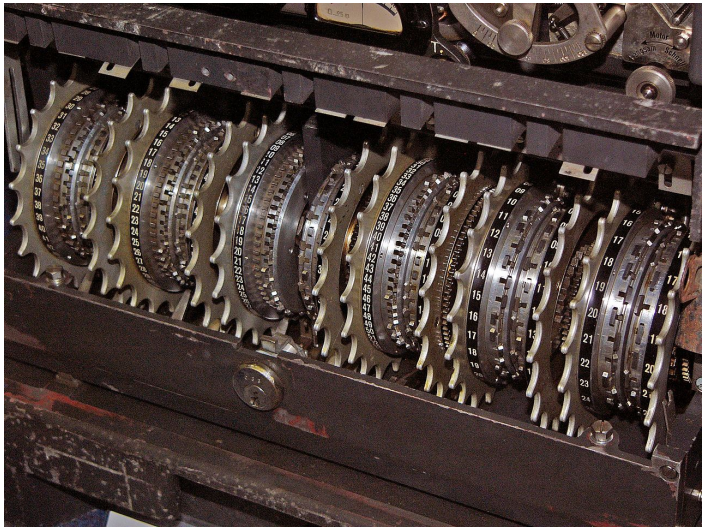
Kryptologie in Lua

Eine Einführung in Programmierung und die Kunst der Kryptologie
für den Unterricht

H. Behrens (CC-BY)



1 Was ist eine Programmiersprache?

Mittels einer Programmiersprache erklären wir dem Computer einen Ablauf von Vorgängen, mit dem wir ein Problem lösen wollen. Das heißt alle Programmiersprachen formulieren Probleme, so dass der Computer diese für uns lösen kann. Diese formulierten Lösungen heißen Algorithmen und das gesamte Paket aus Algorithmen nennen wir ein Programm.

1.1 Warum benutzen wir Lua?

Lua in der Version 5.2 ist eine einfache Sprache, die grundsätzlich nur aus 22 Grundbefehlen besteht, sogenannten „Keywords“. Diese Keywords und ein paar Sonderzeichen with +, -, /, = zum Beispiel sind für ein einfach zu erlernendes Grundvokabular, das wir schnell erlernen können und uns statt auf eine komplexe Sprache besser auf komplexe Lösungen konzentrieren können.

Moderne Sprachen zeigen die Tendenz, es dem Anwender einfach zu machen, sie zu benutzen und sie im Kopf zu behalten, ältere Sprachen verkomplizieren die Dinge häufig und setzen viel voraus. Ein einfaches Programm in Lua kann jeder innerhalb von wenigen Stunden selbst schreiben und während „Anfängersprachen“ der Anwendbarkeit dieser Programme oftmals enge Grenzen setzen.

Lua ist keine Anfängersprache, sondern zielt auf professionellen Einsatz ab, mit einer sehr großen Reichweite von Anwendungen, von herkömmlicher Anwendungs-Programmierung über Spieleprogrammierung, künstliche Intelligenz, Systemprogrammierung oder Scripting (Batch-Programmierung), kann als Erweiterung von Funktionen eines bestehenden Programmes (embedded) benutzt werden oder als Meta-Programmiersprache Daten zusammen fügen aus dem Netz, sie kann verschiedene Datenbanken abfragen, Buchdruck setzen oder eine beliebige Aufgabe von abertausenden übernehmen, mit denen sich andere Sprachen häufig schwer tun.

Lua macht einige dieser schwierigen Aufgaben nicht leichter als sie sind, aber sie stellt sich ihnen nicht in den Weg, indem sie den Programmierer zu sehr dazu zwingt, eine bestimmte Art der Programmierung, einen Stil zu wahren (auch bekannt als Paradigma oder Programmier-Ideologie). Lua ist für sich genommen ideologie-frei, ist aber fähig alle diese Paradigmen zu erfüllen, die sind: imperativ (prozedural), funktional, objektorientiert, reflektiv, scripting. Zudem können wir Systemfunktionen mittels FFI benutzen.

Weitere Paradigmen lassen sich ohne weiteres in Lua implementieren, da die Sprache leicht verständlich und modular erweiterbar gestaltet wurde.

Da das Hauptaugenmerk von Lua jedoch nicht der eingeschränkte Einsatz in der Lehre oder Ausbildung ist, wurde sehr auf effizienz und Geschwindigkeit bei der Ausführung wert gelegt. Vergleichbare Sprachen der selben Kategorie sind häufig genug mehr als zehnmal langsamer, manche Sprachen hinken mehr als Faktor hundert hinterher.

Der Einsatz von Lua im Scripting von Spielen, die alle sehr komplexe Aufgaben in Echtzeit zu erledigen haben, hatten diese Effizienz zur Folge. Zur Zeit gibt es keine Hoch-Sprache, die die benutzerfreundlichen Merkmale von Lua bietet und auch nur annähernd so schnell ist, mehr dazu im Abschnitt „Performance“ auf den Seiten von LuaJIT.

Lua 5.2 Referenz	https://www.lua.org/manual/5.2/
LuaJIT	http://luajit.org/
Wikipedia	https://de.wikipedia.org/wiki/Lua
Zerobrane IDE	https://studio.zerobrane.com/

1.2 Die 22 Keywords von Lua

Lua in der Version 5.2 besitzt nur 22 Keywords, also reservierte Verben, die wir nicht für Variablen oder andere Zwecke benutzen dürfen. Aus diesen 22 Grundworten bestehen alle in Lua geschriebenen Programme. Es ist wichtig, diese Verben zu lernen und auswendig zu kennen, wenn man flüssig programmieren können will und nicht ständig auf dieser Seite nachschlagen.

`and`	`break`	`do`	`else`
`elseif`	`end`	`false`	`for`
`function`	`goto`	`if`	`in`
`local`	`nil`	`not`	`or`
`repeat`	`return`	`then`	`true`
`until`	`while`		

Wir kommen in den nächsten Kapiteln zur Bedeutung dieser einzelnen Keywords zurück. Zusätzlich zu den Keywords gibt es eine Reihe von Zeichen, die ebenfalls fest definiert sind, das sind

`+ - * / % ^ #`
`== ~= <= >= < > =`
`( ) { } [ ]`
`; : , . .. …`

Mehr gehört nicht dazu. Fangen wir also gleich an!

1.3 Was ist Programmieren?

Programmieren ist nicht schwer; Probleme lösen kann allerdings knifflig werden. Aber! Jeder kann programmieren. Programmieren heißt für den Programmierer, den Computer zu einer Erweiterung seines eigenen Gehirns machen zu können. Nicht nur das Nutzen fertiger Lösungen, sondern neues zu erschaffen, Wege zu gehen, die noch keiner vorher gegangen ist. Zu lernen kritisch mit den eigenen Lösungen umgehen, immer nach einem Fehler suchen, nach einer Verbesserung; es gibt keinen perfekten Code. Die Optimisten schätzen, dass alle zehn Zeilen Programm immer noch einen versteckten Fehler enthalten, die Pessimisten trauen keiner einzigen Zeile. Unsere Software besteht aus Aber-Millionen von Zeilen von Code, sei also stolz auf jede Zeile, die funktioniert. Aber trau ihr nicht.

Wir finden neue Wege. Diese neuen Wege finden sich schnell. Die Informatik ist eine junge Wissenschaft und was in diesen sechzig oder siebzig Jahren die Wissenschaftler, Ingenieure und Pioniere erschaffen haben, füllt Bücher, ganze Bibliotheken, aber wir stehen erst ganz am Anfang.

