



Η Γλώσσα Προγραμματισμού Lua

ΕΡΓΑΣΙΑ ΓΙΑ ΤΟ ΜΑΘΗΜΑ ΤΩΝ ΜΕΤΑΓΛΩΤΤΙΣΤΩΝ

Ευστάθιος Καραγιάννης | Κωνσταντίνος Λεοντιάδης | Ιωάννης Παπαδημητρίου

Περιεχόμενα

Ιστορία.....	3
ΕΚΔΟΣΗ 1.....	4
ΕΚΔΟΣΗ 2.....	4
ΕΚΔΟΣΗ 3.....	5
ΕΚΔΟΣΗ 4.....	5
ΕΚΔΟΣΗ 5.....	5
Μερίδιο Αγοράς.....	6
Προδιαγραφές της Lua	9
Τύποι Δεδομένων.....	10
Μεταβλητές	11
Ονομασία Μεταβλητών.....	11
Εκχώρηση τιμών	12
Εύρος Μεταβλητών	12
Τοπικές Μεταβλητές	12
Global Μεταβλητές	12
Κύκλος Ζωής Μεταβλητών	12
Προσπέλαση Μεταβλητών.....	12
Τελεστές	12
Αριθμητικοί Τελεστές.....	12
Σχεσιακοί Τελεστές	13
Λογικοί Τελεστές	13
Τελεστές Ανάθεσης	13
Δυφιοτελής Τελεστές	13
Λοιποί Τελεστές.....	13
Επαναληπτές	14
Ipairs	14
Pairs	14
String.gmatch	14
«Τεχνητοί» Επαναληπτές	14
Δομές Δεδομένων	14
Πίνακες (Tables).....	14

Arrays	14
Linked List	15
Queues	15
Stack.....	15
Set.....	15
Αναφορές.....	15
Δομές Ελέγχου.....	16
Υποθετικές Δομές	16
Δομές Επανάληψης	17
Διαχείριση Λαθών (Error Handling)	17
Διαχείριση Ροής	17
Interoperability	17
Συναρτησιακός προγραμματισμός.....	18
Συναρτήσεις.....	18
Μεταπίνακες και Μεταμέθοδοι.....	19
Κορυτίνες	20
Αντικειμενοστραφής προγραμματισμός	21
Κλάσεις	22
Κληρονομικότητα	23
Πολυμορφισμός.....	24
Ενθυλάκωση	25
Πακέτα.....	25
Garbage Collector.....	26
Weak Tables	28
Incremental Garbage Collector.....	29
Generational Garbage Collector	30
Γραμματική και Συντακτικό.....	31
Εισαγωγή.....	31
Lua Parser	32
Καταστάσεις της Lua	34
Παράδειγμα Γραμματικής Ανάλυσης.....	35
Συντακτική Ανάλυση.....	36
Μεταγλωττιστής.....	37

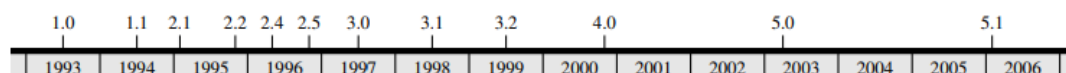
Χρήση της Lua.....	42
FRAMEWORKS.....	42
APPLICATIONS	43
Παιχνίδια και Game Engines.....	44

Ιστορία

Η Lua (Φεγγάρι στα Πορτογαλικά) δημιουργήθηκε από μία Βραζιλιάνικη ομάδα στο Παπικό Καθολικό Πανεπιστήμιο του Ρίο ντε Τζανέιρο (PUC-Rio) με επικεφαλής τους Roberto Ierusalimsky, Waldemar Celes και Luiz Henrique de Figueiredo. Ο Ρομπέρτο ασχολούταν κυρίως με γλώσσες προγραμματισμού, ο Λουίζ Χενρίκε ήταν μαθηματικός με ενδιαφέρον στα εργαλεία λογισμικού και ο Waldemar ήταν μηχανικός που διερευνούσε τις εφαρμογές των γραφικών στους υπολογιστές.

Η ομάδα ήθελε μια γλώσσα που να είναι απλή, αποτελεσματική και να μπορεί εύκολα να ενσωματωθεί σε υπάρχον κώδικα C. Για να είναι μεταφέρσιμη η Lua σχεδιάστηκε με μινιμαλιστική σύνταξη (μόνο 21 δεσμευμένες λέξεις - η Python έχει 33 για σύγκριση) και μικρό κόστος χρόνου εκτέλεσης, καθιστώντας την κατάλληλη για ενσωμάτωση σε άλλες εφαρμογές και συστήματα με περιορισμένους πόρους. Αυτές οι σχεδιαστικές επιλογές έκαναν τη Lua μια δημοφιλή scripting γλώσσα για ένα ευρύ φάσμα εφαρμογών, συμπεριλαμβανομένων βιντεοπαιχνιδιών και ενσωματωμένων συστημάτων.

Το συντακτικό της Lua ήταν εμπνευσμένο από άλλες γλώσσες υψηλού επιπέδου, όπως η Pascal και η Modula, και σχεδιάστηκε για να είναι ευνόητη και εύχρηστη. Η ομάδα είχε στόχο να διατηρήσει τη γλώσσα απλή και ευέλικτη, με έναν μικρό αριθμό βασικών λειτουργιών που θα μπορούσαν εύκολα να επεκταθούν με βιβλιοθήκες C. Η γλώσσα σχεδιάστηκε επίσης για να έχει ένα σαφές και απλό API για ενσωμάτωση σε άλλα software projects.



Χρονοδιάγραμμα διαφορετικών εκδόσεων της Lua

Η αρχική παρουσίαση της Lua το 1993 έγινε στον Όμιλο Τεχνολογίας Γραφικών Υπολογιστών της PUC-Rio, γνωστός και ως Tecgraf, με μεγάλη επιτυχία. Αυτό

οδήγησε στη δημοφιλία της γλώσσας ανάμεσα στους συμμετέχοντες που έτυχε να είναι παρόντες για άλλα μέρη του Tecgraf. Οι νέοι χρήστες ήθελαν να χρησιμοποιήσουν τη Lua για τη διαχείριση μεγάλων μέτα-αρχείων γραφικών, γεγονός που έθεσε δύσβατες τεχνικές προκλήσεις λόγω του μεγέθους και της πολυπλοκότητας των μέτα-αρχείων αυτών. Γενικότερα, το πρόβλημα ήταν ότι ο μεταγλωττιστής της Lua έπρεπε να χειρίζεται τεράστια προγράμματα και εκφράσεις σχετικά γρήγορα.

Για να βελτιωθεί η απόδοση, ο μεταγλωττιστής Lua βελτιστοποιήθηκε. Ο σαρωτής που μέχρι τότε είχε φτιαχτεί χρησιμοποιώντας lex, το γνωστότερο τότε εργαλείο Unix για την ανάπτυξη γλωσσών προγραμματισμού, με έναν χειρόγραφο (from scratch). Επίσης, έγιναν τροποποιήσεις στην εικονική μηχανή της Lua ώστε να χειρίζεται πιο αποτελεσματικά μεγάλα αρχεία σύμφωνα με τις απαιτήσεις τους από μεγάλους κατασκευαστές.

ΕΚΔΟΣΗ 1

Τον Ιούλιο του 1994, η Lua 1.1 κυκλοφόρησε με αυτές τις βελτιστοποιήσεις και έγινε διαθέσιμη στο κοινό με περιοριστική άδεια – δωρεάν για ακαδημαϊκή χρήση αλλά οτιδήποτε άλλο έπρεπε να διαπραγματευτεί με τους δημιουργούς. Ωστόσο, αυτή η άδεια δεν είχε καλή υποδοχή καθώς οι άλλες scripting γλώσσες της εποχής αυτής ήταν open source, που οδήγησε σε λίγες διαπραγματεύσεις και καμία τελική εμπορική συμφωνία. Έτσι, η Lua 2.1 κυκλοφόρησε ως open source λογισμικό χωρίς περιορισμούς. Η άδεια αργότερα άλλαξε σε άδεια MIT, η οποία είναι ευρέως αποδεκτή από την κοινότητα και από τότε δεν έχουν γίνει γνωστές αμφιβολίες και ανησυχίες μεγάλου σκέλους σχετικά με την άδεια χρήσης της γλώσσας.

Η κυκλοφορία της Lua στην μέση της μεγαλύτερης τάσης προς τον Αντικειμενοστραφή προγραμματισμό οδήγησε στην πίεση από τους χρήστες για την προσθήκη αντικειμενοστραφών λειτουργιών. Οι δημιουργοί αποφάσισαν να μην μετατρέψουν τη Lua σε αντικειμενοστραφή γλώσσα, προκειμένου να αποφύγουν τον καθορισμό ενός συγκεκριμένου παραδείγματος προγραμματισμού, προσπαθώντας να «παρέχουν μηχανισμούς, όχι πολιτικές».

ΕΚΔΟΣΗ 2

Πιο συγκεκριμένα προτίμησαν να παρέχουν ευέλικτους μηχανισμούς που θα επέτρεπαν στους προγραμματιστές να δημιουργήσουν τα δικά τους μοντέλα κατάλληλα για τις εφαρμογές τους. Η Lua 2.1, που κυκλοφόρησε τον Φεβρουάριο του 1995, εισήγαγε επεκτάσιμους μηχανισμούς σημασιολογίας, που επέτρεπαν μεγαλύτερη εκφραστικότητα στους χρήστες, το οποίο και έγινε ένα από τα κυριότερα χαρακτηριστικά της γλώσσας.

Βέβαια, αυτό σημαίνει πως οι δημιουργοί ήθελαν να παρέχουν στους χρήστες τη δυνατότητα για αντικειμενοστραφή προγραμματισμό, χωρίς όμως να είναι προαπαιτούμενο για τη χρήση της γλώσσας.

Λόγω πίεσης από τους χρήστες οι δημιουργοί της Lua επίσης επέκτειναν τις δυνατότητες της γλώσσας για debugging. Πιο συγκεκριμένα στην έκδοση 2.2 οι δημιουργοί πρόσθεσαν API εντοπισμού σφαλμάτων (debugger) το οποίο έδινε στους χρήστες τη δυνατότητα να ελέγχουν την κατάσταση των συναρτήσεων κατά τη διάρκεια εκτέλεσής τους. Το feature αυτό, επίσης, επέτρεψε στους χρήστες να γράψουν τα δικά τους εργαλεία, debuggers και μεταγλωττιστές σε C και να τα χρησιμοποιήσουν στη Lua.

ΕΚΔΟΣΗ 3

Για την έκδοση 3.1 της Lua άρχισαν να γίνονται κουβέντες για multithreading και cooperative multitasking, όμως παρόλο που κατά τους δημιουργούς τα features αυτά ήταν δημοφιλή η εισαγωγή τους έγινε για πρώτη φορά στη έκδοση 5.0.

Ενώ κουβέντες για τα προαναφερόμενα features έμεναν στο πίσω μέρος του μυαλού τους, οι δημιουργοί αποφάσισαν να φέρουν τον συναρτησιακό προγραμματισμό (functional programming) στη Lua με την έκδοση 3.1 με την εισαγωγή των ανώνυμων συναρτήσεων και των 'κελιών συναρτήσεων' (function closures).

Η σημαντικότερη αλλαγή που έφερε η Lua 3.2 είναι η παροχή της δυνατότητας στους χρήστες να γράψουν τους debuggers ή οπουδήποτε εργαλείο τους (το προαναφερόμενο feature της έκδοσης 2.2) στη Lua αντί για τη C.

ΕΚΔΟΣΗ 4

Η μεγαλύτερη πρόκληση που οι δημιουργοί της Lua αποφάσισαν να αντιμετωπίσουν στην έκδοση 4.0 της Lua ήταν η δημιουργία ενός επανεισερχόμενου API (reentrant API), δηλαδή η δυνατότητα της γλώσσας να διαχειρίζεται την 'ξαφνική' έξοδο και είσοδο από την μία κατάσταση του προγράμματος σε μία άλλη και πίσω χωρίς να οδηγείται σε απρόβλεπτη συμπεριφορά. Το feature αυτό χρειαζόταν για εφαρμογές που ήθελαν να έχουν πολλαπλές καταστάσεις ταυτόχρονα, αλλά μέχρι στιγμής δεν μπορούσαν.

Η ίδια έκδοση επίσης εισήγαγε το for loop, για πρώτη φορά μετά από 7 χρόνια. Το feature αυτό ήταν στην κορυφή των ζητούμενων features από τους χρήστες και αποτελούσε σχετικά μεγάλο ποσοστό των emails που λάμβαναν οι δημιουργοί. Ο λόγος που δίνουν για την καθυστέρηση του for loop είναι επειδή το while loop είναι πιο γενικό. Το γεγονός αυτό δεν σταμάτησε την εισαγωγή του for loop καθώς, κατά τα λεγόμενα των δημιουργών, «οι χρήστες παραπονιόντουσαν γιατί ξεχνούσαν συνέχεια να ενημερώνουν τη μεταβλητή ελέγχου στο τέλος των while loops, οδηγώντας το πρόγραμμα να κολλάει σε άπειρα loops».

ΕΚΔΟΣΗ 5

Στο «Lua Library Design Workshop» στο Harvard αποφασιστικέ ότι το επόμενο μεγάλο feature που χρειαζόταν η Lua ήταν ένα σύστημα υποστηρίξης modules. Τα modules υλοποιήθηκαν με την μόνη δομή δεδομένων της Lua, τα

tables. Το feature αυτό αποτέλεσε το μόνο σημαντικό feature που προστέθηκε με την έκδοση 5.0.

Ο κυριότερος σκοπός για την έκδοση 5.1 ήταν η προσθήκη incremental garbage collection, ένα feature πολυζήτητο από τους game developers, οι οποίοι είχαν πρόβλημα με τις μεγάλες παύσεις που παρουσιάζονταν όταν ο τότε μοναδικός generational garbage collector διέκοπτε τη ροή του προγράμματος για να γίνει το garbage collection.

Ανάμεσα στην έκδοση 5.0 και 5.1, όμως, οι χρήστες είχαν χρόνο να εξοικειωθούν με την δημιουργία δικών τους modules, που οδήγησε στο επόμενο διεθνές workshop να έχει ως κέντρο τις παρατηρήσεις των χρηστών με τον διαχειριστή modules. Το συμπέρασμα που βγήκε ήταν πως χρειαζόντουσαν μερικές τροποποιήσεις και, χωρίς η Lua να χάσει το κύριο χαρακτηριστικό της ως μία γλώσσα φτιαγμένη με κέντρο τους «μηχανισμούς, όχι πολιτικές», η δημιουργία ενός οδηγού χρήσης των modules.

Ακολουθεί εικόνα με τις λειτουργίες της Lua από έκδοση σε έκδοση.

