Lua

Jani Kajala (jani.kajala@helsinki.fi)

Helsinki 27th March 2004

Principles of Programming Languages -course seminar

HELSINKI UNIVERSITY
Computer science department

Lua

Jani Kajala (jani.kajala@helsinki.fi)

Principles of Programming Languages -course seminar
Computer science department
Helsinki University
27th March 2004, 3+7 pages

This seminar paper describes properties of Lua programming language. Lua is a scripting language well suited for extending C/C++ and other applications with scriptable functionality.

Computing Reviews (1998): D.3.1, I.7.2

Key words: programming languages, scripting languages, Lua

Contents

1 Introduction

Lua is a small language especially well suited for extending C/C++ applications to be scriptable. Its origins are in data definition language designed in 1993 for automating task of data input to simulations in Brazilian oil industry.

2 Lexical conventions

Lexical conventions match closely to most other common languages. Identifiers and key words are case-sensitive. Identifiers need to begin with alphabetic character or underscore, and any alphanumeric character can follow. Only single-line comments are supported; comments start with $--$ (two dashes). String literals can be created with C-like " " pair and contain escape sequences like \n (newline). Lua supports also optionally multiline string literals in form of [[ ]] pair, for example:

```
a = [[ this is
multiline string
literal ]]
```

String literals created with [[ ]] pair can span multiple lines but espace sequences are not expanded inside the pair.

To conform Unix scripting conventions, if *chunk* starts with # character the first line is skipped. Chunk is the unit of execution in Lua – more abount chunks in section 4.

3 Types and values

Lua is dynamically typed language, as most scripting languages are. Basic types in Lua are nil, boolean, number, string, table, function and userdata.

Boolean and number have value semantics. Strings have internally reference semantics, but strings are immutable, so they actually appear to user as having value semantics. Nil is special type which roughly matches C NULL value. Function instantiations, *closures*, are first-class variables in Lua and they have reference copy semantics, as well as tables and userdata.

Lua relies heavily on associative arrays, that is arrays which can be accessed by any type except nil. Associative arrays are implemented with *table* type in Lua. Tables can also be heterogeneous, they can contain values of all types except nil. Tables are the only data structuring mechanism in Lua. In addition to named record access using $array["name"]$, Lua provides syntactic sugar $array.name$ and iteration over table elements using $next(array, key)$ returns key and value of next array element.

Userdata is special data type which can be modified only from C code. This guarantees integrity when extending C application by scripting as the scripts cannot modify the value of userdata. Userdata has reference semantics, as do have tables and functions. *Metatables* can be used to define operations on userdata values, which allows Lua to be extended with custom types. Every table has metatable, which defines behaviour of the original table. For example in in addition Lua calls __add member of the metatable.

Lua is not object oriented language, but it does support object oriented programming by the usage of tables. Lua tables can contain associations to functions and the table itself can be passed as first parameter to the function by using : operator. For example $a : f()$ calls function f of table a and passes the table as an argument

to the function.

Lua provides coercion between strings and numbers in run-time. This is applied so that any arithmetic operation tries to convert a string to number. Also when a string is expected a number is converted to string automatically.

4 Statements

As Lua was intended for extending applications, it has no concept of $main()$ function. Basic unit of execution is *chunk*. Chunk is sequence of Lua statements, which are optionally followed by semicolon. Chunks are interpreted as anonymouys function, so chunks can have local variables and chunks can return values.

In addition to traditional assignment, Lua supports assignment of multiple values. For example

```
a,b,c = 1,2,3
```

assigns a=1, b=2 and c=3. Parameters to assignment are evaluated applicatively so that

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

performs swap operation correctly between x and y.

Lua as controls structures similar to other common imperative languages. For example following code prints odd positive numbers below 100:

```
local n = 1
local odd = true
while n < 100 do
  if odd then
```

```
    print( n )
  end
  odd = not odd
  n = n + 1
end
```

In addition to $while$, Lua has $repeat$ and for loops. Repeat has the form

```
repeat block until exp
```

for loop has two forms, numerical and generic. Numeric form is

```
for i=first,increment,last do block end
```

and generic form is

```
for v1,...,vn in explist do block end
```

Generic for form is shorthand for the code below. Note that $v1, v2..., vn$ are all locals inside for loop.

```
do
  local _f, _s, v1 = explist
  local v2, ... , vn
  while true do
    v1, ..., vn = _f(_s, v1)
    if v1 == nil then break end
    block
  end
end
```

Lua has also $return$ and $break$ statements. $return$ is used to return value from block and $break$ is used to break iteration inside loop. Lua's $return$ statement supports multiple return values. The other difference to C language $return$ semantics is that $return$ and $break$ can only be executed as the last statement of their (inner) blocks.

5 Expressions

Lua expressions are similar to C expressions. In arithmetic operators the main difference is power operator $\wedge$, which is only present as pow standard library function in C. Relational operators are also similar to C, with the difference of inequality which is $\sim=$ in Lua. Logical operators are expressed in written form: and, not, or. Booleans are $true$ and $false$, but they are quite recent addition to Lua and so nil is still considered false as well as in versions preceding Lua 5.

String concatenation is done with .. operator. Due to coercion rules, numbers can be concatenated to strings with this as well.

Precedence order in Lua is similar to other common languages like C. Concatenation comes after arithmetic but before relational operators.

6 Functions

Lua functions are first-class variables with reference copy semantics. Function is defined by

```
function f( parameters )
   ...
end
```

This is actually syntactic sugar for

```
f = function( parameters )
   ...
end
```

which more properly shows what is happening when Lua executes the definition. When Lua executes a function definition, the function is *closed*. Different function instances, *closures*, can have different running environment, if they refer to parent block local variables. For example

```
f = function( a, b )
   return function() a+b end
end
```

creates a function which returns value of $a + b$ at the time when function f was called. As introduced in section 4, functions can return multiple values using $return$ with multiple parameters separated with comma. As previous example suggested, Lua also supports anonymous and non-global functions. Variable number of arguments is supported by defaulting all arguments to nil. Lua also supports proper tail recursion, in other words recursion at the end of the function re-uses the same stack space for the call, so that tail recursion can have unlimited depth.

References

Ier03 R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua 5.0 Reference Manual*. Technical Report MCC-14/03, PUC-Rio, 2003.

Sco00 Scott, M., *Programming Language Pragmatics*. Morgan Kauffman, 2000.