Auch wenn es leicht aussieht, es wird geschätzt, dass die Mehrzahl der Informatiker mit Studienabschluss weltweit (das heißt über 50%) nicht programmieren können und es vermutlich niemals mehr erlernen. Die Grundlage für das Programmieren muss in jungen Jahren erfolgen, da es nicht nur um das Programmieren selbst geht, sondern um das Analysieren von Problemen, das folgerichtige und systematische Denken, kurz die wissenschaftliche Methode zu erlernen. Immanuel Kant, der Vordenker der Aufklärung schreibt:

> *„Aufklärung ist der Ausgang des Menschen aus seiner selbst verschuldeten Unmündigkeit. Unmündigkeit ist das Unvermögen, sich seines Verstandes ohne*

> *Anleitung eines anderen zu bedienen. Selbst verschuldet ist diese Unmündigkeit, wenn die Ursache derselben nicht am Mangel des Verstandes, sondern der Entschließung und des Muthes liegt, sich seiner ohne Leitung eines anderen zu bedienen. Sapere aude [wage es verständig zu sein]! Habe Muth, dich deines eigenen Verstandes zu bedienen! ist also der Wahlspruch der Aufklärung.“*
>
> *– Beantwortung der Frage: Was ist Aufklärung?: Berlinische Monatsschrift, 1784,2, S. 481–494*
>
> *(Zitat aus dem Deutschen Wikipedia)*

Programmieren können heißt in der heutigen Zeit Lesen und Schreiben können, die wichtigsten Maschinen unserer Zeit bedienen zu können, aktiv an der Entwicklung unserer Gesellschaft mitarbeiten zu lernen, statt nur dessen Spielball zu sein. Das Programm, das ist der Golem, die mit Geist beseelte Maschine, der buchstäbliche Zauberspruch mit dem wir Maschinen zum Leben erwecken, mit dem wir wahrhaft zu Magiern werden unser Wort wird Gestalt. Das Wort ist wahrhaftig mächtiger als das Schwert, unsere Bücher leben!

Denn unser Wort allein kann in Form des Programmes Berge versetzen, kann die Gesellschaft verändern, die Zukunft formen und ungelöste Rätsel der Menschheit knacken. Der Computer wird, egal was wir später als Beruf wählen mindestens 80% unseres Tätigkeitsfeldes sein. Sei es in Verwaltung, Kontrolle, Protokoll, Kommunikation oder der Bedienung von speziellen Maschinen oder Protokollen.

Ohne die Fähigkeit der Programmierung sind wir im 21sten Jahrhundert begrenzt in unseren Möglichkeiten, wie es ein Analphabet im 19ten Jahrhundert gewesen wäre. Doch können wir das, steht uns die Welt offen.

Und sei es, dass wir nichts anderes machen, als im nächsten Computerspiel einen entscheidenden Vorteil vor den anderen Spielern zu erlangen, sei es nur, dass wir als Anwalt schneller die wichtigen Teile in Gesetzen finden lassen können, es reicht nicht mit zu schwimmen. Es reicht nicht, die Programme der anderen bedienen zu können.

Programmierung heißt, dass du deine eigenen Vorstellungen für den Computer verständlich formulieren kannst und ihn zum Teil deiner Zukunft werden lassen kannst. Es ist eine Chance, es ist das Wissen wie man einen Bogen baut in einer Zeit, wenn alle anderen höchstens mit Steinen schmeißen können.

Wir Programmierer sind die Kinder von Apollon, der Namensgeber des Mondfahrt-Programms der NASA Ende der 60er bis Mitte der 70er Jahre. Apollo, der Gott des Lichts und der Aufklärung, der Findung von Wahrheit, aber auch der Gott der Dichtkunst!

Denn Programmierung ist Dichtkunst.

Und dieser Apollon hat eine Zwillingsschwester Artemis, die ihm gleich gestellt ist, die Göttin der Jagd, des Bogenschießens, der angewandten Hochtechnologie (der Altgriechischen Zeit). Wir sind ihre Kinder und wir stapfen in den Fußabdrücken von Neil Armstrong, von Buzz Aldrin auf dem Mond, portugiesisch „Lua“. Unsere Programme ticken noch 70 Jahre nach dem Start der Sonde in der Voyager-Sonde am Rande unseres Sonnensystems.

550 Menschen waren bislang im Weltall. 3300 haben es ins Linux-Kernel geschafft, welches das größte zusammenhängende Projekt der Menschheitsgeschichte ist (ausgenommen die Wissenschaft selbst), mit so viel Arbeit darin, so dass weit über 100 Apolloprogramme (nicht nur Missionen) bezahlt werden könnten.

Ein 19jähriges Mädchen schrieb die Assembler-Programme für die Mondlandefähre: Margaret Hamilton.

Unsere Helden heißen Alan Turing, Lady Ada Lovelace, Grace Hopper und die anderen ENIAC-Girls, Dennis Ritchie und Ken Thompson, Richard Stallman oder Linus Torvalds und viele, viele mehr. Wir haben das Wissen befreit, die Kommunikation, die Distanz der Menschen zueinander verringert, so dass wir einfach jemanden in Pakistan etwa fragen können, wie er oder sie die Sache vor Ort beurteilt. Wir sind mitten in einer Revolution.

Künstliche Intelligenz ist am Erwachen, Schach, Go, Poker, überall Programme, Code.

2 Wie funktioniert ein Programm?

Der Computer liest ein Programm von oben nach unten, links nach rechts wie in einem Buchtext. Und jedes Kommando führt er exakt so aus, wie es da steht, was nicht unbedingt heißt er tut das, was du meintest. Computersprachen sind sehr genau, sie lassen keinen Raum für Spekulation oder Unsicherheit, denn der Computer braucht genaue Anordnungen, damit er funktioniert.

Er tut nur, was du ihm sagst. Um eine Entscheidung treffen zu können, ob er etwas tut oder nicht, kennt er nur wahr oder falsch, oder in Lua **`true`** und **`false`**. Das sind schon mal die ersten beiden der 22 keywords. Ein weiteres ist das `if`.

```
if a<5 then
      print(“a ist kleiner als 5.“)
end
```

Das „if“ bestimmt, ob ein Programmteil ausgeführt wird oder nicht. Ist die Bedingung hinter dem „if“ wahr, wird der Teil danach ausgeführt, bis das „end“ kommt. Wenn nicht, springt das Programm zur nächsten Anweisung hinter dem „end“.

2.1 Variablen

„a“ ist hier eine Variable, eine „Veränderliche“, also Variablen beginnen mit einem Buchstaben oder „_“ und können beliebig heißen, dürfen aber nicht heißen wie keywords. Ein Lua-Programm unterscheidet zwischen Groß- und Kleinschrift, also „name“ ist unterschiedlich von „Name“. Variablen sind die einzige Möglichkeit für ein Programm, sich etwas zu merken, etwas zu speichern und zu verarbeiten. Lua kennt nur wenige Arten, was die Sache für den Programmierer vereinfacht, nämlich:

2.2 boolean

Speichert nur einen Wahrheitswert, wie true oder false. Nichts anderes. Ja oder nein, wahr oder falsch. In eine boolean Variable kann man die Ergebnisse von Vergleichen speichern oder von logischen Operationen. Wir tippen in die Lokale Konsole von Zerobrane ein:

```
> stunde=12
> tag=stunde > 6 and stunde < 18
> print(tag)
  true
> print(type(tag))
  boolean
> =tag
  true
> stunde=21
> tag=stunde > 6 and stunde < 18
> =tag
  false
>
```

Wie wir sehen, können wir in der Konsole auch mit „=“ statt print() eine Ausgabe erzeugen. Das funktioniert aber nur in der Konsole, nicht in einem Lua-Programm.