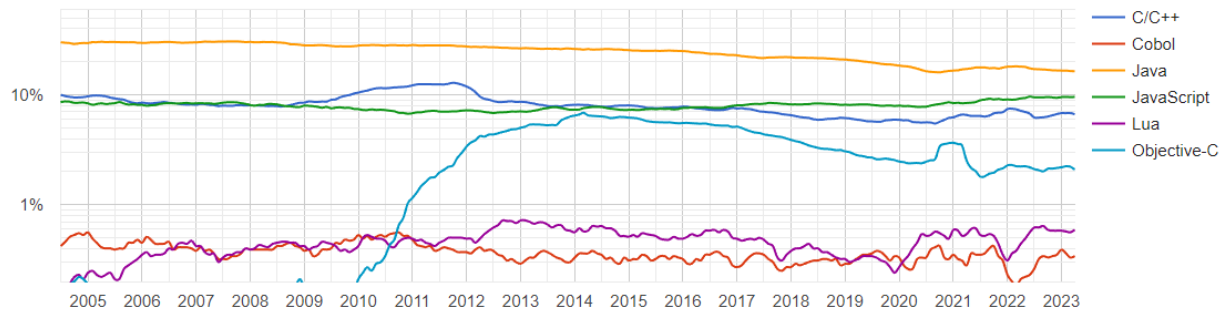
	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
constructors	●	●	●	●	●	●	●	●	●	●	●	●
garbage collection	●	●	●	●	●	●	●	●	●	●	●	●
extensible semantics	○	○	●	●	●	●	●	●	●	●	●	●
support for OOP	○	○	●	●	●	●	●	●	●	●	●	●
long strings	○	○	○	●	●	●	●	●	●	●	●	●
debug API	○	○	○	●	●	●	●	●	●	●	●	●
external compiler	○	○	○	○	●	●	●	●	●	●	●	●
vararg functions	○	○	○	○	○	●	●	●	●	●	●	●
pattern matching	○	○	○	○	○	●	●	●	●	●	●	●
conditional compilation	○	○	○	○	○	○	●	●	●	○	○	○
anonymous functions, closures	○	○	○	○	○	○	○	●	●	●	●	●
debug library	○	○	○	○	○	○	○	○	●	●	●	●
multi-state API	○	○	○	○	○	○	○	○	○	●	●	●
for statement	○	○	○	○	○	○	○	○	○	●	●	●
long comments	○	○	○	○	○	○	○	○	○	○	●	●
full lexical scoping	○	○	○	○	○	○	○	○	○	○	●	●
booleans	○	○	○	○	○	○	○	○	○	○	●	●
coroutines	○	○	○	○	○	○	○	○	○	○	●	●
incremental garbage collection	○	○	○	○	○	○	○	○	○	○	○	●
module system	○	○	○	○	○	○	○	○	○	○	○	●

Features over Versions

Μερίδιο Αγοράς

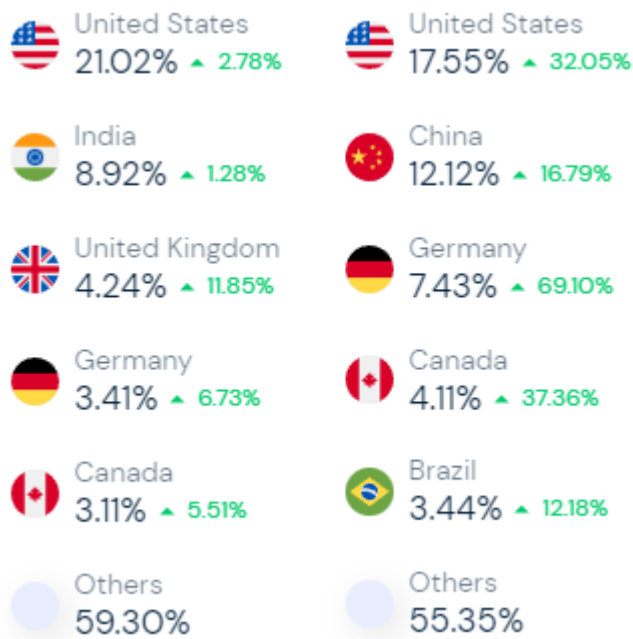
Ακολουθεί διάγραμμα με το ποσοστό χρήσης της Lua σε σύγκριση με γνωστές γλώσσες. Το διάγραμμα είναι λογαριθμικό για να φανούν καλύτερα οι αυξομειώσεις

στην δημοφιλία της γλώσσας, με την ελάχιστη και τη μέγιστη δημοφιλία το 2005 (0.2%) και το 2013 (0.7%) αντίστοιχα.



<https://pypl.github.io/PYPL.html>





Αξιοσημείωτα είναι και τα στατιστικά της κύριας ιστοσελίδας της Lua (Lua.org).



Οι ιστορικές ρίζες της Lua γίνονται εμφανείς από το συγκριτικά μεγάλο ποσοστό χρηστών από την Βραζιλία.

Στατιστικά StackOverflow (Αριστερά) και Lua (Δεξιά) ανά χώρα

Παρόμοιες ιστοσελίδες με τη Lua.org μας δείχνουν πως η Lua ανταγωνίζεται περισσότερο με τις γλώσσες υψηλού επιπέδου (Python, Ruby etc.), το οποίο ήταν αναμενόμενο.

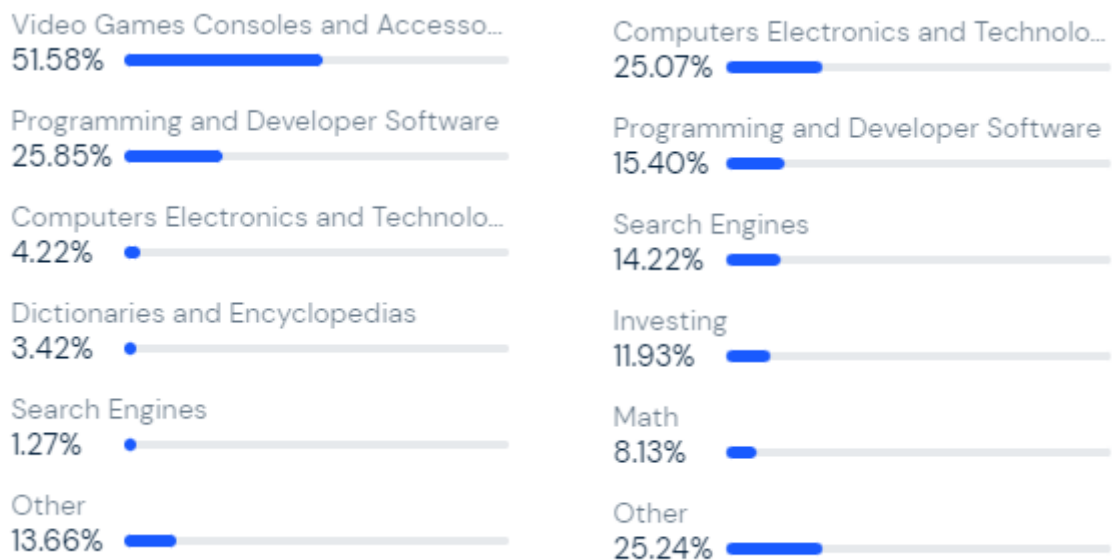
Site	Affinity	Monthly visits	Category	Category rank
 lua-users.org	100% 	81.5K	Computers Electronics and Technology > Programming and Developer Software	#12,195
 ruby-lang.org	74% 	372.5K	Computers Electronics and Technology > Programming and Developer Software	#3,891
 luaforge.net	74% 	11.2K	Computers Electronics and Technology > Programming and Developer Software	#46,444
 lua.gts-stolberg.de	74% 	--	Computers Electronics and Technology > Programming and Developer Software	--
 python.org	73% 	16.8M	Computers Electronics and Technology > Programming and Developer Software	#122
 luajit.org	70% 	17.1K	Computers Electronics and Technology > Programming and Developer Software	#32,582
 keplerproject.org	69% 	569	Computers Electronics and Technology > Programming and Developer Software	#122,824
 perl.org	63% 	192.5K	Computers Electronics and Technology > Programming and Developer Software	#7,140
 scala-lang.org	63% 	377.8K	Computers Electronics and Technology > Programming and Developer Software	#3,778
 erlang.org	62% 	122.1K	Computers Electronics and Technology > Programming and Developer Software	#9,597

<https://www.similarweb.com/website/lua.org>

Ως προς το μερίδιο αγοράς σύμφωνα με την Tiobe η Lua είναι 23^η με το 0.33% ανάμεσα στην Lisp και την Julia

23	Objective-C	0.54%
24	(Visual) FoxPro	0.51%
25	Perl	0.44%
26	F#	0.43%
27	COBOL	0.41%
28	Dart	0.38%
29	Lisp	0.35%
30	Lua	0.33%
31	Julia	0.30%
32	Scala	0.30%
33	Transact-SQL	0.29%
34	Haskell	0.29%

<https://www.tiobe.com/tiobe-index/>



Ακολουθούν διαγράμματα ποσοστού χρήσης ανά τομέα Lua (Αριστερά) και C++ (Δεξιά) για σύγκριση.

Παρατηρούμε ότι η Lua έχει την πλειονότητα του μεριδίου χρηστών της στα βιντεοπαιχνίδια και στους παραπλήσιους τομείς. Για το γεγονός αυτό ευθύνεται η εύκολη ενσωμάτωση της Lua, σε συνδυασμό με την υψηλή ταχύτητα που παρέχει κατά την εκτέλεση του προγράμματος. Τα χαρακτηριστικά είναι απαραίτητα στα βιντεοπαιχνίδια, όπου η ροή του προγράμματος πρέπει να λαμβάνει υπόψη τις κινήσεις του χρήστη από στιγμή σε στιγμή χωρίς να θυσιάζεται η απόδοση του προγράμματος.

Προδιαγραφές της Lua

Η Lua είναι μια ελαφριά, υψηλού επιπέδου γλώσσα δέσμης ενεργειών που συχνά ενσωματώνεται σε web apps, mobile apps, games και χρησιμοποιείται για τη δημιουργία σεναρίων και την επέκταση της λειτουργικότητάς τους.

Η σύνταξη της Lua είναι εμπνευσμένη από διάφορες γλώσσες προγραμματισμού, συμπεριλαμβανομένων των Pascal και C, δίνοντας έμφαση στην απλότητα.

Χρησιμοποιεί έναν συνδυασμό από keywords, symbols, και conventions για να ορίσει τη δομή του κώδικα.

Το πιο απλό πρόγραμμα lua, δηλαδή το γνωστό πρόγραμμα HelloWorld, παίρνει την εξής μορφή:

```
1  --This is how to do a classic Hello World script in Lua
2  print("Hello World!")
```

ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

Ως τύπους δεδομένων ορίζουμε τις διάφορες μορφές με τις οποίες μια γλώσσα προγραμματισμού αντιλαμβάνεται τις τιμές που διαθέτει ένα πρόγραμμα και καθορίζει τις πιθανές τιμές για αυτόν τον τύπο, τις λειτουργίες που μπορούν να επιτελεστούν σε τιμές αυτού του τύπου, την σημασία των δεδομένων και τον τρόπο που οι τιμές αυτού του τύπου μπορούν να αποθηκευτούν.

Στην γλώσσα προγραμματισμού Lua, οι τύποι δεδομένων που διαθέτουμε είναι οι παρακάτω:

- ❖ Nil: Ο τύπος Nil αντιπροσωπεύει την απουσία τιμής ή μη αρχικοποιημένων μεταβλητών.
- ❖ Boolean: Η Lua έχει δύο δυαδικές τιμές: true και false. Τα Booleans χρησιμοποιούνται σε λογικές πράξεις και συνθήκες.
- ❖ Number: Η Lua υποστηρίζει τόσο ακέραιους όσο και αριθμούς κινητής υποδιαστολής. Οι αριθμητικές τιμές μπορούν να χρησιμοποιηθούν σε αριθμητικές πράξεις.
- ❖ String: Οι συμβολοσειρές στην Lua είναι ακολουθίες χαρακτήρων που περικλείονται σε μονά εισαγωγικά ή διπλά εισαγωγικά. Η διαχείριση των συμβολοσειρών πραγματοποιείται μέσω function και operators της Lua
- ❖ Table: Οι πίνακες είναι η κύρια δομή δεδομένων της Lua και μπορούν να χρησιμοποιηθούν ως συσχετιστικοί πίνακες, λίστες, σύνολα ή αντικείμενα. Οι πίνακες είναι ζεύγη ή πίνακες κλειδιού-τιμής που μπορούν να αποθηκεύσουν οποιαδήποτε τιμή ως κλειδί ή τιμή.
- ❖ Function: Οι συναρτήσεις στην Lua είναι τιμές πρώτης κατηγορίας. Μπορούν να αντιστοιχιστούν σε μεταβλητές, να περάσουν ως ορίσματα και να επιστραφούν ως αποτελέσματα. Οι λειτουργίες επιτρέπουν την ενθυλάκωση ενός μπλοκ κώδικα για επαναχρησιμοποίηση.
- ❖ Thread: Η Lua υποστηρίζει ελαφρύ ταυτόχρονο προγραμματισμό μέσω νημάτων. Τα νήματα είναι ξεχωριστές ακολουθίες κώδικα που μπορούν να εκτελούνται ανεξάρτητα και να επικοινωνούν μεταξύ τους.
- ❖ UserData: Το UserData επιτρέπει στην Lua να διασυνδέεται με C ή άλλες γλώσσες προγραμματισμού. Παρέχει έναν τρόπο αποθήκευσης και χειρισμού αυθαίρετων δεδομένων C στην Lua.

- ❖ **Coroutine:** Οι κορουτίνες είναι παρόμοιες με τα νήματα, αλλά ελέγχονται ρητά από τον προγραμματιστή. Επιτρέπουν τη συνεργασία πολλαπλών εργασιών, όπου ο κώδικας μπορεί να αποδοθεί και να συνεχίσει την εκτέλεση.

Ο τύπος `nil` έχει μια ενιαία τιμή, το `nil`, της οποίας η κύρια ιδιότητα είναι να είναι διαφορετική από οποιαδήποτε άλλη τιμή, συχνά αντιπροσωπεύει την απουσία μιας χρήσιμης αξίας. Παρά το όνομά του, το `false` χρησιμοποιείται συχνά ως εναλλακτική του `nil`, με τη βασική διαφορά ότι το `false` συμπεριφέρεται σαν κανονική τιμή σε έναν πίνακα, ενώ το `nil` σε έναν πίνακα αντιπροσωπεύει ένα κλειδί που απουσιάζει.

Tables, functions, threads και (full) userdata values είναι objects, δηλαδή οι μεταβλητές στην πραγματικότητα δεν περιέχουν αυτές τις τιμές, παρά μόνο αναφορές σε αυτές. Η εκχώρηση, η μετάδοση παραμέτρων και οι επιστροφές συναρτήσεων χειρίζονται πάντα τις αναφορές σε τέτοιες τιμές. Αυτές οι λειτουργίες δεν συνεπάγονται κανενός είδους αντιγραφή.

ΜΕΤΑΒΛΗΤΕΣ

Οι μεταβλητές στην Lua δεν έχουν τύπους, μόνο οι τιμές των μεταβλητών έχουν. Δεν υπάρχουν ορισμοί τύπων στη γλώσσα. Όλες οι τιμές έχουν τον δικό τους τύπο.

Αυτό μπορούμε να το δούμε με ένα απλό παράδειγμα. Έστω ότι διαθέτουμε μια μεταβλητή με το όνομα `Label`, η Lua μας επιτρέπει να της περάσουμε και αριθμητικές τιμές αλλά και συμβολοσειρές όπως φαίνεται παρακάτω.

```
1 --In this example label is a variable of our program
2 Label = 1
3 print(Label)
4 --Variables in Lua do not have their own type, that means a certain variable, like label, can save different type of values in Lua
5 Label = "Marco"
6 print(Label)
7
8
```

Ονομασία Μεταβλητών

Τα ονόματα των μεταβλητών στην Lua μπορούν να αποτελούνται από γράμματα, ψηφία και κάτω παύλες (`_`). Οι μεταβλητές πρέπει να ξεκινούν με ένα γράμμα ή μια υπογράμμιση. Η Lua έχει διάκριση πεζών-κεφαλαίων, που σημαίνει ότι οι μεταβλητές `ex_Var` και `Ex_Var` θα θεωρούνται διακριτές. Η ονομασία των μεταβλητών δεν θα πρέπει να ξεκινάει με αριθμούς ή να είναι ίδια με δεσμευμένες λέξεις. Δεσμευμένες λέξεις στην Lua αποτελούν οι `and`, `break`, `do`, `else`, `elseif`, `end`, `false`, `for`, `function`, `goto`, `if`, `in`, `local`, `nil`, `not`, `or`, `repeat`, `return`, `then`, `true`, `until`, `while`.

Εκχώρηση τιμών

Η Lua χρησιμοποιεί τον τελεστή εκχώρησης '=' για να εκχωρήσει τιμές σε μεταβλητές. Οι μεταβλητές στην Lua έχουν ως αρχικοποίηση, (δηλαδή πριν την εκχώρηση κάποιας τιμής) την τιμή nil.

Εύρος Μεταβλητών

Οι μεταβλητές στην Lua έχουν lexical scoping. Εάν μια μεταβλητή έχει δηλωθεί σε ένα block ή function, είναι προσβάσιμη μόνο εντός αυτού του block ή function. Εάν δεν χρησιμοποιείται τοπική λέξη-κλειδί, οι μεταβλητές θεωρούνται global και είναι προσβάσιμες σε όλο το πρόγραμμα.

Τοπικές Μεταβλητές

Η τοπική λέξη-κλειδί 'local' στην Lua χρησιμοποιείται για τη δημιουργία μεταβλητών με τοπικό εύρος. Οι τοπικές μεταβλητές είναι προσβάσιμες μόνο εντός του μπλοκ στο οποίο δηλώνονται. Για παράδειγμα: "local var_local = 12".

Global Μεταβλητές

Οι μεταβλητές που δηλώνονται χωρίς την τοπική λέξη-κλειδί 'global' θεωρούνται καθολικές και μπορούν να προσπελαστούν από οποιοδήποτε μέρος του προγράμματος.

Κύκλος Ζωής Μεταβλητών

Οι τοπικές μεταβλητές έχουν περιορισμένο εύρος και καταστρέφονται κατά την έξοδο από το block στο οποίο έχουν δηλωθεί. Οι καθολικές μεταβλητές παραμένουν καθ' όλη τη διάρκεια της εκτέλεσης του προγράμματος.

