LUA - ERROR HANDLING

Need for Error Handling

Error handling is quite critical since real-world operations often require the use of complex operations, which includes file operations, database transactions and web service calls.

In any programming, there is always a requirement for error handling. Errors can be of two types which includes,

- Syntax errors
- Run time errors

Syntax Errors

Syntax errors occur due to improper use of various program components like operators and expressions. A simple example for syntax error is shown below.

```
a == 2
```

As you know, there is a difference between the use of a single "equal to" and double "equal to". Using one instead of the other can lead to an error. One "equal to" refers to assignment while a double "equal to" refers to comparison. Similarly, we have expressions and functions having their predefined ways of implementation.

Another example for syntax error is shown below −

```
for a= 1,10
   print(a)
end
```

When we run the above program, we will get the following output −

```
lua: test2.lua:2: 'do' expected near 'print'
```

Syntax errors are much easier to handle than run time errors since, the Lua interpreter locates the error more clearly than in case of runtime error. From the above error, we can know easily that adding a *do* statement before print statement is required as per the Lua structure.

Run Time Errors

In case of runtime errors, the program executes successfully, but it can result in runtime errors due to mistakes in input or mishandled functions. A simple example to show run time error is shown below.

```
function add(a,b)
   return a+b
end

add(10)
```

When we build the program, it will build successfully and run. Once it runs, shows a run time error.

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
 test2.lua:2: in function 'add'
 test2.lua:5: in main chunk
 [C]: ?
```

This is a runtime error, which had occurred due to not passing two variables. The **b** parameter is expected and here it is nil and produces an error.

Assert and Error Functions

In order to handle errors, we often use two functions − **assert** and **error**. A simple example is shown below.

```
local function add(a,b)
   assert(type(a) == "number", "a is not a number")
   assert(type(b) == "number", "b is not a number")
   return a+b
end

add(10)
```

When we run the above program, we will get the following error output.

```
lua: test2.lua:3: b is not a number
stack traceback:
 [C]: in function 'assert'
 test2.lua:3: in function 'add'
 test2.lua:6: in main chunk
 [C]: ?
```

The **error** ***message*[, *level*]** terminates the last protected function called and returns message as the error message. This function error never returns. Usually, error adds some information about the error position at the beginning of the message. The level argument specifies how to get the error position. With level 1 *thedefault*, the error position is where the error function was called. Level 2 points the error to where the function that called error was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

pcall and xpcall

In Lua programming, in order to avoid throwing these errors and handling errors, we need to use the functions pcall or xpcall.

The **pcall** ***f, arg*1, . . .** function calls the requested function in protected mode. If some error occurs in function f, it does not throw an error. It just returns the status of error. A simple example using pcall is shown below.

```
function myfunction ()
   n = n/nil
end

if pcall(myfunction) then
   print("Success")
else
 print("Failure")
end
```

When we run the above program, we will get the following output.

```
Failure
```

The **xpcall** ***f, err*** function calls the requested function and also sets the error handler. Any error inside f is not propagated; instead, xpcall catches the error, calls the err function with the original error object, and returns a status code.

A simple example for xpcall is shown below.

```
function myfunction ()
   n = n/nil
end
```

```
function myerrorhandler( err )
   print( "ERROR:", err )
end

status = xpcall( myfunction, myerrorhandler )
print( status)
```

When we run the above program, we will get the following output.

```
ERROR: test2.lua:2: attempt to perform arithmetic on global 'n' (a nil value)
false
```

As a programmer, it is most important to ensure that you take care of proper error handling in the programs you write. Using error handling can ensure that unexpected conditions beyond the boundary conditions are handled without disturbing the user of the program.