Zu den boolean Variablen gehört gleich auch die Verknüpfung „**and**“ und „**or**“ welches das logische und und oder sind. Wenn die Stunde größer als 6 ist UND kleiner als 18 und nur dann ist „tag“ wahr. Für UND müssen beide Bedingungen wahr sein, damit das Ergebnis wahr ist, bei ODER ist es genau umgekehrt. Beide müssen „false“ sein, damit das Ergebnis unwahr ist, ist nur eins davon wahr, bleibt das Ergebnis wahr.

Wir haben dann noch das „**not**“ und damit ist unsere Logik komplett. Die Vergleiche sind dann einfach „==“ gleich, „~=“ ungleich, „<=“ kleiner gleich, „>=“ größer gleich, „<“ kleiner, „>“ größer. Das einfache „=“ wird ausschließlich dazu benutzt, um einer Variablen einen neuen Wert zu zuweisen. Wie bei „stunde=21“ oder „stunde=12“.

Wir haben damit schon die ersten drei Keywords der 22 erledigt:

```
or, and, not
```

sowie die ganze Reihe von Vergleichen, sowie die Zuweisung. Außerdem haben wir mit „type()“ feststellen können, welchen Typ unsere Variable eigentlich hat. Vier erledigt.

2.3 number

Der zweite Variablentyp ist „number“ also eine Zahl. Wir haben schon in der Variable „stunde“ im vorigen Beispiel gesehen, wie das aussehen kann. Zahlen können Ganzzahlen sein oder Zahlen mit Nachkommastellen, sogenannte Fließkommazahlen, sie können positiv oder negativ sein oder in wissenschaftlicher Schreibweise geschrieben werden.

Andere Sprachen unterscheiden zwischen all diesen Zahlen, Lua vereinfacht uns die Sache, wir können natürlich Lua beibringen penibel damit zu werden, aber das brauchen wir in der Regel nicht, also warum uns damit belasten?

```
> a=5
> b=2.5
> =a+b
  7.5
> c=1e2
> =c
  100
> c=-c
> =c
  -100
>
```

Wir können die üblichen Operationen anwenden, wie +, -, *, /. Zu bemerken ist die Operation %, die den Rest einer Division ergibt, also 5/2 sind 2 Rest 1. Außerdem gibt es noch das ^ was die Potenz ist, also 10^2 ist 10*10.

Die Operationen werden in mathematischer Reihenfolge behandelt, also Punktrechnung vor Strichrechnung, Potenz vor dem Rest, Vorzeichen vor allem. Wollen wir die Reihenfolge erzwingen, Klammern wir den Ausdruck. Ein paar Beispiele.

```
> =5/2
  2.5
> =5%2
  1
> =3+4*2
  11
> =3*4^2
  48
> =(3*4)^2
  144
> a=4e100
> =type(a)
  "number"
>
```

Ist eine Variable unbekannt oder soll sie gelöscht werden, setzen wir sie auf „`nil`“, also ohne Wert. Fünf erledigt.

2.4 Weitere Daten-Typen für später

Wir kümmern uns später um sie im Detail für den Momant: eine Variable kann `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, `table` sein. Das war‘s für die Variable.

3 Programm-Strukturen

Wir haben fünf der Keywords durch, erledigen wir gleich mal einen ganzen Satz, die sich um das „if“ herum bilden.

```
if Bedingung then
      weitere Befehle
end
```

Wie im oberen Beispiel schon gezeigt, drei weitere weg, sind wir bei acht.

```
if Bedingung then
      -- trifft zu
elseif nächste Bedingung then
      --die erste war nichts
      --die zweite trifft zu
else
      --ansonsten das hier
end
```

Wir können mit „--“ den Rest der Zeile als Kommentar markieren, ansonsten haben wir mit `elseif` und `else` zwei weitere Keywords erledigt. Und damit sind wir schon halb durch mit den Lua-Keywords.

3.1 Schleifen

Um Programmteile wiederholt ablaufen lassen zu können, braucht Lua eine Reihe von Schleifen-Befehlen. Die erste ist die while-Schleife, das zweite die repeat-until-Schleife, die dritte die for-Schleife.

3.1.1 do-while Schleife

Der Unterschied dieser beiden ist einfach. „**while**“, also solange die Bedingung erfüllt ist, führe alles aus, was zwischen „**do**“ und „**end**“ steht, dann wiederhole, bis die Bedingung nicht mehr erfüllt ist.

Ist die Bedingung von vorn herein `false`, wird diese Schleife eventuell *keinmal* ausgeführt, also wie bei einem if einfach übersprungen.

```
> a=3
> while a>0 do print(a); a=a-1 end
  3
  2
  1
> while a>0 do print(a); a=a-1 end
>
```

3.1.2 repeat-until Schleife

Anders bei einem repeat-until. Diese Schleife wiederholt alles zwischen **`repeat`** und **`until`** bis die Bedingung bei until `true` ist. Die Schleife wird erst am Ende geprüft und wird damit *mindestens einmal* ausgeführt.

```
> a=3
> repeat print(a); a=a-1 until a<1
  3
  2
  1
> repeat print(a); a=a-1 until a<1
  0
> repeat print(a); a=a-1 until a<1
  -1
```

Wir können Befehle der Übersichtlichkeit halber mit „;“ von einander trennen, aber ein einfaches Leerzeichen hätte es auch getan. Es gibt Situationen, wo das „;“ im übrigen wichtig wird, dazu später mehr. Lua kann durchaus in einer einzigen Zeile programmiert werden, man macht es aber in der Regel so, dass jeder Befehle in einer Zeile steht.

3.1.3 for-Schleife

Die letzte Art der Schleife ist die „**`for`**“ Schleife. Die ist etwas komplexer, aber in Lua in der Regel die schnellste Art, irgend etwas in einer Schleife zu durchlaufen, was gezählt wird oder abgearbeitet.

Die „for“-Schleife ist genau genommen nicht nötig, sie ist das selbe wie eine while-Schleife und funktioniert auch genauso. Aber sie hat eine Schleifenvariable mit eingebaut.

```
for i=start,end,step do Befehle end
```

Die Schleifenvariable heißt hier i (wie Index), aber man kann natürlich nehmen, was für die Aufgabe passt. Die Werte start, end und step sind hier start, der Wert mit dem die

Schleife beginnt, end, der Wert ab dem die Schleife abbricht und step, um wieviel sie bei jedem Durchlauf hoch gezählt wird. Wird der Schritt nicht angegeben, wird eine eins angenommen. Natürlich kann Step auch eine negative Zahl sein, die Schleife zählt dann rückwärts.

```
> for i=1,3 do print(i) end
  1
  2
  3
> for i=1,3,0.5 do print(i) end
  1
  1.5
  2
  2.5
  3
> for i=1,0,-0.25 do print(i) end
  1
  0.75
  0.5
  0.25
  0
>
```

Die for-Schleife kann aber noch mehr. Das Keyword „**in**“ ermöglicht dieser Schleife ein paar nette Sachen, die natürlich auch in den anderen Schleifen zur Verfügung stehen, aber die for-Schleife ist das, was man „syntactic sugar“ nennt, also ein Feature, das die Dinge nett und übersichtlich macht. In älteren Lua-Versionen war die for-Schleife daher nicht enthalten.