Προσπέλαση Μεταβλητών

Οι μεταβλητές μπορούν να προσπελαστούν χρησιμοποιώντας τα ονόματά τους.

ΤΕΛΕΣΤΕΣ

Ως τελεστές ορίζουμε σύμβολα που βοηθούν την δημιουργία δομημένου προγραμματισμού και την εκτέλεση μαθηματικών και άλλων πράξεων μεταξύ τιμών. Οι παρακάτω τελεστές μπορούν να συνδυαστούν για να σχηματίσουν πιο σύνθετες εκφράσεις στο Lua. Η προτεραιότητα τελεστών καθορίζει τη σειρά με την οποία αξιολογούνται οι πράξεις. Οι παρενθέσεις '()' μπορούν να χρησιμοποιηθούν για τον έλεγχο της σειράς των πράξεων. Είναι σημαντικό να σημειωθεί ότι το Lua ακολουθεί τους τυπικούς κανόνες προτεραιότητας χειριστή.

Οι τελεστές στην Lua διαθέτουν τις παρακάτω κατηγοριοποιήσεις.

Αριθμητικοί Τελεστές

Πρόσθεση: '+'

Αφαίρεση: '-'

Πολλαπλασιασμός: ‘ * ’

Διαίρεση: ‘ / ’

Modulo (υπόλοιπο): ‘ % ’

Exponentiation(ύψωση σε δύναμη): ‘ ^ ’

Σχεσιακοί Τελεστές

Ισότητα: ‘ == ’

Ανισότητα: ‘ != ’ or ‘ ~= ’

Μεγαλύτερο από: ‘ > ’

Μικρότερο από: ‘ < ’

Μεγαλύτερο ή ίσο από: ‘ >= ’

Μικρότερο ή ίσο από: ‘ <= ’

Λογικοί Τελεστές

Logical AND: ‘ and ’

Logical OR: ‘ or ’

Logical NOT: ‘ not ’

Τελεστές Ανάθεσης

Simple assignment: ‘ = ’

Addition assignment: ‘ += ’

Subtraction assignment: ‘ -= ’

Multiplication assignment: ‘ *= ’

Division assignment: ‘ /= ’

Modulo assignment: ‘ %= ’

Exponentiation assignment: ‘ ^= ’

Δυφιοτελής Τελεστές

Bitwise AND: ‘ & ’

Bitwise OR: ‘ | ’

Bitwise XOR: ‘ ~ ’ (binary)

Bitwise NOT: ‘ ~ ’ (unary)

Left shift: ‘ << ’

Right shift: ‘ >> ’

Λοιποί Τελεστές

Length operator: ‘ # ’ (επιστρέφει το μήκος μιας συμβολοσειράς ή τον αριθμό των στοιχείων από έναν πίνακα)

Table indexing: ‘ [] ’ (χρησιμοποιείται για την πρόσβαση σε στοιχεία ενός πίνακα με ευρετήριο ή κλειδί)

String Concatenation Operator: Ο concatenation operator ‘ .. ’ χρησιμοποιείται για τη σύνδεση δύο string.

ΕΠΑΝΑΛΗΠΤΕΣ

Στην Lua, οι iterators είναι objects ή functions που επιτρέπουν της προσπέλαση μιας συλλογής τιμών.

Ipairs

Ο ‘`ipairs`’ iterator χρησιμοποιείται για την προσπέλαση στοιχείων ενός array-like table.

Επιστρέφει δύο values: το index και την αντίστοιχη τιμή κάθε στοιχείου του table.

Pairs

Ο ‘`pairs`’ iterator χρησιμοποιείται για την προσπέλαση ζευγών κλειδιού-τιμής ενός πίνακα, συμπεριλαμβανομένων των μη αριθμητικών κλειδιών.

Επιστρέφει δύο values: το key και την αντίστοιχη τιμή κάθε στοιχείου του table.

String.gmatch

Η συνάρτηση `string.gmatch` επιτρέπει την προσπέλαση σε substrings ενός string τα οποία διαχωρίζονται βάση ενός δεδομένου μοτίβου.

Επιστρέφει μια συνάρτηση iterator που, όταν καλείται, επιστρέφει την επόμενη αντίστοιχη υποσυμβολοσειρά.

«Τεχνητοί» Επαναληπτές

Η Lua δίνει την δυνατότητα κατασκευής προσαρμοσμένων iterators με την χρήση functions και closures. Όμως μια προσαρμοσμένη συνάρτηση iterator για να μπορέσει να λειτουργήσει θα πρέπει να επιστρέφει την επόμενη τιμή κάθε φορά που καλείται και να παρακολουθεί την κατάσταση της εσωτερικά.

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ως δομές δεδομένων αναφερόμαστε στους διάφορους τρόπους με τους οποίους μπορούμε να οργανώσουμε τα δεδομένα ενός προγράμματος, ώστε να μπορούν να χρησιμοποιηθούν αποδοτικά.

Πίνακας (Table)

Οι πίνακες είναι η κύρια (και μοναδική που διαθέτει από μόνη της η Lua) δομή δεδομένων της Lua και μπορούν να χρησιμοποιηθούν ως arrays, dictionaries, sets, ή objects. Μπορούν να αποθηκεύουν τιμές με αυθαίρετα κλειδιά και να παρέχουν ευέλικτες δυνατότητες ευρετηρίασης. Επιπλέον οι πίνακες μπορούν να είναι nested, επιτρέποντας τη δημιουργία πολύπλοκων δομών δεδομένων ή να χρησιμοποιηθούν για την αναπαράσταση άλλων δομών δεδομένων, όπως lists, maps, graphs.

Array

Τα arrays στην Lua υλοποιούνται χρησιμοποιώντας tables. Arrays είναι ακολουθίες τιμών που αποθηκεύονται σε διαδοχικά ακέραια κλειδιά ξεκινώντας από το 1. Η Lua

παρέχει ενσωματωμένες λειτουργίες όπως ' `table.insert` ' και ' `table.remove` ' για αποτελεσματικό χειρισμό πινάκων.

Linked List

Οι συνδεδεμένες λίστες μπορούν να υλοποιηθούν στην Lua χρησιμοποιώντας tables με συγκεκριμένη δομή. Κάθε κόμβος στη λίστα είναι ένα table που περιέχει την τιμή και μια αναφορά στον επόμενο κόμβο.

Queue

Οι ουρές μπορούν να υλοποιηθούν χρησιμοποιώντας tables, όπου τα στοιχεία εισάγονται στο ένα άκρο και αφαιρούνται από το άλλο άκρο. Η Lua παρέχει ενσωματωμένες λειτουργίες όπως ' `table.insert` ' και ' `table.remove` ' για την υλοποίηση ουρών.

Stack

Οι στοίβες μπορούν να υλοποιηθούν χρησιμοποιώντας tables, όπου τα στοιχεία εισάγονται και αφαιρούνται από το ίδιο άκρο (Last In, First Out). Η Lua παρέχει ενσωματωμένες λειτουργίες όπως ' `table.insert` ' και ' `table.remove` ' για την υλοποίηση στοίβων.

Set

Τα σύνολα μπορούν να υλοποιηθούν χρησιμοποιώντας tables, όπου κάθε στοιχείο αποθηκεύεται ως κλειδί στον πίνακα. Τα keys ενός table της Lua μπορούν να χρησιμοποιηθούν για τον αποτελεσματικό προσδιορισμό της παρουσίας ή της απουσίας ενός στοιχείου.

ΑΝΑΦΟΡΕΣ

Στην Lua οι μεταβλητές περιέχουν αναφορές σε τιμές και όχι αντίγραφα τους, δίνοντας την δυνατότητα άμεσης διαχείρισης των τιμών. Οι αναθέσεις και οι κλήσεις συναρτήσεων στην Lua λειτουργούν μεταβιβάζοντας αναφορές σε τιμές. Όταν εκχωρείται μια τιμή σε μια μεταβλητή ή τη μεταβιβάζεται ως όρισμα σε μια συνάρτηση, μεταβιβάζεται μια αναφορά στην υποκείμενη τιμή. Όταν εκχωρείται μια τιμή σε μια νέα μεταβλητή ή όταν μεταβιβάζεται ως όρισμα συνάρτησης, δεν αντιγράφεται η ίδια η τιμή αλλά η αντίστοιχη αναφορά σε αυτή. Ως αποτέλεσμα, και οι δύο μεταβλητές ή ορίσματα αναφέρονται στην ίδια τιμή.

Οι πίνακες στην Lua μεταβιβάζονται με αναφορά, που σημαίνει ότι η αντιστοίχιση ενός πίνακα σε μια νέα μεταβλητή ή η μετάβασή του σε μια συνάρτηση δημιουργεί μια αναφορά στον ίδιο πίνακα. Η τροποποίηση του πίνακα μέσω οποιασδήποτε αναφοράς επηρεάζει όλες τις αναφορές σε αυτόν τον πίνακα.

Η Lua χρησιμοποιεί έναν συλλέκτη σκουπιδιών για να διαχειριστεί τη μνήμη. Εντοπίζει αυτόματα πότε οι τιμές δεν αναφέρονται πλέον και ελευθερώνει τη σχετική

μνήμη. Αυτό επιτρέπει στον προγραμματιστή να εστιάσει στον χειρισμό τιμών χωρίς να ανησυχεί για τη μη αυτόματη διαχείριση της μνήμης. Για την δημιουργία όμως ενός ανεξάρτητου αντίγραφου ενός πίνακα ή άλλης σύνθετης τιμής, η Lua παρέχει λειτουργίες όπως για παράδειγμα ‘table.copy’ για τη δημιουργία ενός νέου πίνακα με ξεχωριστές αναφορές σε τιμές. Αυτό το σύστημα επιτρέπει την αποτελεσματική χρήση της μνήμης, την εύκολη κοινή χρήση τιμών και τον ευέλικτο χειρισμό τιμών.

ΔΟΜΕΣ ΕΛΕΓΧΟΥ

Οι δομές ελέγχου διευκολύνουν τον έλεγχο της ροής εκτέλεσης των προγραμμάτων Lua, την λήψη αποφάσεων με βάση τις συνθήκες, την επανάληψη block κώδικα και την αποτελεσματική διαχείριση σφαλμάτων.

Υποθετικές Δομές

‘if-then ‘ statement: Εκτελεί ένα block κώδικα εάν μια συγκεκριμένη συνθήκη είναι αληθής.

‘if-then-else ‘ statement: Εκτελεί ένα block κώδικα εάν μια συνθήκη είναι αληθής και ένα άλλο μπλοκ εάν η συνθήκη είναι ψευδής.

‘if-then-elseif-else ‘ statement: Επιτρέπει τη διαδοχική δοκιμή πολλαπλών συνθηκών.

Ας δώσουμε ένα παράδειγμα χρήσης υποθετικών δομών. Ένα πολύ απλό πρόγραμμα που χρησιμοποιεί υποθετικές δομές είναι το λεγόμενο FizzBuzz. Το πρόγραμμα αυτό παίρνει ως όρισμα έναν ακέραιο και εκτυπώνει:

- ❖ Fizz αν διαιρείται μόνο με το 2
- ❖ Buzz αν διαιρείται μόνο με το 3
- ❖ FizzBuzz αν διαιρείται και με το 2 και με το 3
- ❖ Nil (ένα string Nil όχι τον τύπο δεδομένων) άμα δεν διαιρείται ούτε με το 2 ούτε με το 3

Η διαδικασία αυτή αναπαράγεται πολύ εύκολα με την χρήση υποθετικών τελεστών ως εξής:

```
--This line says to our program that we will read a number (*n) from the console and save it to the Input variable
print("Insert a number:")
Input = io.read("*n")
--if statements in Lua use the keywords if/then/elseif/else/end
if(Input % 2 == 0)
then
    if(Input % 3 == 0)
    then
        print("FizzBuzz")
    else
        print("Fizz")
    end...
else
    if(Input % 3 == 0)
    then
        print("Buzz")
    else
        print("Nil")
    end
end...
end...
```

Δομές Επανάληψης

- ‘ while ‘ loop: Επαναλαμβάνει ένα block κώδικα εφόσον μια συνθήκη είναι αληθής.
- ‘ repeat-until ‘ loop: Επαναλαμβάνει ένα block κώδικα έως ότου μια συνθήκη είναι αληθής.
- ‘ for ‘ loop: Προσπελάζει ένα εύρος τιμών.
- ‘ for-in ‘ loop: Προσπελάζει τα στοιχεία ενός table ή τους χαρακτήρες ενός string.

Ας δώσουμε ένα παράδειγμα χρήσης επαναληπτικών δομών. Έστω ότι θέλαμε να εκτυπώσουμε στην κονσόλα ένα απλό δέντρο με αριστερή στοίχιση που να έχει συνολικά n σειρές (το οποίο αποτελεί και το input του προγράμματος) και σε κάθε σειρά με αρίθμηση i να εκτυπώνει i χαρακτήρες «i».

Αυτό το πρόγραμμα με επαναληπτικές δομές στην γλώσσα Lua γράφεται ως εξής:

```
function TreeA (input)
    print("Printing Tree A:")
    --numeral for loops in Lua are stated as: for variable=start,end,step do (...) end, with default step=1
    for i=1,input do
        for j = 1,i do
            --We use io.write instead of print in this case, because the print function always inserts a \n character at the end
            io.write("i ")
        end
        io.write("\n")
    end
    return
end
```

Διαχείριση Λαθών (Error Handling)

- ‘ assert ‘ function: Ελέγχει μια συνθήκη και στέλνει ένα σφάλμα εάν η συνθήκη είναι ψευδής.
- ‘ pcall ‘ function: Εκτελεί μια συνάρτηση και εντοπίζει τυχόν σφάλματα που προκύπτουν, επιτρέποντάς μια πιο οργανωμένη και γρήγορη αντιμετώπιση τους.

Διαχείριση Ροής

- ‘ break ‘ statement: Τερματίζει την εκτέλεση της πιο εσωτερικής εντολής βρόχου ή διακόπτη.
- ‘ return ‘ statement: Έξοδος από μια συνάρτηση με τη δυνατότητα επιστροφής μιας τιμής.
- ‘ goto ‘ statement: Μεταφέρει τη ροή ελέγχου σε μια δήλωση με ετικέτα στον κώδικα.

INTEROPERABILITY

Το Lua μπορεί εύκολα να ενσωματωθεί σε άλλες εφαρμογές ως γλώσσα δέσμης ενεργειών και παρέχει μηχανισμούς επικοινωνίας με την εφαρμογή υποδοχής, συμπεριλαμβανομένης της πρόσβασης σε API κεντρικού υπολογιστή και δομές δεδομένων.

ΣΥΝΑΡΤΗΣΙΑΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Συναρτήσεις

Στην γλώσσα προγραμματισμού Lua οι περισσότερες διεργασίες γίνονται μέσω συναρτήσεων (functions). Ειδικότερα, στην Lua οι συναρτήσεις είτε μπορούν να εκτελούν μία συγκεκριμένη διαδικασία, όπως για παράδειγμα μια κατάταξη τιμών, είτε μπορούν να διαβάζουν και να επιστρέφουν τιμές. Συνεπώς, οι συναρτήσεις στην Lua μπορούν και να καλούνται κατάλληλα και να λαμβάνονται ως ορίσματα.

Όταν καλούμε μία ήδη ορισμένη συνάρτηση στην Lua, είτε συνάρτηση βιβλιοθήκης είτε συνάρτηση που όρισε πιο πάνω ο προγραμματιστής, γράφουμε το όνομά της και μέσα σε παρενθέσεις τα ορίσματα με τα οποία καλείται.

Στην Lua μια συνάρτηση μπορεί να κληθεί με διαφορετικό αριθμό ορισμάτων από τον αριθμό των παραμέτρων της. Στην περίπτωση που δοθούν λιγότερα από τα απαιτούμενα ορίσματα κάθε επιπλέον παράμετρος παίρνει την τιμή `nil`. Στο αντίθετο σενάριο, κάθε επιπλέον όρισμα απορρίπτεται από την συνάρτηση.

Πέρα από πολλά ορίσματα, οι συναρτήσεις στην Lua μπορούν να επιστρέφουν πολλαπλές τιμές, χωρίς να βρίσκονται σε μορφή δομής ή να είναι του ίδιου τύπου. Ο προγραμματιστής μπορεί να ορίσει με την λέξη-κλειδί `return` ποιες τιμές τοπικών μεταβλητών να επιστραφούν. Όταν καλείται μια συνάρτηση οι τιμές που επιστρέφει μπορούν είτε να αποθηκευτούν σε μεταβλητές, είτε να χρησιμοποιηθούν από το πρόγραμμα είτε και να απορριφτούν από τον κώδικα. Στην περίπτωση που η κλίση μιας συνάρτησης είναι κλεισμένο μέσα σε παρενθέσεις, λόγω της χρήσης από μια συνάρτηση, τότε λαμβάνεται μόνο η πρώτη επιστρεφόμενη τιμή, αν υπάρχει.

