Programming in Lua – Handling Errors

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Errors

- Lua functions treat erroneous inputs in two ways: returning `nil` plus an error message, or *raising* an error

- Functions use the first way when problems are expected; for example, opening a file is always a risk, as the file might not exist, or the user may not have permissions to open it:

```
> print(io.open("notafile.txt"))
nil     notafile.txt: No such file or directory 2
```

- Functions use the second way when problems are *exceptional*, such as problems resulting from bugs in the code:

```
> print(math.sin("foo"))
stdin:1: bad argument #1 to 'sin' (number expected, got string)
stack traceback:
        [C]: in function 'sin'
        stdin:1: in main chunk
        [C]: in ?
```

From error messages to errors

- The `assert` built-in function turns errors of the first kind into errors of the second kind:

```
> print(assert(io.open("foo.txt")))
file (000007FF650BE2D0)
> print(assert(io.open("notafile.txt")))
stdin:1: notafile.txt: No such file or directory
stack traceback:
        [C]: in function 'assert'
        stdin:1: in main chunk
        [C]: in ?
```

- The error built-in function takes an error message and raises an error:

```
> error("raising an error")
stdin:1: raising an error
stack traceback:
        [C]: in function 'error'
        stdin:1: in main chunk
        [C]: in ?
```

Integer division, with and without errors

- The two implementations of an integer division function below show the two kinds of error reporting:

```lua
function idiv1(a, b)
  if b == 0 then
    return nil, "division by zero"
  else
    return math.floor(a/b)
  end
end

function idiv2(a, b)
  if b == 0 then
    error("division by zero")
  else
    return math.floor(a/b)
  end
end
```

```
> print(idiv1(2,0))
nil     division by zero
> print(idiv2(2,0))
stdin:3: division by zero
stack traceback:
        [C]: in function 'error'
        stdin:3: in function 'idiv2'
        stdin:1: in main chunk
        [C]: in ?
```

Shifting blame

- Notice that Lua reports the “division by zero” error as ocurring in line 3 of function `idiv2`; this is the default behavior of error

- But we may want to shift the blame to `idiv2`’s caller, as it is responsible for passing the 0 that is leading to the error; we can do this with an optional second argument to error:

```
function idiv2(a, b)
  if b == 0 then
    error("division by zero", 2)
  else
    return math.floor(a/b)
  end
end
```

ERROR LEVEL

```
> print(idiv2(2,0))
stdin:1: division by zero
stack traceback:
        [C]: in function 'error'
        stdin:3: in function 'idiv2'
        stdin:1: in main chunk
        [C]: in ?
```

- The argument is the *level*, where 1 is the function calling `error`, 2 is its caller, 3 its caller’s caller, and so on; shifting the blame does not change the traceback

Catching errors

- Raising an error aborts execution by default; if we are in the REPL, we go back to the REPL's prompt
- We can catch and handle errors using the `pcall` built-in function, which takes a function to call and returns:
  - → `true` followed by the results of the function, if there are no errors
  - → `false` followed by the error message, if there was an error
- Any extra arguments to `pcall` are passed along

```
> print(pcall(idiv2, 5, 2))
true    2
> print(pcall(idiv2, 5, 0))
false   division by zero
```

Catching errors (2)

- `pcall` returns just the error message when an error occurs

- If the code needs more information it can use the `xpcall` builtin function; `xpcall` takes two arguments, a 0-parameter function to call and an error handler function to call if an error occurs:

```
> handler = function (err) return err .. "\n" .. debug.traceback() end
> print(xpcall(function () return idiv2(5, 2) end, handler))
true    2
> print(xpcall(function () return idiv2(5, 0) end, handler))
false   division by zero
stack traceback:
        stdin:1: in function <stdin:1>
        [C]: in function 'error'
        stdin:3: in function <stdin:1>
        (...tail calls...)
        [C]: in function 'xpcall'
        stdin:1: in main chunk
        [C]: in ?
```

Quiz

- Calling `nil` raises an “attempt to call a nil value” error. What is the result of `pcall(nil)`? And `pcall(pcall, nil)`?

false · attempt to call a nil value

true false attempt to call a nil value