Aber sagen wir, wir haben eine `string` variable, also eine Zeichenkette.

```
> s="hallo"
> for c in s:gmatch('.') do print(c) end
  h
  a
  l
  l
  o
>
```

Mit Hilfe des Keywords „**in**“ kann man mit einer for-Schleife also durch Mengen blättern. Beispielsweise durch die Charaktere eines strings mit gmatch(), zu dem wir später genauer kommen oder durch den Inhalt einer `table`.

```
> t={"Apfel", "Birne", "Banane"}
> for i,frucht in pairs(t) do print(i,frucht) end
  1 Apfel
  2 Birne
  3 Banane
>
```

Damit haben wir auch die Keywords „in“ und die Iterations-Funktionen „pairs“ sowie im letzteren Beispiel wäre auch ipairs() gegangen. Die Feinheiten von pairs und ipairs vertiefen wir, wenn wir mehr über Tabellen lernen.

Man kann eine Schleife übrigens abbrechen, wenn man ein „`if` Bed `then` **`break`** `end`“ benutzt. Break beendet alle Arten von Schleifen sofort und macht am Ende dieser Schleife weiter. Break verläßt nicht verschachtelte Schleifen, sondern nur die aktuelle.

3.1.4 goto

Bleibt ein letzter Schleifenbefehl: das **`goto`**. Es gab Ende der 60er Jahre viel Aufregung um das goto und die Ideologie der Zeit verteufelte es. Tatsache ist, dass das Goto, der Sprung an eine andere Programmstelle das einzige ist, was unsere Prozessoren tatsächlich können, um den Programm-Ablauf zu ändern. Aber auch in einer Hochsprache wie Lua findet das goto seinen Platz.

Die Anwendungen sind nicht häufig und oft nicht sinnvoll, aber wenn man einen Fall für ein Goto hat, dann verbessert es die Lesbarkeit des Codes. Zum Beispiel, wenn man wegen einer Abbruch-Bedingung aus mehreren geschachtelten Schleifen springen muss oder wenn es um die Reaktion auf einen Fehlerfall geht. Das Goto kann auch eine sogenannte Finite-State Maschine darstellen.

Wir kümmern uns wegen dieser zwar wichtigen, aber eher detail-orientierten Fälle erst einmal nicht ums goto außer einem kurzen Beispiel. Mehr dazu in der Diskussion zu dem Thema im Lua-Wiki.

```
1   #!/usr/bin/env luajit
2   for z=1,10 do
3      for y=1,10 do
4         for x=1,10 do
5            if x^2 + y^2 == z^2 then
6               print('found a Pythagorean triple:', x, y, z)
7               goto done
8            end
9         end
10     end
11  end
12  ::done::
13
```

Das Programm bricht die dreifache geschachtelte Schleife bei einem gültigen Ergebnis ab und gibt aus:

```
found a Pythagorean triple:   4     3     5
```

und fährt dann weiter im Programm fort. Eine Konstruktion, die ansonsten an Übersicht verlieren würde ohne „goto“. Es ist jedoch der Fall, dass dieser kleine Befehl meistens falsch eingesetzt wird und daher sollte man ihn vermeiden, es sei denn man weiß, was man tut. Eine mit Gotos aufgebaute Schleife kann alles, was die anderen Schleifen können, aber durch ihr freies, unstrukturiertes und anarchisches Wesen ist sie manchmal in der Ausführung langsamer als einer der strukturierten Schleifenbefehle. Zumindest innerhalb einer Hochsprache wie Lua.

Ein falscher Einsatz kann zu Kritik durch andere Programmierer führen und ermüdende Dispute mit Reinlichkeitsfanatikern von der Inquisition und vor allem Leuten, die dieses nützliche kleine Werkzeug grundsätzlich für falsch und schlecht halten und es am Stab auf dem Marktplatz brennen sehen wollen.

Das Goto wurde trotz der ideologisch bedingten Widerstände mit Lua 5.2 neu in die Sprache eingeführt. Einer der Gründe, die für die Sprache sprechen; sich frei von Ideologie halten und pragmatisch zu bleiben, lösungsorientiert. Computer mögen rein rational sein, Programmierer sind es oft leider nicht.

Der Disput um das „Goto considered harmful“ von Djikstra aus dem Jahre 1968 ist ansonsten höchst unterhaltsam und war fast vierzig Jahre unanfechtbare Doktrin. Aber das Goto löst eine Menge Probleme und es ist gut, es zu haben. Man sollte es trotzdem nur in besonderen Fällen nutzen.

Wir bewegen uns innerhalb der letzten zehn Jahre weg von den Doktrinen und orientieren uns wieder mehr an Problemen und Lösungen. Lua ist der frische Wind dieser Bewegung.

3.2 Funktionen

Eine Funktion ist ein Programmteil oder ein Algorithmus, den man aufrufen kann und der nach Beendigung zurückkehrt zum aufrufenden Programmteil. Hätte man also irgendwo eine Funktion „Quadrat“ definiert, kann man mit x=Quadrat(y) in diese Funktion springen, das Ergebnis berechnen lassen und zurück kehren.

Wenn eine Funktion einen Rückgabewert hat, bezeichnet man sie üblicherweise als Funktion, hat sie keinen Rückgabewert, ist es eine Prozedur oder auch „Unterprogramm“.

Das ist praktisch, wenn man einen Programmteil wiederverwenden möchte, beispielsweise aufrufen von verschiedenen Punkten des Programmes. Die Funktion, die wir kennen ist zum Beispiel die Bibiotheksfunktion „print()“.

Funktionen können am Beginn des Programmes stehen oder mittels „**`require`**“ aus anderen Dateien nachgeladen werden. Diese Dateien bezeichnet man dann als Module oder wenn man mehrere dieser Module zusammen fasst auch als Bibliothek.

Eine Bibliothek ist also eine Sammlung von Modulen, was eine Anzahl von Unterprogrammen oder Funktionen ist, die sich in der Regel um eine bestimmte Aufgabe drehen.

In Lua mitgliefert ist eine Bibliothek für `strings`, `tables`, `bit`, `io`, `os` (System) und `debug` sowie `ffi` und `jit`. Später dazu mehr.

Eine Funktion beendet entweder beim `end` oder beim `return`, wobei mit return auch Werte zurück gegeben werden können.

```
1  function quadrat(x)
2      return x^2
3  end
4
5  print(quadrat(3))
6  print(quadrat(4))
```

Dieses Programm druckt die Zahlen 9 und 16, mittels „return“ ein Wert zurück gegeben wird. Es können mit return auch mehrere Werte zuück gegeben werden.

Die hier gezeigte Art eine Funktion zu definieren ist wiederum „Syntaktischer Zucker“, denn was Lua eigentlich macht ist:

```
quadrat = function(x) return x^2 end
```

Das heißt Lua weist der Variablen „quadrat“ den Wert einer Funktion zu. Wenn wir also mit type(quadrat) den Typ der Variablen abfragen, bekommen wir „function“ als Antwort. Wir hatten im Kapitel über Variablen dies bereits kurz erwähnt. Funktionen sind also genau genommen auch nichts weiter als Werte in Variablen für Lua und aufgerufen wird diese über die Klammern„()“.

3.3 Scope von Variablen: global vs local

Bisher kennen wir nur globale Variablen, also Variablen, auf die von überall zugegriffen werden kann. Es ist jedoch nützlich (und nebenbei schneller), wenn wir Variablen haben können in Funktionen, in Programmteilen, die uns allein gehören und auf die kein anderer zugreifen kann. Diese Art von Variablen wird eine lokale Variable genannt und wir deklarieren sie mit dem Keyword „`local`“.