Παρότι στην Lua το σύστημα ανάθεσης τιμών σε τοπικές μεταβλητές της συνάρτησης είναι θεσιακό, δηλαδή αντιστοιχίζει την τιμή μιας μεταβλητής με βάση την θέση της τιμής στα ορίσματα, κάποιες φορές είναι χρήσιμο να δίνουμε ονόματα στις παραμέτρους της συνάρτησης. Η Lua δεν διαθέτει άμεση υποστήριξη για κάτι τέτοιο ωστόσο είναι δυνατόν να εφαρμοστεί χρησιμοποιώντας ως όρισμα έναν πίνακα, τον οποίο εφαρμόζουμε ως παράμετρο της συνάρτησης, και δίνοντας ονόματα στα κλειδιά του πίνακα.

Ένα χαρακτηριστικό παράδειγμα χρήσης των συναρτήσεων σε μια γλώσσα προγραμματισμού είναι για παράδειγμα οι συναρτήσεις `sorting`. Έστω λοιπόν ότι θέλαμε να φτιάξουμε συναρτήσεις για να κατατάξουμε ένα `array`, που όπως είπαμε μπορεί να υλοποιηθεί μέσω πινάκων, μια επιλογή θα ήταν να φτιάξουμε μια απλή συνάρτηση `Selection Sort`, η οποία θα έπαιρνε ως όρισμα το `array` προς `sorting` και θα έκανε την διαδικασία της κατάταξης των στοιχείων **χωρίς** να χρειάζεται να επιστρέφει κάποιο όρισμα, καθώς οι θέσεις μνήμης του `array` δεν θα συλλεχθούν από τον `Garbage Collector`.

```

function SelectionSort(arr)
    local arr_size = #arr
    for i = 1, arr_size-1 do
        local id = i
        for j = i+1, arr_size do
            if arr[j] < arr[id] then
                id = j
            end
        end
        --Swap Values
        arr[i], arr[id] = arr[id], arr[i]
    end
end

```

Πέρα από απλές συναρτήσεις είναι σημαντικό να αναφέρουμε ότι στην Lua ότι μπορούμε να κάνουμε και αναδρομικές συναρτήσεις, το οποίο στο context των συναρτήσεων sorting μπορούμε να το δείξουμε με την πιο αποδοτική Quicksort (όπως και τις βοηθητικές συναρτήσεις με την partition)

```

--Helper function for QuickSort
function partition(arr, left, right)
    pivot = arr[right]
    i = left - 1
    for j=left,right-1 do
        if(arr[j] < pivot) then
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
        end
    end
    arr[i+1], arr[right] = arr[right], arr[i+1]
    return i+1
end

function Quicksort(arr, left, right)
    if right > left then
        local pivotNewKey = partition(arr, left, right)
        Quicksort(arr, left, pivotNewKey - 1)
        Quicksort(arr, pivotNewKey + 1, right)
    end
end

```

Μεταπίνακες και Μεταμέθοδοι

Συνήθως, στην Lua οι πίνακες, όπως είδαμε παραπάνω, έχουν ένα αρκετά προβλέψιμο σύνολο πράξεων και δεν έχουν από μόνοι τους δυνατότητες όπως σύγκριση ή πρόσθεση πινάκων. Αυτό γίνεται δυνατό με την χρήση μεταπινάκων, δηλαδή μια οντότητα που μπορεί να συνδέσει δύο διαφορετικούς πίνακες. Οι μεταπίνακες έχουν την δυνατότητα να αλλάξουν την συμπεριφορά ενός πίνακα με βάση έναν άλλο πίνακα.

Για τους μεταπίνακες υπάρχουν ειδικά κλειδιά που μπορούμε να εισάγουμε για να δώσουμε ειδικευμένη χρήση καλώντας συγκεκριμένες συναρτήσεις που ονομάζουμε μεταμεθόδους.

Κάποιες βασικές μεταμέθοδοι είναι:

- ❖ `__index`: Στην περίπτωση που στον κανονικό πίνακα `table` ψάξουμε να βρούμε την τιμή με κλειδί `key`, τότε ελέγχουμε τι τιμή έχει η μεταβλητή `__index` του μεταπίνακα. Άμα είναι συνάρτηση καλείται με ορίσματα τον πίνακα και το κλειδί, ενώ αν είναι πίνακας επιστρέφει το `value[key]`
- ❖ `__len`: Κάθε φορά που ζητάμε το μήκος ενός πίνακα με μεταπίνακα που έχει το κλειδί `__len` στο όρισμα της τότε καλείται η συνάρτηση που είναι αποθηκευμένη στο προαναφερόμενο κλειδί.
- ❖ `__gc`: Η συνάρτηση που είναι αποθηκευμένη στο κλειδί αυτό καλείται, ως `destroy_value function`, για τον πίνακα πριν συλλεχθεί από τον GC
- ❖ `__add`: Όταν προσπαθήσει ο προγραμματιστής να εκτελέσει πρόσθεση μεταξύ του πίνακα με μεταπίνακα που έχει αυτό το κλειδί, και ένα άλλο αντικείμενο, τότε καλείται η συνάρτηση που είναι αποθηκευμένη στο κλειδί του μεταπίνακα με ορίσματα τον πίνακα και το αντικείμενο. (αντίστοιχα λειτουργούν και οι `__sub`, `__mul`, `__div` κτλ. με τις αντίστοιχες πράξεις)

Για παράδειγμα, έστω ότι θέλαμε να εφαρμόσουμε την πρόσθεση μεταξύ δύο πινάκων. Αυτό που θα κάναμε είναι δημιουργούσαμε έναν πίνακα με το μοναδικό του περιεχόμενο την μεταβλητή `__add` με τιμή μια συνάρτηση για να προσθέτουμε τα στοιχεία του δεύτερου πίνακα στον πρώτο.

```

1  table = setmetatable({ 10, 22, 49 }, {
2      __add = function(table, table_to_add)
3
4          for i = 1, table.maxn(table_to_add) do
5              table.insert(table, table.maxn(table)+1, table_to_add[i])
6          end
7          return table
8      end
9  })

```

Κορουτίνες

Η Κορουτίνη στον προγραμματισμό είναι αντίστοιχη με τα νήματα με την έννοια ότι αντιπροσωπεύει μια δικιά της γραμμή εκτέλεσης, με δικιά της στοίβα, δικές της τοπικές μεταβλητές και με τον δικό της δείκτη εντολών με την μόνη διαφοροποίηση σε σχέση με την πολυνηματοποίηση να είναι ότι οι κορουτίνες μπορούν να συνεργάζονται μεταξύ τους.

Η Lua τοποθετεί όλες τις κορουτίνες του προγράμματος στον πίνακα κορουτίνων που διαθέτει. Μια νέα κορουτίνη δημιουργείται μέσω της εντολής `coroutine.create()` της οποίας το αποτέλεσμα αποθηκεύουμε σε μια μεταβλητή. Μια κορουτίνη διαθέτει τρεις καταστάσεις: αναστολή, που είναι η αρχική κατάσταση κάθε κορουτίνας και σημαίνει ότι δεν τρέχει αυτή την στιγμή, εκτέλεση, στην οποία φτάνουμε με την εντολή `coroutine.resume(<name_of_the_coroutine>)` και σημαίνει ότι μέχρι να καλεστεί η εντολή `coroutine.yield()` εντός της κορουτίνας αυτής θα εκτελείται κανονικά, και νεκρή.

Το πλέον παραδειγματικό πρόβλημα που χρησιμοποιεί κορουτίνες είναι το πρόβλημα του καταστήματος, δηλαδή ένα πρόβλημα που περιλαμβάνει μια συνάρτηση καταναλωτής, που συνεχώς ξοδεύει τιμές, και μια συνάρτηση παραγωγός, που δημιουργεί συνεχώς τιμές.

Για την δημιουργία του παραπάνω κώδικα χρησιμοποιούμε δύο concept μοναδικά στις κορουτίνες τον σωλήνα, δηλαδή μια ανεξαρτησία μεταξύ κάθε κορουτίνας, και φίλτρα, που είναι ουσιαστικά μια συνάρτηση ελέγχου και διαχείρισης της σχέσης μεταξύ των δύο «οντοτήτων».

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Η Lua υποστηρίζει αντικειμενοστραφή προγραμματισμό έμμεσα. Ο λόγος που λέμε έμμεσα είναι διότι στην γλώσσα δεν είναι ορισμένα πολλά από τα βασικά concept του αντικειμενοστραφούς προγραμματισμού, όπως οι κλάσεις, η κληρονομικότητα, ο πολυμορφισμός κτλ. Συνεπώς, τίθεται το ερώτημα πως μπορούμε υπό αυτές τις συνθήκες να μιλήσουμε για αντικειμενοστραφή προγραμματισμό;

Η απάντηση βρίσκεται στην μοναδική δομή που μας προσφέρει η Lua, δηλαδή τους πίνακες. Ένας πίνακας μπορεί να χρησιμοποιηθεί για να αντιπροσωπεύει ένα αντικείμενο στον αντικειμενοστραφή προγραμματισμό, καθώς διαθέτει τα παρακάτω:

- ✓ Οι πίνακες έχουν κάποια κατάσταση, όπως τα αντικείμενα.
- ✓ Οι πίνακες έχουν ταυτότητα (selfness) που είναι ανεξάρτητη από την εκάστοτε τιμή των μεταβλητών τους.
- ✓ Οι πίνακες έχουν έναν κύκλο ζωής εξαρτημένο από το ποιος και πότε τους δημιουργήσει
- ✓ Μέσω των μεταπινάκων, οι πίνακες υποστηρίζουν πράξεις μεταξύ τους
- ✓ Επίσης μέσω των μεταπινάκων υποστηρίζουν και συναρτήσεις για κάθε πίνακα
- ✓ Τέλος, οι πίνακες έχουν και κατασκευαστές (constructors)

Στο κεφάλαιο αυτό θα εισάγουμε ένα παράδειγμα αντικειμενοστραφούς προβλήματος που θα μας βοηθήσει να κατανοήσουμε σε μεγαλύτερο βαθμό τις έννοιες που θα παρουσιαστούν στην συνέχεια μέσω κώδικα.

Έστω ότι έχουμε έναν φύλακα σε ζωολογικό κήπο. Ο φύλακας αυτός είναι υπεύθυνος για να κρατάει κάποιες πληροφορίες για κάθε ζώο όπως το όνομα του, τον αριθμό των ποδιών που έχει, την ηλικία του, την δίαιτα του (σαρκοφάγο ή χορτοφάγο), τον αριθμό του κλουβιού του και το συγκεκριμένο είδος του.

Στις ενότητες που ακολουθούν θα μελετήσουμε βήμα προς βήμα πως μεταφράζεται σε κώδικα Lua η αντικειμενοστραφής μορφή του προβλήματος αυτού.

Κλάσεις

Η Lua σαν γλώσσα προγραμματισμού, όπως αναφέρθηκε και παραπάνω, δεν διαθέτει την δυνατότητα δημιουργίας κλάσεων, δηλαδή πολύπλοκων τύπων βάση των οποίων ορίζουμε αντικείμενα. Ωστόσο, σε τέτοιου είδους γλώσσες, όπως είναι και η Self και η NewtonScript, μπορούμε να ορίσουμε ένα αντικείμενο πρότυπο προκειμένου να υποκαταστήσουμε όλες τις λειτουργίες μιας κλάσης.

Το ερώτημα όμως που ανέρχεται είναι: Πως χρησιμοποιούμε τα πρότυπα αυτά στην Lua;

Όπως είδαμε στο εισαγωγικό κομμάτι της ενότητας στην Lua τα αντικείμενα αντιπροσωπεύονται από έναν πίνακα, που διαθέτει τις δικές του τιμές, τις δικές του μεθόδους και τον δικό του κατασκευαστή. Συνεπώς, για να δημιουργήσουμε ένα νέο αντικείμενο με βάση το πρότυπο-κλάση που διαθέτουμε θέτουμε το πρότυπο ως μεταπίνακα του νέου αντικειμένου προκειμένου, όπως είδαμε στην <insert ενότητα>, να περασθούν όλα τα χαρακτηριστικά του προτύπου στο νέο αντικείμενο, αποκτώντας ωστόσο δική του ξεχωριστή ταυτότητα. Η ιδεολογία αυτή στον αντικειμενοστραφή προγραμματισμό εκφράζεται συνήθως με τον όρο πολυμορφισμός, ωστόσο στην Lua αποτελεί τον θεμέλιο λίθο όλου της Lua.

Επιστρέφοντας τώρα στο πρόβλημα που αναφέρθηκε στην εισαγωγή με τον φύλακα ζωολογικού κήπου το βήμα που καλούμαστε να κάνουμε είναι η δημιουργία της κλάσης Ζώο (Animal), η οποία θα είναι ένας πίνακας με τις μεταβλητές να αναφέρονται στα χαρακτηριστικά του ζώου που χρειαζόμαστε και οι τιμές που έχουν για αρχή να είναι Garbage Values.

```
-- We Create the Base Class Animal
--The values below are just placeholders and NOT the constant value of an Animal
Animal = { __name = "Diego", __numOfLegs = 4, __age = 1, __diet = Animal_Diet.HERBIVORE, __cageNumber = 1, __species = "Lion" }
```

Ωστόσο για να δημιουργήσουμε νέα αντικείμενα από αυτόν τον πίνακα-κλάση χρειάζεται να φτιάξουμε μια συνάρτηση constructor, η οποία, προκειμένου να διαθέτει το νέο αντικείμενο-πίνακας τις ιδιότητες της κλάσης Animal, θα κάνει τα παρακάτω:

- Θα δημιουργεί το νέο αντικείμενο ως κενό πίνακα εφόσον ο χρήστης δεν έδωσε ήδη ως όρισμα ένα αντικείμενο όρισμα
- Θα κάνει τον πίνακα Animal **Μεταπίνακες** και Μεταμέθοδοι του νέου αντικειμένου προκειμένου να κληρονομήσει τα χαρακτηριστικά του
- Θα κάνει self-indexing (Ο σκοπός αυτού αφορά την λειτουργία των πινάκων στην Lua και γίνεται σε κάθε συνάρτηση κατασκευαστή)
- Θα θέσει κάθε μεταβλητή του νέου αντικειμένου είτε ως το δοσμένο όρισμα για αυτήν είτε, άμα το όρισμα είναι nil, στην default τιμή **Unknown**.
- Θα επιστρέψει το αντικείμενο


```

function Animal:create (obj, name, numOfLegs, age, diet, cageNumber, species)
    --obj essentially serves as the new object that is created from the class Animal
    obj = obj or {}
    setmetatable(obj, self)
    self.__index = self
    --The or represents the default values that will be applied if the inserted value is nil
    self.__name = name or "Unknown"
    self.__age = age or "Unknown"
    self.__numOfLegs = numOfLegs or "Unknown"
    self.__cageNumber = cageNumber or "Unknown"
    self.__diet = diet or "Unknown"
    self.__species = species or "Unknown"
    return obj
end

```

Κληρονομικότητα

Όπως είδαμε στην Lua κάθε κλάση είναι ένα αντικείμενο στο περιβάλλον της. Αυτό καθιστά την κληρονομικότητα, μια από τις τέσσερις θεμελιώδεις ιδιότητες του αντικειμενοστραφούς προγραμματισμού, αρκετά εύκολη να γίνει στο περιβάλλον της Lua.

Πιο συγκεκριμένα, όπως αναφέραμε οι δομές των κλάσεων και των αντικειμένων στην Lua δεν διαθέτουν καμία δομική διαφορά, καθώς και τα δύο αποτελούν πίνακες. Συνεπώς, αντί να θεωρήσουμε το παράγωγο του κατασκευαστή από μια κλάση ως αντικείμενο της κλάσης, μπορούμε να το θέσουμε και να το χρησιμοποιήσουμε ως υποκλάση με δικές τις ιδιότητες. Πως μπορούμε να το πετύχουμε αυτό; Θέτοντας την νέα υποκλάση αρχικά ως αντικείμενο της κλάσης-βάσης και να δημιουργήσουμε έναν κατάλληλο κατασκευαστή για την νέα υποκλάση.

