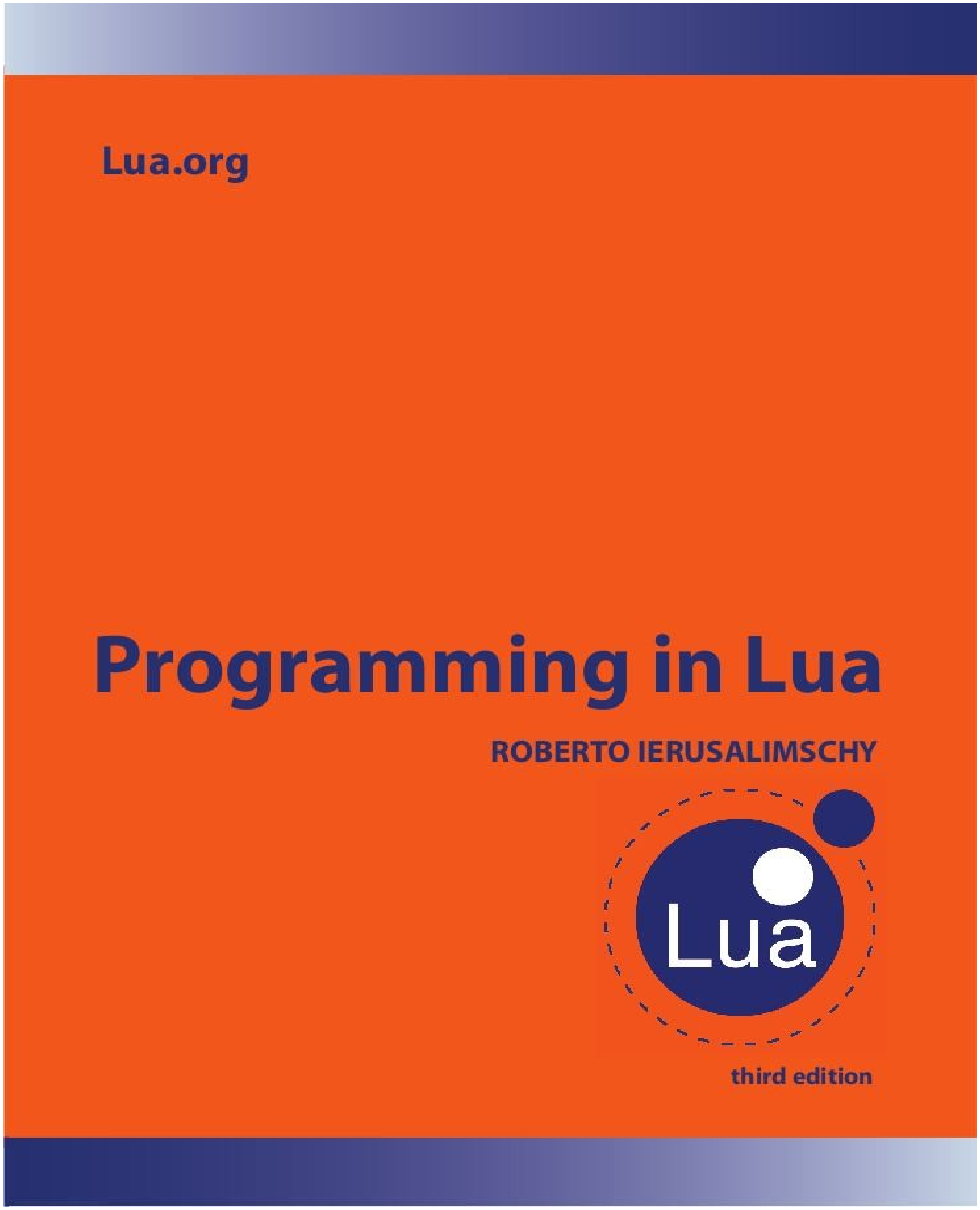
- `lua_pushinteger` creates an integer, `lua_pushnumber` creates a float
- `lua_tointeger` converts to integer, `lua_tonumber` converts to float
  - following the same conversion rules of x//1 and x/1
- `lua_Unsigned` probably will be the unsigned version of `lua_Integer`

Integers: Other Issues

- `tonumber`: result can be integer or float, following the same rules of the scanner
- `io.read("*n")`: result is float; new format (`"*i"`?) for reading integers
- Coercion from string to number: always results in float
  - simpler implementation, compatible, and leading to extinction
- `tostring` (and `print`): floats always have a decimal mark?

That is it.

(for now...)



978-85-903798-5-0

Coming soon to a bookstore “near” you...