Wir können nicht nur in Funktionen diese lokalen Variablen deklarieren, sondern auch in Programm-Blöcken, die wir mit „do“ beginnen und „end“ beenden. Alles, was wir auf einer lokalen Variable machen, bleibt lokal und wir brauchen uns nicht darum zu kümmern, ob wir damit etwa außerhalb unserer Funktion irgend eine globale Variable überschreiben.

```
1    a=5
2    print(a)
3    do
4        local a=7
5        print(a)
6    end
7    print(a)
```

Das kleine Programm druckt also 5, 7, 5 aus, da die lokale Variable a die globale Variable a nicht verändert und am Ende ihrer Gültigkeit „Scope“ einfach wieder vergessen wird, also gelöscht. Lokale Variablen exisitieren nur im Kontext ihrer Erschaffung, innherhalb ihres „scopes“, also innherhalb eines Blockes wie hier oder innerhalb einer Funktion.

Das klingt vielleicht erstmal nicht besonders, bedeutet aber, dass wir das selbe Programm mehrfach und verschachtelt, „rekursiv“ aufrufen können und dass während jedes Aufrufes der innere Zustand dieser Funktion einmalig ist und sich nicht beeinflussen läßt.

Wir kommen darauf zurück, wenn wir uns um Rekursionen kümmern, für den Moment sei nur angemerkt, dass wir ab jetzt versuchen werden, so viele Variablen aus dem global en Bereich heraus zu halten und so viel wir können lokal zu deklarieren. Allein schon, weil der Zugriff des Programmes, also durch Lua, auf lokale Variablen deutlich schneller ist.

Das heißt im übrigen auch, dass die Funktionen, die wir aus Bibliotheken holen beschleunigt werden können, wenn wir sie lokalen Variablen zuweisen. Zum Beispiel gibt es in der „os“ Bibliothek die Funktion „clock“, die die verbrauchte Rechenzeit seit Programmstart zurück gibt.

```
1    local clock=os.clock
2    local start
3
4    konto=0.01
5    start=clock()
6    for jahr=1,2018 do
7        konto=1.035*konto
8    end
9    print(konto, clock()-start)
10
11   local konto=0.01
12   start=clock()
13   for jahr=1,2018 do
14       konto=1.035*konto
15   end
16   print(konto, clock()-start)
17
```

Das Programm ergibt auf meinem Rechner

```
1.411321254786e+28      9.1e-05
1.411321254786e+28      3.2e-05
```

Wir sehen also, dass die globale Zinseszins-Rechnung fast dreimal langsamer läuft als die lokale. Natürlich handelt es sich hier um die Frage, wie reich jemand wäre, der im Jahre Null ein Cent zu einer Bank gebracht hätte, bei einer Verzinsung von 3.5%.

Die Rechenzeit 3.2e-5 bedeutet 0.000032 Sekunden oder 32 Mikrosekunden gedauert, unser Programm hat diese Aufgabe also ein 32 Millionstel Sekunden gelöst und der Zinseszins akkumuliert sich auf einen Goldklumpen von der Masse unseres Planeten, der leider nur eine Masse von nur 5,974e24 kg besitzt.

Aber der Heiland hatte nicht so vorausplanende Eltern und falls er tatsächlich zurück kehrt, wird er leider dreißig und mittellos sein, wie unser Computerprogramm in wenigen Mikrosekunden berechnet hat.

3.4 Das waren alle Keywords!

Und damit haben wir alle Keywords, die Lua kennt, kennen gelernt. Das heißt wir wissen alles! Weltherrschaft! Science!

Theoretisch zumindest.

Kleinen Moment noch, bevor ihr alle weiße Perser-Katzen kauft. Da sind noch ein paar Details, die auf euch warten. Aber so schlimm kann das ja sicher nicht sein.

Willkommen im Kaninchenloch, Alice.

4 Daten-Strukturen

Genauso wie wir unsere Programme in Strukturen formen können, können wir dies mit Daten machen. Bisher kennen wir *nil*, *boolean* und *number,* wobei nil der triviale Fall der leeren Variablen ist. Also eine nicht gesetzte oder gelöschte Variable ist nil.

Außerdem wissen wir inzwischen, dass eine Variable eine Funktion beinhalten kann, dass jede Funktion eigentlich eine Variable ist. Aber der nächste Fall ist uns eigentlichauch schon bekann, da wir mit ihm bereits gearbeitet haben.

4.1 String

Der String ist einfach eine Zeichenkette oder auch ein einzelnes Zeichen.

Wir verketten, also hängen mehrere Strings aneinander mit dem Operator „..“ also zwei Punkten. Im folgenden Beispiel sehen wir das bei `v=s..t` etwa. Und wir können abfragen, aus wieviel Zeichen unser String besteht, indem wir das „#“ davor stellen. Im folgenden Beispiel hat „v“ am Ende die Länge von 48 Zeichen. Dabei werden die Bytes des Strings gezählt und die Steuerzeichen.

Steuerzeichen werden etwa mit „\“ begonnen und danach folgt das Zeichen, \n für „Newline“ oder \t für „Tabulator“ etwa. Einige Zeichen gehen über mehrere Bytes, wie etwa das „ä“. In unserem Beispiel wäre „Hallo“ 5 Zeichen, dann 1 Zeichen für das Newline, „Welt!“ weitere 5 Zeichen, das Leerzeichen ein weiteres Zeichen und ein 36 Zeichen für den Rest, macht 48 Zeichen, wobei wir das „ä“ wie alle Deutschen Sonderzeichen mit 2 Bytes zähle. Der Grund dafür ist, dass „ä“ Unicode kodiert ist und nicht mehr zum reinen ASCII gehört.

ASCII steht hier für American Standard Code of Information Interchange und im Text wurden die beiden betreffenden Wikipedia-Artikel verlinkt, falls jemand mehr dazu wissen möchte.

Es gibt jedenfalls Unicode-Zeichen die bis zu vier Zeichen lang sein können und wir haben auch ein Steuercode, mit dem wir Unicodes erzeugen können in Lua. Das heißt, wir können auch Emoji oder Futhark-Runen, Chinesische, Vietnamesische oder Sprachen jeder anderer Länder umsetzen.

Allerdings ist die Implementation, also die Umsetzung der Unicode-Bereiche unter Windows allgemein jämmerlich, wir müssen vorher besser ausprobieren, ob Windows das entsprechende Zeichen auch druckt. Unter Linux sieht das sehr viel besser aus. Auf einem derzeitigen Kubuntu-System etwa läßt sich ohne Probleme die erst seit wenigenMonaten vom Standardgremium definierten Emoji-Zeichen darstellen. Es ist zu bezweifeln, dass dies innerhalb des nächsten Jahrzehnts unter Windows zu erwarten wäre.

```
> s="Hallo Welt!"
> =s
  "Hallo Welt!"
> s="Hallo\nWelt!"
> =s
  "Hallo\nWelt!"
> print(s)
  Hallo
  Welt!
> t=" Ein weiterer String wird angehängt."
> v=s..t
> =v
  "Hallo\nWelt! Ein weiterer String wird angehängt."
> print(v)
  Hallo
  Welt! Ein weiterer String wird angehängt.
> print(#v)
  48
```

Als 1985 der letzte internationale Zeichen-Standard vor Unicode verabschiedet wurde, benutzten alle Computer diesen Code. Erst 1995 zog Windows 95 nach, allerdings zögerlich, so dass wir erst um das Jahr 2003 herum mit Windows XP allgemein ISO-8859-1 zur Verfügung hatten (nicht das aktuell definierte ISO-8859-15 mit dem €-Zeichen, versteht sich), während bereits im Jahr 1991 bereits ISO-8859 als veraltet galt und Linux von Anfang an Unicode umgesetzt hatte (nämlich 1991).