Επιστρέφοντας στο πρόβλημα μας, ο ζωολογικός κήπος συναντάει την ανάγκη να χωρίσει όλα τα ζώα σε υποκατηγορίες προκειμένου να δημιουργήσει ζώνες για διάφορα είδη ζώων. Έτσι αποφασίζει να χωρίσει ο φύλακας τα ζώα του ζωολογικού κήπου σε τρεις ομάδες: τα πτηνά (Birds), τα θηλαστικά (Mammals) και τα ερπετά (Serpents).

Κάθε μία από τις παραπάνω κατηγορίες αποτελεί μια υποκλάση την οποία χρειάζεται να αρχικοποιήσουμε ως αντικείμενο της κλάσης Animal με κενές τιμές και να δημιουργήσουμε μια αντίστοιχη συνάρτηση κατασκευαστή για την νέα κλάση.

Παράδειγμα για Serpent:

```

--At your zoo there are three types of animals Serpents, Mamals and Birds
Serpent = Animal:create()

function Serpent:create(obj, name, numOfLegs, age, diet, cageNumber, species)
    obj = obj or Animal:create(obj, name, numOfLegs, age, diet, cageNumber, species)
    setmetatable(obj, self)
    self.__index = self
    return obj
end

```


Προφανώς, στην νέα κλάση `Serpent` μπορούμε να ορίσουμε δικές τις συναρτήσεις και μεταβλητές και συναρτήσεις που δεν υπάρχουν στην βασική κλάση, όπως την συνάρτηση `Serpent:venom` που φαίνεται παρακάτω.

```
function Serpent:venom()
    print(self.__name .. " shot venom ")
end
```

Πολυμορφισμός

Ο πολυμορφισμός εκφράζει την ιδιότητα στον αντικειμενοστραφή προγραμματισμό που μας επιτρέπει να δώσουμε σε μια μεταβλητή ή συνάρτηση μιας κλάσης νέα μορφή. Αυτό μπορεί να γίνει είτε στατικά, που ονομάζεται υπερφόρτωση μεθόδων και γίνεται πριν την μεταγλώττιση του κώδικα, είτε δυναμικά, που συνήθως εμφανίζεται με την μορφή επαναορισμού μιας συνάρτησης της κλάσης-βάσης στην υποκλάση της κατά την μεταγλώττιση.

Όσον αφορά την Lua, ο πολυμορφισμός σε αυτήν δεν διαθέτει τις ίδιες διαστάσεις. Πιο συγκεκριμένα η Lua από μόνη της **ΔΕΝ** διαθέτει στατικό πολυμορφισμό. Μιας και κάθε συνάρτηση βρίσκεται στον ίδιο πίνακα-αντικείμενο, η Lua δεν διαθέτει κάποιον άλλο τρόπο πέρα από το όνομα της συνάρτησης για να τα ξεχωρίσει, όπως και σε κάθε άλλη μεταβλητή που περιέχει.

Ωστόσο, ο δυναμικός πολυμορφισμός είναι δυνατός στην Lua. Ειδικότερα, σε κάθε υποκλάση μπορούμε να δημιουργήσουμε μια συνάρτηση με το ίδιο όνομα όπως η βάση-κλάση και αν τις δώσουμε νέες ιδιότητες είτε να χρησιμοποιήσουμε εξαρχής την συνάρτηση που κληρονόμησε. Το ερώτημα όμως είναι αφού ορίσουμε νέα συνάρτηση μπορούμε, όπως για παράδειγμα στην C++, να κάνουμε κάτι για να καλέσουμε την συνάρτηση του πατέρα; Η απάντηση είναι όχι καθώς στην Lua το `method overriding` συνεπάγεται και `method overwriting`, καθώς όπως αναφέραμε κάθε μέθοδος αποθηκεύεται σε μια μεταβλητή της οποίας αλλάξαμε την τιμή.

Στο παράδειγμα του ζωολογικού μας κήπου, μια βασική ιδιότητα κάθε ζώου είναι να μετακινείται. Συνεπώς, ορίζουμε μια συνάρτηση μέλος της κλάσης `Animal` που θα την ονομάσουμε `move`.

```
function Animal:move (dist)
    -- The double dot merges two strings together
    print(self.__name .. " moved " .. dist .. " meters")
end
```

Συνεπώς, αυτή η συνάρτηση θα περαστεί και στην υποκλάση `Serpent`. Ωστόσο, λεκτικά δεν είναι πολύ σωστό να χρησιμοποιήσουμε το ρήμα `moved` και θα προτιμούσαμε η συνάρτηση αυτή, δηλαδή το `move`, να χρησιμοποιεί το ρήμα

crawled. Αυτό μπορούμε να το πετύχουμε με τον δυναμικό πολυμορφισμό, ορίζοντας την συνάρτηση `Serpent:move`.

```
--Override functions
function Serpent:move(dist)
    print(self.__name .. " crawled " .. dist .. " meters")
end
```

Ενθυλάκωση

Αντίθετα, με τον πολυμορφισμό που είδαμε παραπάνω, η ενθυλάκωση δεν εμφανίζει τεράστια απόκλιση από τις πιο διαδεδομένες γλώσσες προγραμματισμού.

Ειδικότερα, προκειμένου να προστατεύσουμε τα δεδομένα του πίνακα-κλάσης χρειάζεται απλά να αρχικοποιήσουμε τα αντικείμενά μας με το σωστό scope, δηλαδή ως private.

Η εφαρμογή της ενθυλάκωσης είναι επίσης απλή. Ο λόγος είναι ότι οι `accessors` και `mutators` μιας και είναι στην ίδια δομή με τις ιδιωτικές μεταβλητές, δηλαδή στον ίδιο πίνακα, έχουν πρόσβαση και μπορούν να μεταβάλουν τις τιμές τους.

Τελειώνοντας το παράδειγμα του φύλακα παρατηρούμε ότι οι μεταβλητές μας στην κλάση `Animal` είναι ιδιωτικές και δεν έχουμε πρόσβαση σε αυτές εκτός του πίνακα. Για να το λύσουμε αυτό το πρόβλημα, για κάθε χαρακτηριστικό του πίνακα `Animal` δημιουργούμε μια συνάρτηση πρόσβασης (`accessor`) και μια συνάρτηση μετάλλαξης (`mutator`). Ως παράδειγμα θα δείξουμε το όνομα του ζώου.

```
--Encapsulation
function Animal:GetName()
    return self.__name
end

function Animal:SetName(newName)
    self.__name = newName
end
```

ΠΑΚΕΤΑ

Η Lua από μόνη της δεν διαθέτει υποστήριξη για την δημιουργία πακέτων, δηλαδή δομών οργάνωσης συναρτήσεων και τύπων της γλώσσας. Ωστόσο, όπως και με τις κλάσεις μας δίνεται η δυνατότητα να αναπαράγουμε τον τρόπο αυτό οργάνωσης των πακέτων με την χρήση πινάκων για να αποθηκεύουμε τα δεδομένα του πακέτου. Αυτό, δίνει και ένα πλεονέκτημα στην γλώσσα, αφού τα πακέτα αποτελούν δομές πρώτης τάξης, δηλαδή μπορούν να περαστούν και σε μεταβλητές/συναρτήσεις, κάτι που σε άλλες γλώσσες απαιτείται ειδικός μηχανισμός προκειμένου να εφαρμοστεί αυτό.

Στην Lua οι δομές αυτές, που συνήθως αναφέρονται με τον όρο modules(από το λατινικό modulo) ορίζονται σε ένα ξεχωριστό αρχείο της γλώσσας. Σε αυτό το αρχείο ορίζουμε έναν πίνακα με τοπική εμβέλεια και, όπως και στα αντικείμενα, ορίζουμε τις διάφορες συναρτήσεις και μεταβλητές που περιέχονται στο πακέτο. Στο τέλος του προγράμματος επιστρέφουμε τον τοπικό πίνακα. Συνήθως στον προγραμματισμό η δεσμευμένη λέξη return χρησιμοποιείται σχεδόν αποκλειστικά στο τέλος συναρτήσεων και όχι στο τέλος προγραμμάτων. Η Lua μας επιτρέπει να επιστρέφουμε τις τιμές μεταβλητών στο τέλος ενός προγράμματος. Ο τρόπος με τον οποίο μπορούμε να πάρουμε τις οντότητες αυτές σε ένα άλλο πρόγραμμα είναι με τη χρήση της συνάρτησης require με όρισμα το όνομα του αρχείου στο οποίο ορίσαμε το πακέτο και αποθηκεύοντας αυτό που επιστρέφει, δηλαδή τον πίνακα που μας δίνει πρόσβαση στα περιεχόμενα του πακέτου, σε μια τοπική μεταβλητή.

Ένα πακέτο που θα φαινόταν πολύ χρήσιμο στην Lua θα ήταν για την συνάρτηση print. Πιο συγκεκριμένα, η συνάρτηση αυτή στην Lua έχει δύο αρνητικά: Βάζει τον χαρακτήρα tab μεταξύ κάθε ορίσματος με κόμμα και χαρακτήρα αλλαγής γραμμής στο τέλος κάθε καλέσματος.

Η λύση αυτού έρχεται με την δημιουργία ενός πακέτου ως εξής:

- a. Δημιουργούμε ένα νέο αρχείο με το όνομα myPrint.lua
- b. Δημιουργούμε έναν νέο πίνακα με όνομα myprint
- c. Εισάγουμε στον πίνακα τις συναρτήσεις που θέλουμε να περιέχει το πακέτο
- d. Επιστρέφουμε τον πίνακα myprint στο τέλος του αρχείου
- e. Όταν θέλουμε να χρησιμοποιήσουμε τις συναρτήσεις του πακέτου καλούμε την συνάρτηση require με όρισμα το όνομα του αρχείου και αποθηκεύουμε αυτό που επιστρέφει σε μια τοπική μεταβλητή, δημιουργώντας ένα τοπικό αντίτυπο του πακέτου.

```
1 local myprint = {}
2
3 -- Prints a string without a line change character at the end
4 function myprint.prints(string)
5 |   io.write(string)
6 end
7
8 -- Prints a string with a line change character at the end
9 function myprint.printsln(string)
10 |   io.write(string, "\n")
11 end
12
13 return myprint
```

GARBAGE COLLECTOR

Η διαχείριση μνήμης στην Lua, παρότι η ίδια στηρίζεται στην γλώσσα προγραμματισμού C, δεν έχει καμία σχέση με την διαχείριση μνήμης της προαναφερόμενης. Σε αντίθεση με την C, η οποία αφήνει την διαχείριση μνήμης ολοκληρωτικά στον προγραμματιστή, η Lua έχει ένα ημιαυτόματο σύστημα για την

διαχείριση των δεσμεύσεων μνήμης των προγραμμάτων της που ονομάζεται Garbage Collector. Ο ρόλος της οντότητας αυτής είναι να αποδεσμεύει αυτόματα κομμάτια μνήμης που δεν χρειάζονται πλέον στο πρόγραμμά μας.

Στην Lua κάθε αντικείμενο μπορεί να συλλεχθεί από τον Garbage Collector από συναρτήσεις και πίνακες, μέχρι πακέτα και κορουτίνες. Υπάρχουν μόνο συγκεκριμένα μέρη του προγράμματος τα οποία δεν μπορεί να συλλέξει και αυτά είναι ο παγκόσμιος πίνακας, δηλαδή ο πίνακας που περιέχει όλες τις παγκόσμιες μεταβλητές, το κεντρικό thread λειτουργίας του προγράμματος, την οντότητα `package.loaded` καθώς και οποιοδήποτε μεταπίνακα είναι κοινό μεταξύ πινάκων, όπως μια κλάση που χρησιμοποιείται από τις υποκλάσεις της ή τα αντικείμενά της.

Ο Garbage Collector στην Lua έχει περάσει ιστορικά από δύο εκδοχές. Η πρώτη, που ήταν εν ισχύ μέχρι και την έκδοση 5.0 ακολούθησε την μέθοδο `mark & sweep`. Ειδικότερα, σε κάθε κύκλο εκτέλεσης του GC (Garbage Collector) ξεκινάγε περνώντας από τον Γράφο όλων των αντικειμένων (ξεκινώντας από τον πίνακα-ρίζα) και «χρωμάτιζε» κατάλληλα κάθε αντικείμενο που ήταν ενεργό στο πρόγραμμα (`mark`). Στην συνέχεια, έτρεχε την λίστα των αντικειμένων και διέγραφε τα αντικείμενα που δεν ήταν χρωματισμένα (`sweep`). Η ανάγκη για δημιουργία ενός νέου GC στην Lua προκλήθηκε από τις μεγάλες παύσεις μεταξύ των δύο βημάτων (`mark & sweep`), οι οποίες ήταν το χρονικό διάστημα στο οποίο ο GC διαχωρίζει τα αντικείμενα. Έτσι, με την έκδοση 5.1 της Lua εισάχθηκε ο αυξητικός GC (Incremental Garbage Collector), ο οποίος εκτελεί όλες τις διεργασίες του GC ταυτόχρονα με την εκτέλεση του προγράμματος.

Η νέα αυτή μορφή του GC, προκειμένου να αποφασίσει ποια αντικείμενα είναι προς αποδέσμευση στο πρόγραμμα χρησιμοποιεί τρεις χρωματισμούς: Οι μαύροι κόμβοι, οι οποίοι είναι οι κόμβοι που έχουμε επισκεφθεί και γνωρίζουμε ότι δεν πρέπει να αποδεσμευτούν, οι γκρι κόμβοι, που είναι οι κόμβοι που έχουμε επισκεφθεί αλλά δεν έχουμε διασπίσει, και οι λευκοί που είναι οι κόμβοι που δεν έχουμε επισκεφθεί.

Η διαδικασία αυτή εκτελείται αναλυτικά ως εξής: Όλοι οι κόμβοι ξεκινάν χρωματισμένοι λευκοί, με κάθε κόμβο να αντιπροσωπεύει ένα αντικείμενο, και το πρόγραμμα δημιουργεί έναν αρχικό Γράφο με τα ήδη προσβάσιμα στον GC δεδομένα, δηλαδή τα αντικείμενα που βρίσκονται στο σύνολο της ρίζας. Στην συνέχεια, ο GC ξεκινάει την διαδικασία της σημείωσης (`marking`) και ψάχνει να βρει διασταυρώσεις μεταξύ δύο κόμβων ενός λευκού και ενός γκρι. Άμα βρει, χρωματίζει τον λευκό κόμβο ως γκρι και σχεδιάζει και την αντίστοιχη ακμή (που πηγαίνει από τον εξαρχής γκρι κόμβο στον εξαρχής λευκό). Ένας γκρι κόμβος χρωματίζεται μαύρος όταν όλα τα παιδιά του γίνουν γκρι. Η διαδικασία της σημείωσης ολοκληρώνεται όταν δεν υπάρχουν γκρι κόμβοι, που σημαίνει ότι ο Γράφος είναι πλέον συνεκτικός.

Μετά το τέλος της διαδικασίας αυτής ακολουθεί το ατομικό βήμα, δηλαδή η διαδικασία στην οποία θα καθαρίσει αδύναμους πίνακες και θα ξεχωρίσει εντός της μνήμης τα συλλεγμένα και διαγραφμένα αντικείμενα. Το τελευταίο βήμα είναι η περισυλλογή των αποδεσμεύσεων (sweep), με την μόνη ιδιαιτερότητα να εμφανίζεται όταν ένα αντικείμενο προς διαγραφή να έχει μεταπίνακα, που σε αυτή την περίπτωση ο GC προσπαθεί να εκτελέσει την μεταμέθοδό του.

Τέλος, υπάρχουν ορισμένες ιδιαιτερότητες που συναντάμε όσο αφορά τον GC οι οποίες είναι:

- ❖ Οι αλλαγές σε έναν πίνακα τον κάνουν από μαύρο σε γκρι
- ❖ Οι αλλαγές σε έναν μεταπίνακα τον μετατρέπουν από λευκό σε γκρι
- ❖ Τα αντικείμενα που μετατρέπονται σε γκρι με τους παραπάνω τρόπους αποθηκεύονται σε μια ξεχωριστή από τον GC λίστα μέχρι και το ατομικό βήμα
- ❖ Όλες οι στοιβες είναι γκρι

Weak Tables

Όπως και στους περισσότερους GC, έτσι και ο GC της Lua επιλέγει, υπό κανονικές συνθήκες να μην διαγράψει αντικείμενα στα οποία έχουμε αναφορές, καθώς είναι αρκετά προφανές ότι αυτά τα αντικείμενα χρησιμοποιούνται ακόμα από τον χρήστη. Ωστόσο, υπάρχουν περιπτώσεις όπου θέλουμε, παρότι υπάρχει αναφορά σε ένα αντικείμενο, αυτό να συλλεχθεί από τον GC. Για να γίνει αυτό πιο κατανοητό θα δώσουμε ένα παράδειγμα: Έστω ότι θέλει ο προγραμματιστής να κρατάει κάθε ζωντανό αντικείμενο του προγράμματός μας σε μια συλλογή. Η συλλογή αυτή θα αναφέρεται σε κάθε αντικείμενο που είναι ζωντανό την ώρα της δημιουργίας του. Ωστόσο, αυτό δημιουργεί ένα θέμα της μορφής του ότι δεν είναι δυνατόν πλέον να διαγράψουμε κανένα αντικείμενο στο πρόγραμμά μας! Ακόμα και ένα αντικείμενο obj να μην έχει καμία πλέον χρήση στον προγραμματιστή και να μην υπάρχει και καμία άλλη αναφορά του αντικειμένου αυτού, συνεχίζει να υπάρχει στην συλλογή.

Λύση σε αυτό το θέμα δίνουν οι αδύναμοι πίνακες. Οι αδύναμοι πίνακες είναι μια μέθοδος που διαθέτει η Lua προκειμένου να πει ο προγραμματιστής στην γλώσσα ότι μια αναφορά δεν χρειάζεται να αποτρέψει την συλλογή του αντικειμένου αυτού από τον GC. Ως αδύναμη αναφορά ορίζουμε την αναφορά σε ένα αντικείμενο το οποίο δεν είναι στο score του GC. Άμα κάθε αναφορά σε ένα αντικείμενο obj είναι αδύναμη τότε το αντικείμενο συλλέγεται από τον GC και κάθε μία από τις αδύναμες αναφορές διαγράφεται. Πως όμως εφαρμόζουμε αυτές τις αδύναμες αναφορές στην Lua? Η απάντηση βρίσκεται στους αδύναμους πίνακες. Άμα ένα αντικείμενο βρίσκεται μόνο μέσα σε αδύναμους πίνακες, τότε το αντικείμενο αυτό εν καιρό θα συλλεχθεί από τον GC.

Πως όμως ορίζουμε στον κώδικα μας ότι ένας πίνακας έχει weak references; Η απάντηση βρίσκεται στους μεταπίνακες. Ειδικότερα, θα ορίσουμε έναν νέο πίνακα με μοναδικό όρισμα την μεταβλητή `__mode` και θα την κάνουμε μεταπίνακα του πίνακα που θέλουμε να έχει αδύναμες αναφορές. Η τιμή της μεταβλητής αυτής καθορίζει ποιες αναφορές είναι αδύναμες.

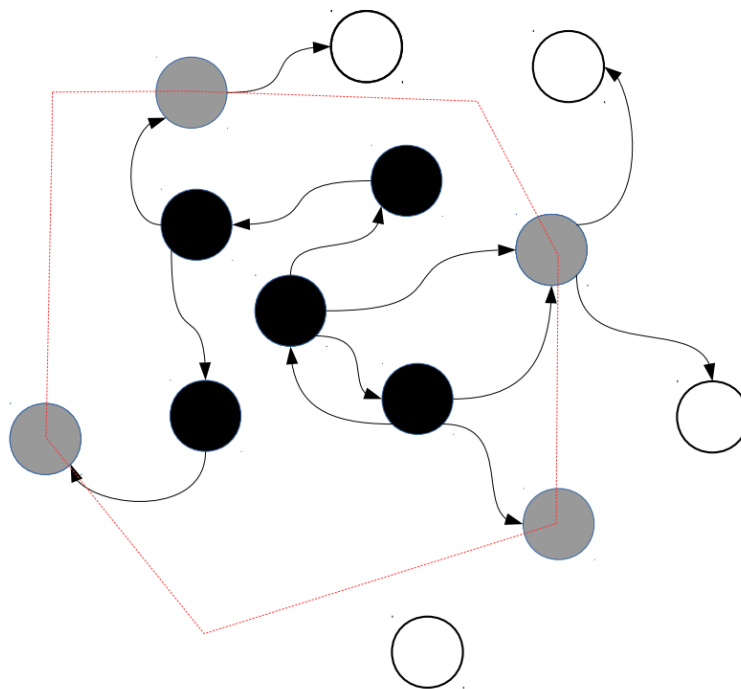
```
-- This table will have weak keys.
keys = {}
setmetatable(keys, { __mode = "k" })

-- This table will have weak values.
values = {}
setmetatable(values, { __mode = "v" })

-- This table will have weak keys and values.
both = {}
setmetatable(both, { __mode = "kv" })
```

Incremental Garbage Collector

Η πρώτη μορφή του GC που διαθέτει η Lua είναι ο αυξητικός. Στον αυξητικό GC διαθέτουμε δύο φάσεις διεργασιών, καθώς ο συγκεκριμένο GC συνεργάζεται με τον mutator, δηλαδή την κανονική εκτέλεση του προγράμματός μας. Συγκεκριμένα, ο GC περιμένει μέχρι να δεσμευτεί συγκεκριμένη ποσότητα μνήμης μέχρι να δράσει και να απορρίψει τα κατάλληλα κομμάτια μνήμης. Ο αυξητικός GC διαθέτει δύο μεταβλητές προς χρήση, την παύση (pause) η οποία ελέγχει κατά πόσο πρέπει να πολλαπλασιαστεί η δεσμευμένη μνήμη μέχρι να δράσει ο GC, και τον πολλαπλασιαστή, που ασχολείται με την μετάφραση των byte σε διεργασίες για τον GC.



Generational Garbage Collector

Η δεύτερη και λίγο πιο περίπλοκη εκδοχή του Garbage Collector ονομάζεται γενεαλογικός. Η λογική θεμέλιο στην χρησιμότητα του GC αυτού είναι ότι τα περισσότερα από τα αντικείμενα σε ένα πρόγραμμα δεν έχουν μεγάλη διάρκεια ζωής. Ως αποτέλεσμα, θα ήταν έξυπνο για τον GC να «συγκεντρωθεί» στα νεότερα αντικείμενα του προγράμματος, αφού αυτά έχουν μεγαλύτερες πιθανότητες να αχρηστευτούν από το πρόγραμμα. Η λογική αυτή στηρίζεται στο γεγονός ότι αντικείμενα με μεγάλη διάρκεια ζωής, οφείλουν την ύπαρξή τους στην συνεχή χρησιμότητά τους από το πρόγραμμα και συνεπώς θα επιζήσουν μέχρι το τέλος του προγράμματος. Ο γενεαλογικός garbage collector κατατάσσει όλα τα αντικείμενα με βάση το αν έχουν επιζήσει δύο κύκλους συλλογής σε νέα (young) και παλιά (old). Σε μια μικρή περισυλλογή, ο γενεαλογικός GC επιλέγει να τρέξει μόνο τα νέα αντικείμενα.

Η γραμματική της Lua δεν παρέμεινε κατά την ύπαρξη της ίδια. Μάλιστα στην πρώιμη έκδοση της Lua η γραμματική χρησιμοποιούσε την λέξη κλειδί `type` για να εκτυπώσει, που σήμερα έχει εγκαταλειφθεί, ενώ κάθε εντολή δημιουργίας αντικειμένων καλούταν με το σύμβολο `@`, ενώ στην σημερινή έκδοση το σύμβολο έχει εγκαταλειφθεί τελείως.

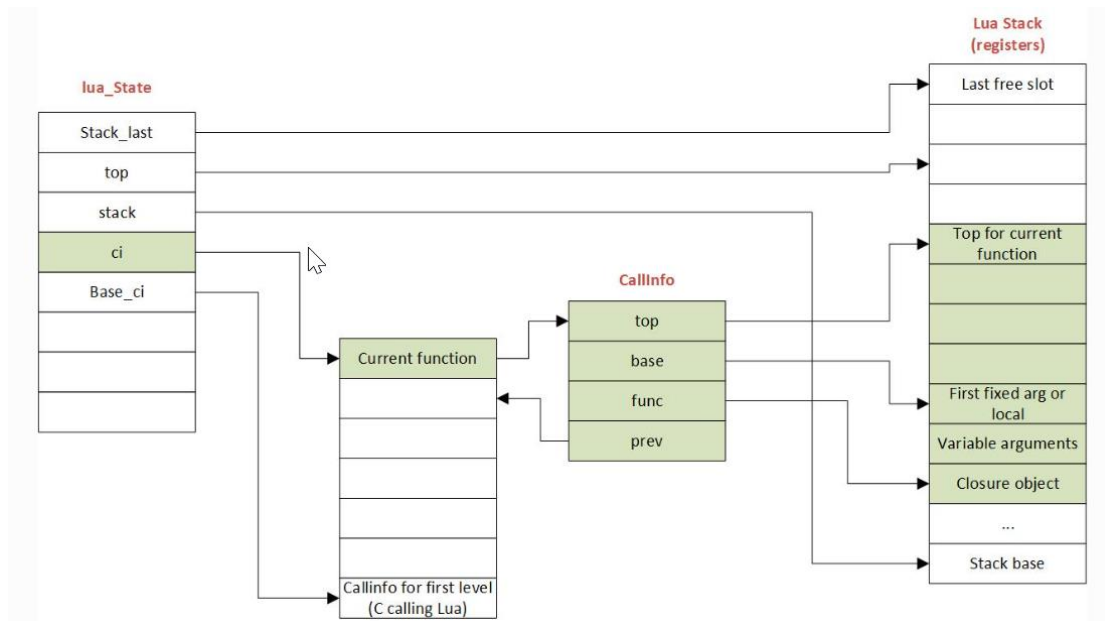
LUA PARSER

Ο parser της Lua διαθέτει μια περίπλοκη δομή και δεν ακολουθεί έναν κλασικό τύπο. Ειδικότερα, οι δομές και συναρτήσεις που περιέχονται στο αρχείο `lparser.c`, δηλαδή το αρχείο που αποτελεί τον ορισμό του γραμματικού αναλυτή, φαίνεται να περιγράφουν έναν parser τύπου LL(1), δηλαδή διαβάζει από τα αριστερά προς τα δεξιά με αριστερότερο σχηματισμό παραγώγου και ένα στοιχείο πρόβλεψης. Ωστόσο, δεν είναι αυστηρά LL(1) καθώς διαθέτει κάποιες επιπλέον τεχνικές πρόβλεψης για να λυθούν συγκεκριμένα θέματα με τον κώδικα. Αξιοσημείωτο είναι ότι έχουν δημιουργηθεί πολλοί ενδιαφέροντες parsers από προγραμματιστές για την γλώσσα αυτή με τον πιο διάσημο να είναι της μορφής LALR(1).

Παρατηρώντας λοιπόν το αρχείο του αναλυτή το πρώτο πράγμα που αξίζει να σημειώσουμε αφορά την χρήση της στοίβας από την Lua. Η Lua σαν γλώσσα διαθέτει δύο στοίβες: η βασική στοίβα `Callinfo` η οποία είναι υπεύθυνη για την καταγραφή των καλεσμάτων συναρτήσεων και μεταβλητών όσο τρέχει το πρόγραμμα και η στοίβα `L->Stack` που αποθηκεύει αποκλειστικά αντικείμενα της μορφής `TValue` και στην οποία δείχνουν οι τιμές της `Callinfo`.

Σημείωση: Το `TValue` ορίζεται στον κώδικα της C που απαρτίζει την ρίζα της Lua και αντιπροσωπεύει ένα αντικείμενο από το union `Value`, που αποτελεί τον εσωτερικό ορισμό **κάθε** αντικειμένου στην Lua, με type tag `"tt_"` (η χρησιμότητα του tag είναι εσωτερική του garbage collector).

Το σχήμα αυτό δείχνει πως συνδέονται οι στοίβες μεταξύ τους, όπου `Lua Stack` είναι το `L->Stack` και `lua_state` αναφέρει την τωρινή κατάσταση του προγράμματος που θα αναφέρουμε παρακάτω πως λειτουργεί:



Ας μιλήσουμε τώρα για την ουσιαστική δουλειά του ίδιου του αναλυτή για την Lua. Ο parser στην Lua συνεργάζεται με τον code generator προκειμένου να μεταφράσει το αρχείο κώδικα σε μορφή bytecode. Ο αναλυτής δεν διαθέτει build phase του κώδικα. Αντίθετα, δημιουργεί τον κώδικα όσο αναλύει (parsing) τον κώδικα της Lua. Ο κυριότερος λόγος για αυτή την διαφοροποίηση από άλλες γλώσσες φαίνεται να είναι η χρονική αποτελεσματικότητα. Το αρχείο του parser χρησιμοποιεί δομές δεδομένων στην στοίβα για να αποθηκεύει τα αντικείμενα, χωρίς να δεσμεύει καθόλου τον σωρό της C. Σε κάποιες περιπτώσεις η στοίβα χρησιμοποιείται κιόλας για την δημιουργία δομών δεδομένων με κύριο παράδειγμα να είναι η αναδρομή.

Όλες οι διεργασίες του parser στηρίζονται σε ένα και μόνο αντικείμενο που είναι το `expdesc` και έχει τον παρακάτω κώδικα:

```

1  typedef struct expdesc {
2      expkind k;
3      union {
4          struct { /* for indexed variables (VININDEXED) */
5              short idx; /* index (R/K) */
6              lu_byte t; /* table (register or upvalue) */
7              lu_byte vt; /* whether 't' is register (VLOCAL) or upvalue (VUPVAL) */
8          } ind;
9          int info; /* for generic use */
10         lua_Number nval; /* for VKFLT */
11         lua_Integer ival; /* for VKINT */
12     } u;
13     int t; /* patch list of 'exit when true' */
14     int f; /* patch list of 'exit when false' */
15     int ravi_type; /* RAVI change: type of the expression if known, else LUA_TNONE */
16 } expdesc;

```

Το αντικείμενο αυτό περνάει από πολλές καταστάσεις και συνεπώς είναι δύσκολο να ακολουθήσουμε την κατάστασή του. Ο τρόπος με τον οποίο λειτουργεί ο parser είναι ότι διατρέχει κανονικά τον κώδικα, δημιουργεί τα κατάλληλα κομμάτια bytecode και στην περίπτωση που κάνει λάθος γυρνάει πίσω και το διορθώνει. Ένα χαρακτηριστικό παράδειγμα αφορά τις συναρτήσεις: Έστω ότι ο προγραμματιστής έχει ορίσει μια συνάρτηση, που όπως είδαμε στην Lua δεν ορίζει πόσα ορίσματα θα επιστρέφει η συνάρτηση κατά τον ορισμό της. Ο parser θα υποθέσει ότι επιστρέφει ένα όρισμα και θα δημιουργήσει τον αντίστοιχο κώδικα. Ωστόσο, μετά όταν η συνάρτηση δημιουργηθεί όντως μπορεί ο προγραμματιστής να έχει επιλέξει να επιστρέφει δύο ορίσματα. Έτσι, θα επιστρέψει πίσω στον κώδικα ο parser και θα διορθώσει το λάθος του. Αντίστοιχο παράδειγμα είναι όταν μια τιμή δεν έχει ακόμα αποφασιστεί σε ποιόν καταχωρητή θα αποθηκευτεί στο register window και γίνεται parse το σύμβολο της ισότητας. Τότε, το αποθηκεύει σε έναν τυχαίο register και επιστρέφει να το διορθώσει όταν έρθει η σωστή πληροφορία.

Τέλος, ο parser δεν επιλέγει να κάνει μεγάλη προεπεξεργασία του προγράμματος με την μοναδική εξαίρεση να αποτελούν τα ορίσματα του τύπου require, που φορτώνουν ένα άλλο αρχείο Lua στο τρέχον πρόγραμμα που έχουμε.