Unicode auf Windows haben wir etwa seit 2010, wobei das die erste Version ist, die es 1991 gab. Wir können an dieser Stelle mit Fug und Recht ein allgemeines Facepalm machen, den Kopf schütteln und die Situation unter Windows als hoffnungslos beschreiben.

Tatsache ist, dass die Kodierung von Zeichen alle möglichen Standards kennt und alle möglichen Kodierungen. Zeichen. Das Problem des „String“ ist also auf keinen Fall mit einer Handbewegung abzutun. Wir haben einen verlässlichen Standard, aber ob dieser Standard von Windows vor dem allgemeinen Verschwinden dieses Systems von unseren Desktops noch jemals umgesetzt wird, bleibt abzuwarten.

Die Windows Dateisysteme arbeiten selbstverständlich in der Regel nach wie vor auf ISO-8859-1. Also als Programmierer müssen wir uns mental auf einen gewissen Schmerz vorbereiten und auf einige böse Überraschungen. In jedem Fall sollten wir jedes Zeichen besser ausprobieren, bevor wir es benutzen. Gute Nachricht: unsere mobilen Geräte basieren alle auf Linux und arbeiten selbstverständlich durchgehend auf Unicode.

\a	Klingel
\b	Rückschritt (Backspace)
\f	Seitenvorschub (Formfeed)
\n	Zeilenvorschub (Newline)
\r	Wagenrücklauf (Carriage Return)
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator
\\	Backslash, also das „\“ Zeichen selbst
\“	Doppelte Anführungszeichen
\‘	Einfache Anführungszeichen
\ddd	Byte im Dezimal-System
\xXX	Byte im Hexadezimal-System

4.1.1 String Bibliothek

Um besser mit Strings arbeiten zu können, stellt uns Lua eine String-Library zur Verfügung mit etlichen Funktionen. Die Referenz für diese Library finden wir im 3rd Ed Lua Buch, sowie der Lua 5.2 Referenz im Web.

```
> s="Hallo Welt!"
> print(#s)
  11
> print(string.sub(s, 1, 1))
  H
> print(string.sub(s, 2, 2))
  a
> print(string.byte("A"))
  65
> print(string.char(65))
  A
> t="B"
> print(t:byte()-1)
  65
> =s:byte()
  72
> =s:byte(2)
  97
```

Die wichtigste Funktion für den Anfang ist für uns, wie wir ein Zeichen in den ASCII Code wandeln und umgekehrt und wie wir ein einzelnes Zeichen aus einem String holen können. Ein paar kurze Beispiele dazu, ansonsten verweise ich auf die Referenz.

Dazu können wir entweder string.byte(“A“) benutzen oder sobald wir einen String in einer Variablen gespeichert haben, können wir die string-Funktionen auch direkt auf den String mittels „:“ anwenden. Der „:“ ist eine Klassenfunktion, Klassen sind ein Konzept auf das wir später nochmal zurück greifen, zur Zeit ist nur wichtig, dass wir alle Funktionen, die in der string-Library definiert sind auf eine Variable selbst anwenden können

mit diesem „:“. Das ist sehr viel weniger Schreiberei und wir sollten das mit Variablen so halten, allein schon aus Faulheit.

Aber natürlich hält niemanden einen davon ab, jedesmal string.sub(s, 1, 2) zu schreiben statt s:sub(1,2). Es gibt ansonsten nicht viel Geheimnis um den String, der Rest findet sich in der Referenz und man spielt damit einfach etwas herum.

4.2 Tabellen (oder Tables)

Die gute Nachricht ist, was nicht trivial ist wie boolean oder number, was keine Zeichenkette ist wie ein string ist in Lua eine Table. Alles ist eine Tabelle. Der Prozessor kennt eigentlich nichts anderes als eine Tabelle und daher ist die Table wahnsinnig schnell.

Die schlechte Nachricht ist: alles ist eine Table, also damit machen wir alles und das eröffnet eine riesige Menge Möglichkeiten, die wir in diesem Buch gar nicht alle abdecken können.

Die meisten Sprachen unterscheiden an dieser Stelle zwischen Arrays, Tables, Sets, Mulitdimensionalen Arrays, Hashtables, verknüpften Listen, Objekte und Klassen und so weiter. Für uns ist alles eine Tabelle. Und das Schöne ist, dass es eigentlich auch so ist für den Computer.

Das Konzept der Table in Lua ist einfach und genial. Und es klappt eine Menge „Kopf“ auf ein einfaches Konzept herunter, aber das heißt, dass wir im Rahmen der Table auch ein bisschen ausholen müssen. Wir haben deutlich mehr Möglichkeiten Daten zu speichern als bisher und alle Daten die wir bisher hatten, können wir in jeder erdenklichen Weise auch in eine Tabelle packen.

Die Lua-Tabelle ist so schnell implementiert, dass wir häufig Lösungen über dieses Konstrukt programmieren können, die schneller sind selbst als an den Haaren herbei gezogene Lösungen in C oder gar Assembler, zumindest solange wir uns nicht hinsetzen und wirklich wissen, was wir in C oder Assembler tun.

Für einen Anfänger ist die Table daher ebenso interessant wie für einen Profi. Während der letztere sich billig um aufwändige Implementationen drücken kann, um schneller zum Ziel zu kommen, kann sich der Anfänger enorme Leistungsfähigkeit der Sprache ausleihen, ohne dabei genau wissen zu müssen, was eigentlich im Inneren passiert.

Das Problem für Computer ist, dass sie alles sequenziell abarbeiten. Das heißt sucht man etwas in einem String etwa, muss man jedes Zeichen einzeln überprüfen. (Es gibt hier Pattern Matching, was das erleichtert, aber Details). Die Tabelle kann nicht nur sequenziell arbeiten, sondern auch assoziativ. Assoziativ zu arbeiten, also beispielsweise abfragen zu können: „Ich habe hier ‚Deutschland‘, was ist die Hauptstadt?“ wäre assoziativ. Der Computer muss jetzt eigentlich erst alle bekannten Länder durch gehen, bis er „Deutschland“ gefunden hat, dann kann er die Antwort geben. Diese Suche dauert Zeit und ist aufwändig. Es gibt Methoden, Suchalgorithmen, die diesen Zugriff beschleunigen.

Gute Nachricht! Lua kann das und wir merken davon nichtmal was. Assoziationen sind einer der großen Stärken der Tabelle, genau genommen nennt sich das „Hash-Table“, aber statt viel reden einfach ein paar Beispiele. Wir merken uns nur: eine Hash Table hilft uns für einen assoziativen Zugriff, also etwa von einem Begriff zum nächsten zu kommen. Oder von einem Code zum nächsten, wenn wir Kryptographie betreiben.