ΚΑΤΑΣΤΑΣΕΙΣ ΤΗΣ LUA

Σε κάθε γλώσσα προγραμματισμού μελετάμε την μετάβαση από μία κατάσταση σε μία άλλη κατά την διάρκεια του parsing. Πως λειτουργεί αυτό στην Lua;

Όπως αναφέραμε, στον ορισμό του γραμματικού αναλυτή της Lua η βασική δομή που ελέγχει την πρόοδο της διαδικασίας είναι το αντικείμενο expdesc. Συνεπώς, θα μελετήσουμε τις καταστάσεις που παίρνει το συγκεκριμένο αντικείμενο οι οποίες είναι η παρακάτω:

- ❖ **VVOID:** Δείχνει ότι δεν υπάρχει κάποια τιμή στον parser, όπως για παράδειγμα όταν φτάσει στα ορίσματα μιας συνάρτησης η οποία δεν καλείται με κάποιο όρισμα. **State Transition:** Δεν διαθέτει.
- ❖ **VRELOCABLE:** Δείχνει ότι η τιμή που αναλύσαμε μόλις πρέπει να αποθηκευτεί σε έναν καταχωρητή. Η πράξη που οδήγησε σε αυτή την κατάσταση έχει αποθηκευτεί σε μια τοπική μεταβλητή u.info. **State Transition:** Το πρώτο πράγμα που γίνεται είναι ότι προστίθεται ο παρακάτω κανόνας στο πρόγραμμα

VRELOCABLE : **VVARARG** **VUPVAL** (**OP_GETUPVAL** **VINDEXED** (**OP_GETTABUP** or **OP_GETTABLE**

Επιπλέον, εφόσον μέσω της πρόβλεψης δούμε ότι η επόμενη κατάσταση είναι της μορφής VNONRELOC τότε μπορεί να προστεθούν και άλλοι κανόνες.
- ❖ **VNONRELOC:** Η κατάσταση αυτή υποδηλώνει ότι έχουμε αποθηκεύσει σε μια τοπική μεταβλητή u.info τον καταχωρητή εξόδου για μια πράξη και συνεπώς μπορεί να εκτελεστεί. Η πράξη βρίσκεται στον καταχωρητή που ορίσαμε

παραπάνω. **State Transition:** Δεν διαθέτει επόμενο κανόνα κατάστασης, απλά συνεχίζεται το parsing στον επόμενο χαρακτήρα.

- ❖ **VLOCAL:** Αναφέρουμε ότι μόλις διασχίσαμε μια τοπική μεταβλητή, της οποίας τον καταχωρητή αποθηκεύουμε στο u.info. **State Transition:** Πηγαίνει κατευθείαν στο VNONRELOC.
- ❖ **VCALL:** Είναι η κατάσταση που βρισκόμαστε μόλις καλείται μια συνάρτηση. Στο u.info αποθηκεύεται η εντολή `OP_CALL` και αλλάζει σε `OP_TAILCALL` εάν το πρόγραμμα επιστρέφει ορίσματα. Σε μετέπειτα χρόνο ενημερώνεται σωστά και ο αριθμός των ορισμάτων `c`. **State Transition:** Πηγαίνει στην κατάσταση VNONRELOC με την μη τελική μεταβλητή A να παίρνει την τιμή (u.info
- ❖ **VINDEXED:** Η κατάσταση αυτή αναφέρεται στο ότι μόλις αναφέρθηκε σε κλειδί ενός πίνακα. Η μεταβλητή `u.ind.t` περιέχει τον καταχωρητή του πίνακα, η μεταβλητή `u.ind.idx` τον καταχωρητή του κλειδιού και η μεταβλητή `u.ind.vt` μία από τις τιμές VLOCAL ή VUPVAL. **State Transition:** Πηγαίνει στο VRELOCABLE και το u.info παίρνει ως τιμή του το offset του κώδικα που περιέχει των κωδικό της διαδικασίας opcode.

ΠΑΡΑΔΕΙΓΜΑ ΓΡΑΜΜΑΤΙΚΗΣ ΑΝΑΛΥΣΗΣ

Θα ερευνήσουμε ένα απλό κομμάτι κώδικα, προκειμένου να δούμε πως το αναλύει βήμα προς βήμα ο γραμματικός αναλυτής τη Lua. Δεν θα αναφερθούμε σε κάθε σύμβολο και γράμμα του κώδικα, αλλά μόνο σε αυτά που φέρουν ενδιαφέροντος ή παρουσιάζουν κάποια ιδιαιτερότητα στην Lua.

Έστω λοιπόν το κομμάτι κώδικα `local i,j; j = i*j+i` που προφανώς αποτελείται από τον ορισμό δύο τοπικών μεταβλητών `i` και `j` και την αποθήκευση μιας τιμής στην μεταβλητή `j` που βγαίνει από πράξεις πολλαπλασιασμού και πρόσθεσης μεταξύ των μεταβλητών `i` και `j`.

Κατά την εκκίνηση του parsing για τον κώδικα, ο μεταγλωττιστής δεσμεύει τα παρακάτω κομμάτια μνήμης για τις τοπικές μεταβλητές, σταθερές και upvalues που πιθανώς να έχουμε ως εξής:

constants (0) **for** 0000007428FED950:

locals (2) **for** 0000007428FED950:

0	i	2	5
1	j	2	5

upvalues (1) **for** 0000007428FED950:

0	_ENV	1	0
---	------	---	---

Μόλις φτάσει στο σύμβολο `i`, κατά τον ορισμό των τοπικών μεταβλητών, δημιουργεί την μεταβλητή και όπως είδαμε την αποθηκεύει στον καταχωρητή 0. Ως `p` ορίζουμε την διεύθυνση του δείκτη που περιέχεται στην βασική δομή `expdesc` και το

αντικείμενο αυτό εξελίσσεται ως:

```
{p=0000007428E1F170, k=VLOCAL, register=0}
```

Αντίστοιχα, εξελίσσεται το αντικείμενο όταν διαβάσει την τοπική μεταβλητή j, που βρίσκεται στον καταχωρητή 1 ως εξής:

```
{p=0000007428E1F078, k=VLOCAL, register=1}
```

Αφού περάσει ο parser στην πράξη και συγκεκριμένα στο σύμβολο της πράξης MUL, δηλαδή τον αστερίσκο, το πρώτο πράγμα που γίνεται είναι ότι ο καταχωρητής i περνάει σε κατάσταση VNONRELOC ως εξής:

```
{p=0000007428E1F170, k=VNONRELOC, register=0} MUL {p=0000007428E1F078, k=VLOCAL, register=1}
```

Μόλις φτάσει ο parser να δημιουργήσει τον κώδικα για τον ίδιο τον operator, το παραπάνω κομμάτι θα αντικατασταθεί από μια έκφραση τύπου VRELOCABLE, με την εντολή να φαίνεται παρακάτω:

```
{p=0000007428E1F170, k=VRELOCABLE, pc=1, instruction=(MUL A=0 B=0 C=1)}
```

Μετά βλέπει ότι ξαναχρειάζεται για την ADD την τοπική μεταβλητή i και κάνει αναφορά ως εξής:

```
{p=0000007428E1F078, k=VLOCAL, register=0}
```

Στην πράξη ADD μεταξύ του i που βρίσκεται στον καταχωρητή 0 και το προσωρινό καταχωρητή για το αποτέλεσμα του πολλαπλασιασμού (register = 2) πηγαίνουμε τον καταχωρητή 2 στην κατάσταση VNONRELOC και παίρνουμε:

```
{p=0000007428E1F170, k=VNONRELOC, register=2} ADD {p=0000007428E1F078, k=VLOCAL, register=0}
```

Αντίστοιχα με το MUL καταλήγουμε στην κατάσταση VRELOCABLE με instruction την πρόσθεση:

```
{p=0000007428E1F170, k=VRELOCABLE, pc=2, instruction=(ADD A=0 B=2 C=0)}
```

Τέλος, γίνεται η αποθήκευση στον καταχωρητή j και επιστρέφει έτσι στην κατάσταση VNONRELOC. Ο τελικός κώδικας μοιάζει ως εξής:

```
main <(string):0,0> (4 instructions at 0000007428FED950)
0+ params, 3 slots, 1 upvalue, 2 locals, 0 constants, 0 functions

 1      [1]    LOADNIL      0 1
 2      [1]    MUL          2 0 1
 3      [1]    ADD          1 2 0
 4      [1]    RETURN      0 1
```

ΣΥΝΤΑΚΤΙΚΗ ΑΝΑΛΥΣΗ

Σε αυτό το σημείο έχοντας αναφερθεί και στα specification της γλώσσας, αλλά και στην αναλυτική λειτουργία του parser, η συντακτική ανάλυση μπορεί να επικεντρωθεί μόνο στην ανάλυση του αρχείου που περιέχει το συντακτικό της Lua.

Στο εισαγωγικό κομμάτι αναλύσαμε πως το συντακτικό της Lua αναφέρεται ολοκληρωτικά σε μορφή extended Backus-Naur. Ναι αλλά το ερώτημα είναι πως αναλύουμε την μορφή αυτή για την κατανόηση των terminal και non-terminal τιμών;

Η μορφή αυτή αναπαριστά τους κανόνες αντικατάστασης που ακολουθεί ο parser της Lua κατά την αντικατάσταση συγκεκριμένων non-terminal, δηλαδή μη τερματικών, τιμών και διαβάζονται από αριστερά προς τα δεξιά.

Κάποιες ιδιαιτερότητες που παρουσιάζει η μορφή είναι:

- ❖ Στα αριστερά του κάθε κανόνα βρίσκεται η μη τερματική τιμή προς αντικατάσταση και συνήθως στις γραμματικές EBNF συμβολίζεται με <symbol>
- ❖ Ακολουθείται από τον χαρακτήρα αντικατάστασης ::= . Παρότι στις περισσότερες γλώσσες τύπου EBNF ο χαρακτήρας αυτός έχει «εξελιχθεί» είτε στο ίσων (=) είτε στο :=, η Lua χρησιμοποιεί τον ίδιο χαρακτήρα με τον τύπο BNF.
- ❖ Στα δεξιά του χαρακτήρα βρίσκεται το κομμάτι που θα αντικαταστήσει την μη τερματική τιμή <symbol> και συμβολίζεται συνήθως με __expression__ . Μπορεί να αποτελεί οποιονδήποτε συνδυασμό από τερματικές και μη-τερματικές τιμές, χωρίς να γίνεται αυτοαναφορά. Έτσι καταλήγουμε στην μορφή των κανόνων ως

```
<symbol> ::= __expression__
```
- ❖ Σε μονά εισαγωγικά περιέχονται τα σύμβολα που διαβάζονται από τον αναλυτή της Lua, ενώ το σύμβολο | συμβολίζει την διάζευξη μεταξύ των κανόνων στην Lua. Για παράδειγμα οι διπλοί τελεστές γράφονται ως

```
binop ::= '+' | '-' | '*' | '/' | '/' | '^' | '%' | '&' | '~' | '|' | '>>' | '<<' | '..' | '<' | '<=' | '>' | '>=' | '==' | '~=' | and | or
```
- ❖ Οι διάφορες δεσμευμένες λέξεις κλειδιά της Lua παρουσιάζονται με έντονη επισήμανση όπως πχ οι **return** και **break**. Παράδειγμα ο ορισμός των expressions ως

```
exp ::= nil | false | true | Numeral | LiteralString | '...' | functiondef | prefixexp | tableconstructor | exp binop exp | unop exp
```
- ❖ Σε άγκιστρα {} περιέχονται μη τερματικές τιμές που μπορεί να υπάρχουν είτε μηδέν είτε και πολλαπλές φορές στον κανόνα όπως στο

```
attname::list ::= Name attrib {',' Name attrib}
```

 και χρησιμοποιούνται επίσης οι αγκύλες [] για να δείξουν προαιρετικά κομμάτια του κανόνα όπως στο

```
block ::= {stat} [retstat]
```
- ❖ Τέλος, υπάρχουν τέσσερις μη τερματικές τιμές που δεν ορίζονται στο σχήμα EBNF, καθώς αποτελούν τα τρία βασικά μη τερματικά και είναι οι Name, Numeral και LiteralString.

Μεταγλωττιστής

Η Lua είναι μία interpreted γλώσσα προγραμματισμού με κύρια χαρακτηριστικά της η μεταφερισιμότητα με τις λιγότερες δυνατές θυσίες σε κόστος μνήμης και

σημαντικότερα χρόνου. Όπως έχει είδη αναφερθεί πρακτικά όλα τα κομμάτια της Lua, από τον μεταγλωττιστή μέχρι την εικονική μηχανή, είναι γραμμένα σε C.

Πιο συγκεκριμένα είναι γραμμένη σε ANSI C (όπου ANSI σημαίνει American National Standards Institute), που αποτελεί την παγκοσμίως καθιερωμένη υλοποίηση της C, καθώς και τους κανόνες που οι προγραμματιστές ευθύνονται να ακολουθούν. Το γεγονός αυτό είναι κρίσιμο για την μεταφερσιμότητα της γλώσσας, αφού ακολουθώντας το ANSI standard οποιοδήποτε σύστημα που μπορεί να μεταγλωττίσει C89/90 μπορεί να τρέξει και να ενσωματώσει τη Lua.

Η ευθύνη του μεταγλωττιστή αρχίζει στην λεκτική ανάλυση και τελειώνει με την παράδοση του ενδιάμεσου bytecode κώδικα στον διερμηνέα (και τυχούσες βελτιστοποιήσεις από τον Just In Time Compiler αν υπάρχει). Μετά από τα γνωστά στάδια, Λεκτική ανάλυση (Tokenization) και Συντακτική ανάλυση (Parsing), ακολουθεί σημασιολογική ανάλυση (Semantic Analysis), όπου ο μεταγλωττιστής ελέγχει την εγκυρότητα των τύπων, των scopes και άλλων περιορισμών της γλώσσας. Ύστερα, το πρόγραμμα μετατρέπεται σε ενδιάμεση γλώσσα.

Για το κομμάτι της ενδιάμεσης αναπαράστασης ο μεταγλωττιστής μετατρέπει το δέντρο σε Bytecode. Οι εντολές Bytecode εκτελούνται μία προς μία από την εικονική μηχανή της Lua.

Κάθε εντολή Bytecode της Lua είναι 32 bits. Αυτό το μέγεθος εντολής Bytecode είναι σημαντικά μεγαλύτερο από τις περισσότερες εικονικές μηχανές. Για αυτό ευθύνεται η μετατροπή της εικονικής μηχανής της Lua από stack machine σε register machine με την κυκλοφορία της έκδοσης 5.0.

Οι μηχανές στοίβας, όπως δηλώνει και το όνομα, χρησιμοποιούν μία στοίβα στην πάνω στην οποία εκτελούνται όλες οι εντολές. Στοιχεία προστίθενται και αφαιρούνται από την στοίβα όταν δηλώνονται και απελευθερώνονται. Οι πράξεις γίνονται πάνω στην στοίβα που σημαίνει ότι για τις περισσότερες εντολές μπορούν να εννοηθούν οι τελεστές.

Οι μηχανές καταχωρητών επίσης χρησιμοποιούν την στοίβα καθώς και, όπως φανερώνει το όνομα καταχωρητές. Για τον λόγο αυτό πρέπει να γίνουν περισσότερες διευκρινίσεις εντός της εντολής ως προς το ποιοι καταχωρητές ή ποιες σταθερές χρησιμοποιούνται. Παρόλα αυτά το μέγεθος του παραγόμενου κώδικα είναι ασήμαντο, αφού παρόλο που αυξάνεται το μέγεθος των εντολών, ο αριθμός τους μειώνεται λόγω της μεγαλύτερης ακρίβειας των πράξεων.

Τα πρώτα 6 bits είναι πάντα opcode, δηλαδή διευκρινίζουν ποια εντολή αναπαριστά το κομμάτι αυτό και τι αναπαριστούν τα υπόλοιπα 26 bits.

Τα υπόλοιπα bits μπορούν να έχουν πολλές χρήσεις:

8 bits για το A που συνήθως είναι ο καταχωρητής προορισμού που αποθηκεύονται τα αποτελέσματα των εντολών.

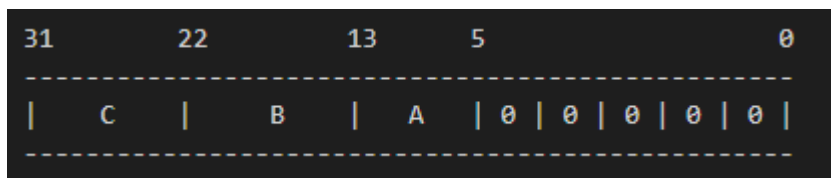
9 bits για τα B και C που μπορούν να είναι είτε καταχωρητές είτε σταθερές, τα οποία διαφοροποιούνται από το επιπλέον bit.

26 bits για το A,B και C μαζί, το οποίο αποκαλείται Ax.

18 bits για το τα B και C μαζί, το οποίο αποκαλείται Bx.

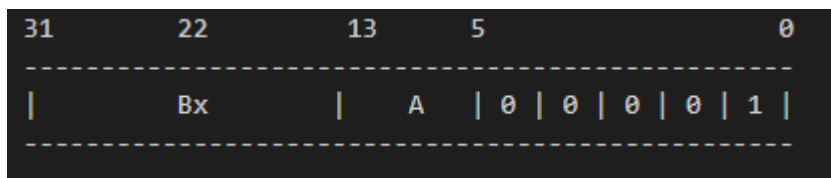
18 bits για το τα B και C μαζί αλλά προσημασμένα, το οποίο αποκαλείται sBx.

Ανάλογα με την εντολή χρησιμοποιούνται διαφορετικοί τελεστές. Ακολουθούν παραδείγματα Bytecode εντολών:

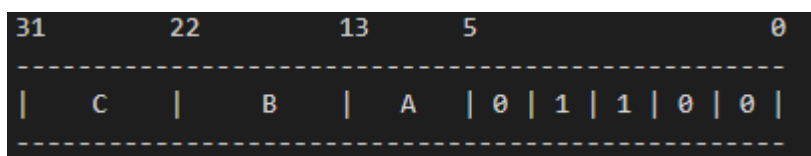


OP_MOVE: Η εντολή αυτή μετακινεί τα περιεχόμενα του B στο A. Τα bits του C είναι περιττά και άρα δεν χρησιμοποιούνται.

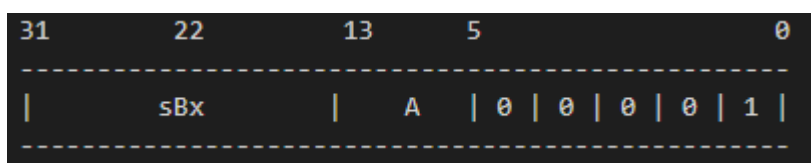
Η εντολή έχει opcode 0.



OP_LOADK: Η εντολή κάνει load μία σταθερά από τη διεύθυνση Bx στον καταχωρητή A. Η εντολή έχει opcode 1.



OP_ADD: Η εντολή προσθέτει τις B και C αποθηκεύοντας το αποτέλεσμα στην A. Η εντολή έχει opcode 12



OP_JMP: Η εντολή αυξάνει τον μετρητή προγράμματος κατά sBx (ή μειώνει καθώς ο χρησιμοποιείται το sBx που είναι προσημασμένο)

Μετά την μετατροπή του AST (abstract syntax tree) σε Bytecode ο το πρόγραμμα μεταφράζεται σε κώδικα μηχανής και εκτελείται. Εδώ μπορεί να γίνει βελτιστοποίηση του κώδικα, όπως στην έκδοση LuaJIT (Lua Just In Time). Η συγκεκριμένη έκδοση της Lua χρησιμοποιεί διάφορες τεχνικές με τις οποίες μειώνει το κόστος χρόνου και μνήμης του προγράμματος, κατά την διάρκεια τις εκτέλεσης.

Οι τεχνικές αυτές συμπεριλαμβάνουν Trace-Based βελτιστοποίηση, όπου μέρη του κώδικα που εκτελούνται συχνά μετατρέπονται σε κώδικα μηχανής εξειδικευμένο για την εκτέλεση αυτού του κομματιού, ανάλυση ελέγχου ροής, όπου γίνονται βελτιστοποιήσεις σε βρόγχους, καθώς και ειδικευση τύπων (Type Specialization), όπου γίνονται βελτιστοποιήσεις στον κώδικα μηχανής για συγκεκριμένους τύπους, μειώνοντας τον χρόνο συγκρίσεων και πράξεων μεταξύ τους.

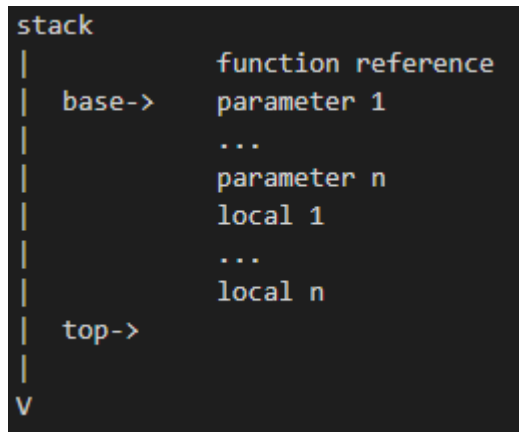
Όπως αναφέρθηκε προηγουμένως, η εικονική μηχανή της Lua είναι μηχανή καταχωρητών, που σημαίνει ότι χρησιμοποιεί στοίβα καθώς και εικονικούς καταχωρητές. Για μία συνάρτηση η Lua αποθηκεύει όλες τις παραμέτρους σε συνεχείς καταχωρητές. Ύστερα, όταν κληθεί η συνάρτηση οι καταχωρητές γίνονται μέρος του ιστορικού ενεργοποιήσεων της συνάρτησης αυτής. Το αποτέλεσμα αυτής της πράξης είναι η γρήγορη πρόσβαση στους καταχωρητές σαν τοπικές μεταβλητές. Όταν η συνάρτηση τελειώσει οι καταχωρητές εισάγονται στο ιστορικό ενεργοποιήσεων της καλούσας συνάρτησης.

Η Lua χρησιμοποιεί δύο στοίβες για κάθε συνάρτηση που καλείται. Η πρώτη έχει ένα στοιχείο για κάθε ενεργή συνάρτηση. Το στοιχείο αυτό περιλαμβάνει:

- Την συνάρτηση που καλείται αυτή τη στιγμή.
- Την διεύθυνση επιστροφής όταν η συνάρτηση κάνει μία κλίση.
- Ένα βασικό ευρετήριο, το οποίο δείχνει στο ιστορικό ενεργοποιήσεων της συνάρτησης.

Η άλλη στοίβα είναι απλά ένας μεγάλος πίνακας που κρατάει τα ιστορικά ενεργοποιήσεων. Τοπικές μεταβλητές αποθηκεύονται στο ιστορικό ενεργοποιήσεων της συνάρτησης.

Για μία συνάρτηση η στοίβα έχει την εξής μορφή:



Ο top δείχνει ακριβώς μετά από τις μεταβλητές της συνάρτησης.

Ο base δείχνει ακριβώς μετά το όνομα της συνάρτησης.

Για περισσότερες εμφωλευμένες συναρτήσεις όπως στον ακόλουθο κώδικα:

```

function f2(arg1, arg2, ..., argN)
  local local1, local2, ...
  ...
  return ret1, ret2, ..., ret0
end

function f1(arg1, arg2, ..., argM)
  local local1, local2, ...
  ...
  local ret1, ret2, ..., retP = f2(arg1, arg2, ..., argN)
  ...
end

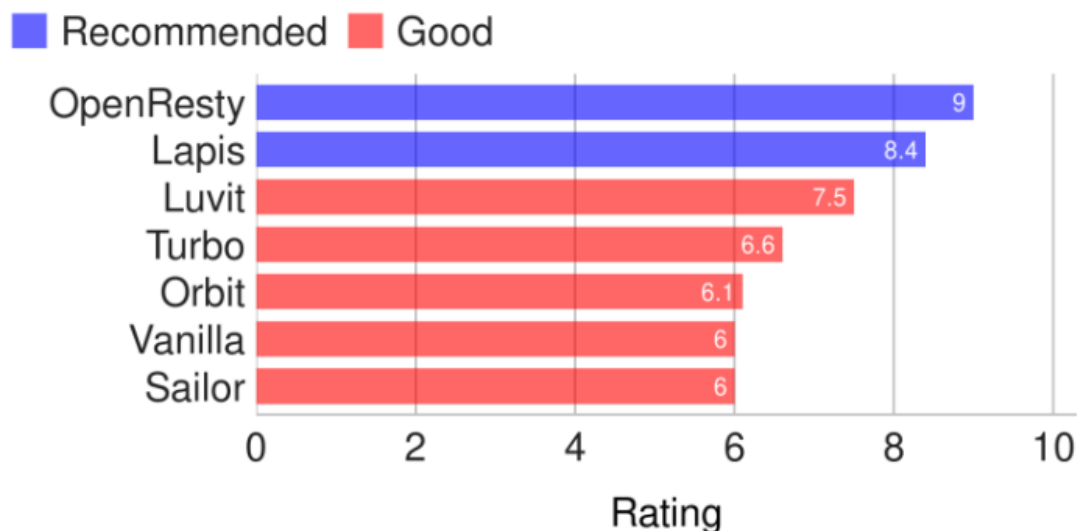
```

η εικονική μηχανή αποθηκεύει τις πληροφορίες της συνάρτησης σε διαφορετικά τμήματα της στοίβας. επόμενες συναρτήσεις βρίσκονται ακριβώς μετά από τις παραμέτρους, ενώ οι τιμές που επιστρέφονται πρέπει να αντιγραφούν, καθώς ο μεταγλωττιστής δεν γνωρίζει πόσες τιμές μπορεί να επιστρέψει μία άλλη συνάρτηση.

Έτσι το από πάνω τμήμα κώδικα μετατρέπεται στις εξής «εικόνες»:

- ❖ OpenResty: Αποτελεί έναν ολοκληρωμένο server για διαδικτυακές εφαρμογές με την ομαδοποίηση διάφορων Nginx modules, πάνω στον πυρήνα του Nginx. Το πλαίσιο OpenResty πετυχαίνει την ενίσχυση του web server Nginx με την χρήση της Lua ως την γλώσσα διαχείρισης των διάφορων modules, γραμμένα σε C και Lua, με σκοπό να δημιουργεί πιο αποδοτικές εφαρμογές.
- ❖ Lapis: Αποτελεί ένα πλαίσιο δημιουργίας διαδικτυακών εφαρμογών μέσω της γλώσσας Lua(ή της Moonscript) χρησιμοποιώντας ως server την προαναφερόμενη OpenResty. Ο συνδυασμός των δύο εφαρμογών, μαζί με την σχετικά μεγάλη ταχύτητα εκτέλεσης που προσφέρει η Lua οδηγεί στην δημιουργία εξαιρετικά γρήγορων εφαρμογών
- ❖ Luvit: Η Luvit αποτελεί μια συλλογή από πακέτα/modules με τα οποία μπορούμε να εφαρμόσουμε ένα API(Διεπαφή Προγραμματισμού Εφαρμογών) που παρομοιάζει το, ευρέως χρησιμοποιημένο, `node.js`. Η Luvit περιέχει το virtual machine της Lua, το openssl (βιβλιοθήκη κρυπτογράφησης δεδομένων), το miniz (βιβλιοθήκη συμπίεσης δεδομένων) καθώς και πολλές άλλες βιβλιοθήκες.

Το παρακάτω διάγραμμα δείχνει 7 από τα πιο χρήσιμα πλαίσια που χρησιμοποιούν την Lua, τα οποία είναι κιόλας δωρεάν προς χρήση και open-source.



APPLICATIONS

Κόντρα στην άποψη του μεγαλύτερου ποσοστού των προγραμματιστών, πολλές εταιρίες χρησιμοποιούν την γλώσσα προγραμματισμού Lua για την διαχείριση των εσωτερικών τους συστημάτων, καθώς και για την δημιουργία εφαρμογών.

Κάποια από τα πιο σημαντικά παραδείγματα αποτελούν τα παρακάτω:

- ❖ Cisco Systems, ευρύτερα αναφερόμενη ως Cisco, είναι μια εταιρία που εξειδικεύεται στις τεχνολογίες ψηφιακής επικοινωνίας και έχει δημιουργήσει επίσης την γνωστή πλατφόρμα Webex. Η εταιρία αυτή χρησιμοποιεί την Lua για την εφαρμογή κανόνων δυναμικής πρόσβασης, που επιτρέπει στην εταιρία των έλεγχο των application authorizations.
- ❖ Η πλατφόρμα επεξεργασίας εικόνων Adobe Photoshop Lightroom, που αποτελεί μία πιο ελαφριά έκδοση του Photoshop, χρησιμοποιεί την Lua για την δημιουργία του interface του.
- ❖ Άλλες δύο πλατφόρμες επεξεργασίας εικόνων, το Blackmagic Fusion και το DaVinci Resolve, χρησιμοποιούν κώδικα Lua για την αυτοματοποίηση λειτουργιών και την επέκταση ορισμένων δυνατοτήτων μέσω του API της Lua.
- ❖ Η πλατφόρμα δημιουργίας κινούμενων σχεδίων Moho, που έχει χρησιμοποιηθεί για την δημιουργία εξαιρετικών ταινιών κανονικής διάρκειας όπως την Romeo & Juliet: Sealed with a Kiss του Phil Nibbelink και το My Father's Dragon από τα στούντιο Cartoon Saloon, τα οποία στούντιο να αναφερθεί πως χρησιμοποιούν σχεδόν αποκλειστικά την Moho για τις παραγωγές τους. Η Moho χρησιμοποιεί την Lua ως την βασική γλώσσα προγραμματισμού για την δημιουργία αυτοματοποιημένων διαδικασιών από τον χρήστη.
- ❖ Η γνωστή εφαρμογή διαχείρισης και επεξεργασίας βάσεων δεδομένων MySQL Workbench χρησιμοποιεί την Lua για τις επεκτάσεις της.
- ❖ Μία από τις πιο γνωστές εφαρμογές αναπαραγωγής και επεξεργασίας βίντεο VLC media player, χρησιμοποιεί την Lua για υποστήριξη scripting.

ΠΑΙΧΝΙΔΙΑ ΚΑΙ GAME ENGINES

Η Lua σαν γλώσσα χρησιμοποιείται ευρέως για την δημιουργία βιντεοπαιχνιδιών (video games), η οποία είναι και προφανώς η πιο συχνή της χρήση. Πολλοί μεγάλοι τίτλοι έχουν φτιαχτεί αποκλειστικά με την χρήση της γλώσσας προγραμματισμού Lua και όλων των εργαλείων και βιβλιοθηκών που διαθέτει. Υπάρχουν πάρα πολλοί λόγοι που η Lua έχει γνωρίσει τέτοιων διαστάσεων εκτίμηση από τον τομέα της ανάπτυξης βιντεοπαιχνιδιών.

Πιο συγκεκριμένα:

- ❖ Είναι απλή στην εκμάθηση και κατανόηση. Η Lua καταφέρει να είναι πιο απλή από γλώσσες όπως η Python, με την δεύτερη να θεωρείται ήδη ευρέως ως εύκολη προς εκμάθηση γλώσσα, λόγω του εύκολου στην κατανόηση συντακτικό σε συνδυασμό με το γεγονός ότι κάθε δομή δεδομένων στηρίζεται στην ίδια οντότητα (tables)
- ❖ Λόγω του interpreter που διαθέτει η Lua καταφέρει να είναι πολύ γρήγορη και ελαφριά γλώσσα. Συγκεκριμένα, θεωρείται από τις πιο γρήγορες interpreted γλώσσες, σημειώνοντας μεγάλη βελτίωση από τον «ανταγωνιστή» της στην Python, ενώ χρησιμοποιεί και μικρό κομμάτι μνήμης

- ❖ Ο προγραμματιστής έχει την δυνατότητα να εφαρμόσει έναν interpreter της Lua εντός της εφαρμογής/παιχνιδιού. Αυτό επιτυγχάνει την γρηγορότερη εκτέλεση του προγράμματος σε κάθε μηχανήμα
- ❖ Είναι δωρεάν στην χρήση και open-source με τον κώδικά της να είναι διαθέσιμο στο site της
- ❖ Η Lua διαθέτει μια τεράστια κοινότητα προγραμματιστών που προσφέρει μεγάλη υποστήριξη στην εκμάθηση και εξέλιξη της γλώσσας, ενώ ακόμα και σήμερα δημιουργούνται από προγραμματιστές συνεχώς νέα εργαλεία για την γλώσσα

Ερευνώντας τους τίτλους και τα στούντιο που χρησιμοποιούν την Lua ως βασική γλώσσα προγραμματισμού θα βρούμε διάφορες κατηγορίες παιχνιδιών καθώς και πάρα πολύ διάσημους και αναγνωρίσιμους τίτλους.

Ειδικότερα:

- Κλασικά αγαπημένα όπως το **Angry Birds**, **Far Cry** και **World Of Warcraft**
- Παιχνίδια βασισμένα σε επιτυχημένα σειρές, όπως **Star Wars** και **Pokémon**, και επιτυχημένα φυσικά παιχνίδια, όπως **Lego** και **Magic: The Gathering**
- Σειρές παιχνιδιών που έγραψαν την ιστορία του 21^{ου} αιώνα όπως **Mafia**, **S.T.A.L.K.E.R**, **Saints Row**, **Warhammer 40,000**, **Chocolatier** με την μεγαλύτερη επιτυχία να γνωρίζει η σειρά **Civilization**, επιλέγοντας την Lua για τους δύο πιο πρόσφατους από τους 5 πολύ επιτυχημένους τίτλους της.
- Παιχνίδια που βρήκαν την επιτυχία τους την τελευταία δεκαετία όπως το **Roblox** και το **Dota 2**.

Πέρα από τους πολλούς τίτλους που παρουσιάσαμε η Lua αποτελεί και την βασική γλώσσα προγραμματισμού σε αρκετές μηχανές δημιουργίας παιχνιδιών όπως η **Leadwerks Game Engine**, η **CRYENGINE** και η, προσφάτως δημοσιευμένη, **Core**.