4.2.1 Array

Das erste was wir mit einer Table machen können, ist ein Array. Ein Array ist eine Variable, die mehrere Felder hat, die mit einem Index durch gezählt werden können. Wir legen eine Table grundsätzlich mit name={} and, also „{…}“ ist eine Table, wenn nichts drin steht, ist es eine leere Table. Tun wir das nicht, haben wir keinen Index und falls wir mit Index drauf zugreifen wollen, gibt es natürlich eine Fehlermeldung.

```
> a={}
> a[1]=3
> a[2]=4
> for i, v in ipairs(a) do print(i, v) end
  1 3
  2 4
> a={"Januar", "Februar", "März", "April"}
> for i, v in ipairs(a) do print(i, v) end
  1 Januar
  2 Februar
  3 März
  4 April
> for i=1,4 do print(a[i]) end
  Januar
  Februar
  März
  April
> =#a
  4
```

Die Länge eines Arrays ist wie beim String mit #array abfragbar und ein Array geht immer von 1 bis zu diesem Wert. Man kann auch 0 als Index nehmen, so wie Arrays in vielen anderen Sprachen durchnumeriert werden, aber dann funktioniert `ipairs()` nicht.

`ipairs()` laufen immer von 1 bis zum maximalen Index, es darf kein Wert nil sein, der Index darf nicht negativ sein. Hat man so eine Art Array, muss man mit `pairs()` drauf zugreifen.

Im gezeigten Beispiel kann man aber auch sehen, dass statt mit `ipairs()` oder `pairs()` man auch einfach durch die Index-Werte des Arrays blättern kann, in diesem Falle `for i=1,4` beziehungsweise, wir hätten auch schreiben können `for i=1,#a`.

In einem Array können wir alle anderen Variablen speichern, natürlich auch weitere Arrays oder besser: Tables. Während wir beim Array üblicherweise eine Zahl haben, die auf einen Inhalt verweist, hier 1 gibt den String „Januar“, ist das die typische Weise wie ein Computer auf etwas zugreift. Adresse zeigt auf Inhalt.

Aber wir wissen ja, dass eine Table auch assoziativ arbeiten kann, also ein Inhalt verweist auf eine Adresse (oder einen anderen Inhalt). Das ist ein Hash-Table und das schreibt sich so.

```
> a={Januar=1, Februar=2, März=3}
> print(a.Januar)
  1
> print(a["März"])
  3
> print(#a)
  0
> for k, v in pairs(a) do print(k, v) end
  Februar 2
  Januar  1
  März  3
>
```

Wie wir sehen können, bei einer Hash-Table, das man auch assoziatives Array, Map oder Dictionary nennt (je nach Programmiersprache), ist die Länge Null. Sprich da es sich nicht durchnumerieren läßt, läßt es sich für Lua nicht auf die Weise berechnen wie es beim Array passiert.

Aber wir können mit dem Inhalt auf den gespeicherten Wert zugreifen, in diesem Falle die Monatszahl, aber es könnten auch die Tage im Monat sein oder irgend ein anderer Inhalt, wie eine Übersetzung oder ein Kodewort etwa, wie wir es bei Kodebüchern vorfinden. „Schiff“ bedeutet vielleicht „Hammer“ oder im Schwarzbuch einer Verbrecherclique kodiert seinen Profi-Killer „Gamaschen-Ede“ vielleicht als „Der Geigenspieler“ oder wie auch immer. Vermutlich kann der Mann gar keine Geige spielen, was nicht überraschen würde.

Wir können aber auch eine Assoziation aufbauen im Sinne von „a“ bedeutet „n“, „b“ ist „m“ wie wir es vom klassischen Cäsar her kennen. Ob das effizienter ist, als die Zielbuchstaben zu berechnen, kann man nicht mit Sicherheit voraus sagen. Man muss das messen, während man ein paar tausend Buchstaben auf diese Weise übersetzt und dabei die Zeit nimmt (mittels `os.clock()` etwa, wie schon gehabt).

Man kann dabei, wie man im Beispiel sieht, die Felder der Hash-Table auf verschiedene Art und Weise ansprechen. Man wählt am besten eine Art, die wenig Schreibarbeit bedeutet. Außerdem, wie man am Ausdruck in der letzten for-Schleife sieht kann man sich nicht drauf verlassen, dass die Elemente in der Reihenfolge in der Hash-Table gefunden werden, in der man sie ursprünglich eingespeichert hat. Hash-Tables kennen keine Sortierreihenfolge, nur Arrays können sortiert werden. Die for-Schleife hat zwei Variablen k(ey) und v(alue), deren Name man natürlich wählen kann, wie man will. Aber key ist in diesem Falle der linke Teil der Assoziation, value der rechte, assozierte Teil.

Super interessant werden diese Assoziationen, wenn es darum geht, bestimmte Zeichen oder Worte zu zählen und so Statistiken aufzubauen, als eines der vielen Beispiele, was man damit machen kann.

Kommen wir zur dritten Anwendung, der Menge, English „Set“. Im Prinzip handelt es sich dabei auch nur um ein Hash wie eben, wobei alle Dinge die vorhanden sind einfach true sind. Als Beispiel eine Liste für den Einkauf.

Der Apfel taucht in der Einkaufs-Liste auf und in der Obst-Liste. Am Ende ist aber nur ein Apfel auf der Liste. Hier wurden also zwei Mengen vereinigt. Man kann mit Hilfe der Table alle Formen der Datenstrukturen verwirklichen, die wir aus anderen Programmiersprachen vielleicht kennen, ohne Ausnahme.

Nur dadurch, dass Lua nahe der Art bleibt, wie der Computer diese Daten am liebsten präsentiert bekommt, ist die Sprache ganz besonders schnell in der Verarbeitung.

```
> obst={Apfel=true, Birne=true, Banane=true}
> erinnerung={Seife=true, Apfel=true, Mehl=true}
> for k in pairs(erinnerung) do print(k) end
Mehl
Seife
Apfel
> for key in pairs(obst) do erinnerung[key]=true end
> for key in pairs(erinnerung) do print(key) end
Birne
Banane
Apfel
Mehl
Seife
```

4.2.2 Table Bibliothek

Genauso wie bei dem String stellt Lua schon eine kleine Bibliothek bereit, um zusätzliche Operationen zu machen. Die Referenz finden wir wie immer in der Lua 5.2 Referenz.

Was wir in dem kleinen Programm benutzen, ist die sub() Funktion, die aus dem String von, bis einen Teilstring heraus schneidet, in diesem Falle von 1 bis 1 von 2 bis 2 weil

wir immer von i bis i wählen. Damit kriegen wir jedes einzelne Zeichen unserer Zeichenkette einzeln.

Leider ist die Ausgabe noch nicht sortiert, aber darum kümmern wir uns gleich. Das einzige, was noch rätselhaft erscheint, ist hier vielleicht dieses seltsame (cnt[char] or 0) +1. Aber das sieht kompilizierter aus als es ist. Solange cnt[buchstabe] noch nicht vorgekommen ist, ist der Wert natürlich nil. Wenn wir einfach nil+1 machen würden, bekämen wir einen Fehler, denn nicht gesetzter Wert plus eins ist? Genau. Lua wirft einen Fehler aus, kann jeder gern selbst probieren.

```
> cnt={}
> text="DERANGRIFFISTIMMORGENGRAUEN"
> for i=1,#text do char=text:sub(i,i); cnt[char]=(cnt[char] or 0)+1 end
> for k,v in pairs(cnt) do print(k, v) end
O 1
S 1
D 1
A 2
E 3
I 3
F 2
U 1
N 3
M 2
T 1
R 4
G 3
```

Statt dessen nehmen wir eine OR Vernüpfung. Wir wissen, dass ODER immer wahr ist, solange nur einer der beiden Seiten wahr ist, also nil ODER 0 ist wahr, egal ob nil nun ungesetzt ist oder nicht. Da Lua „faul“ ist (ja, man nennt es tatsächlich „lazy“) werden alle Ausdrücke nur ausgewertet, solange das Ergebnis nicht ohnehin schon fest steht. Ist der Wert von cnt[buchstabe] bekannt, muss der Ausdruck WAHR sein und Lua nimmt den ersten Wert, ignoriert die 0 und addiert einfach +1. Ist er jedoch nil, testet er den zweiten Wert, der 0 ist und damit nicht mehr „nil“ also bekannt und kann jetzt die +1 addieren.

Wir setzen so unbekannte Indizes in unserer Table. Diese Formulierung ist etwas tricky, aber da das immer das selbe ist, sollte es nicht schwer fallen das selbst so zu machen. Man hätte auch mit „if“ arbeiten können, aber man macht es in Lua halt so. Kann man, muss man aber nicht.

„Default“-Werte setzt man halt so, also Werte die genommen werden, wenn noch keine gesetzt wurden. Es gibt in anderen Sprachen dafür extra sprachliche Formulierungen, aber am Ende ist das auch nur syntaktischer Zucker. Lua verzichtet daher drauf.

```
> tab={}; for k,v in pairs(cnt) do tab[#tab+1]={k, cnt[k]} end
> table.sort(tab, function(a, b) return a[2] > b[2] end)
> for k,v in ipairs(tab) do print(v[1], v[2]) end
R 4
G 3
E 3
N 3
I 3
M 2
F 2
A 2
T 1
U 1
S 1
D 1
O 1
```

Wir kopieren in ein leeres Array „tab“ den Inhalt unserer bisherigen Erfassungen der Zeilen der Buchstabenhäufigkeiten. Mit #tab+1 sorgen wir dafür, dass jeder neue Eintrag ans Ende des Arrays gesetzt wird, also zum maximalen Index+1. Also eine Tabelle mit Tabellen, in denen das erste Feld der Buchstabe, das zweite Feld die Anzahl des Vorkommens dieses Buchstaben ist.

Danach schicken wir das in die table.sort-Funktion die als ersten Parameter das zu sortierende Array nimmt und als zweites Argument eine Funktion mit zwei zu vergleichenden Elementen (Tabellen in diesem Falle) a und b, wobei wir die das zweite Feld (die Anzahl) vergleichen und den Wert dieses Vergleichs zurück geben. Falls die Bedingung nicht erfüllt ist, tauscht der Sortieralgorithmus die Elemente aus, so funktioniert das Sortieren, rein abstrakt, wir kümmern uns überhaupt nicht darum, wie das am Ende implementiert wird. Wir schreiben nur den Vergleich und was wir vergleichen wollen zwischen zwei Elementen.

Dann geben wir die sortierte Tabelle aus, das war‘s: wir haben eine sortierte Häufigkeits-Tabelle, das sollte später nochmal nützlich für uns sein.

Wenn wir nun diese Tabelle haben, wäre es doch nett, wenn wir das mit ein paar Zeichen grafisch darstellen? Es gibt die String-Funktion „rep“, also Repeat, die das für uns erledigt.

```
> for k,v in ipairs(tab) do print(v[1], ("*"):rep(v[2])) end
R ****
G ***
E ***
N ***
I ***
M **
F **
A **
T *
U *
S *
D *
O *
```

Damit wir das „*“ auch mit der Klassenfunktion behandeln können, müssen wir es allerddings in die runden Klammern packen, ihr könnt ja ausprobieren, was passiert, wenn man es nicht macht.

4.2.3 Ein paar kleine Tricks

Um einen String in eine Tabelle von ASCII-Werten zu wandeln und zurück, gibt es ein paar Tricks, die wir uns merken sollten.

```
> s="abcdefghijklmnopqrstuvwxyz"
> s=s:upper():rep(10) -- in Großbuchstaben und 10x wiederholen
> t={s:byte(1,-1)} -- den String in bytes wandeln von Anfang(1) bis Ende (-1)
> --wir haben jetzt eine Tabelle mit den ASCII Werten des Strings, wir können das auch zurück wandeln
> v=string.char(unpack(t))
> print(v:sub(1,10)) -- die ersten zehn Buchstaben drucken
  ABCDEFGHIJ
> for i=1,10 do print(t[i]) end
  65
  66
  67
  68
  69
  70
  71
  72
  73
  74
```

Wie wir sehen, können wir bequem unsere Strings in Tabellen mit ASCII-Codes wandeln und zurück. Natürlich können wir das selbe auch mit einer „for“-Schleife machen, aber Programmierer sind faul. Wir machen nichts, was die Maschine für uns nicht abnehmen kann.

Der gewiefte Programmierer, merkt sich die Tricks und spart sich das Schreiben. Wer diese kleinen Tricks nicht lernt, schreibt und schreibt und schreibt. Es lohnt sich also, ein wenig die Bibliotheken zu durchsuchen und auszuprobieren, ob es nicht irgend eine Abkürzung gibt. Das Bespiel für solche Abkürzungen hatten wir schon mit der Formel, die durch faule Auswertung uns ein if-then-else erspart hat.

Man muss bei solchen Verkürzungen immer abwägen, was gut für die Lesbarkeit ist und was nicht. Nicht jede Verkürzung ist besser, Programmieren ist eine Kunst, die eine Menge mit Ästethik und einem Gefühl für das gute Aussehen und die richtige Lösung zu tun hat. Und dann ist da noch, dass einige Lösungen, die gut aussehen vielleicht langsam sind.

Es gibt Programmierer, die sich stundenlang über die Form eines Programmes streiten können und für die diese Form am Ende wichtiger als die Funktion ist. In der Praxis jedoch gibt es meistens keine endgültig „richtige“ oder „falsche“ Lösung. Weswegen Programmierer unterschiedlicher Stile durchaus im Stande sind, sich heiße Wortgefechte über den einen Stil oder den anderen zu liefern.

4.2.4 Optimierung und Rationalität

Doch die letzte Richterin ist die CPU: was ihr gefällt, wird schneller ausgeführt. Und die CPU gibt Recht.

Daher testen wir die verschiedenen Möglichkeiten aus und machen uns ein eigenes Bild von der Lage. Was auf dem einen System (Prozessor) schnell ist, kann auf einem anderen langsam sein oder mit der Version von Lua oder ob wir eine JIT Version benutzen oder nicht kann es sich verändern.

Manche Dinge aber bleiben gleich. Am Ende ist das Programmieren mehr als nur eine Ingenieurstätigkeit, es ist eine Kunst, wie ein guter Dichter ein Gefühl für seine Zeilen bekommt, bekommt ein guter Programmierer ein Gefühl für gute oder schlechte Lösungen für ein Problem.

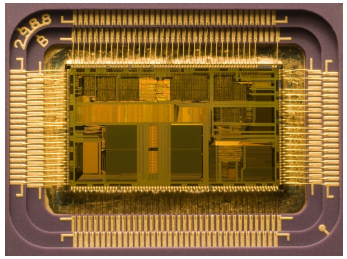
Lesbarkeit, Kürze und Ausführgeschwindigkeit sind manchmal im Widerspruch stehende Werte, die wir manchmal meinen nicht lösen zu können, bis wir den Kode von jemanden sehen, der es dennoch geschafft hat.

Wir sind vielleicht noch eine Weile entfernt davon, wirklich kunstvollen Kode zu schreiben, aber wir müssen dies im Hinterkopf behalten, wenn wir jemals dort hin kommen wollen. Gute Wissenschaft ist immer auch Schönheit und Poesie und hat Tiefe, Wahrheit und birgt Erkenntnis.

Welche Dinge mag ein Prozessor und welche nicht so gern? Der Prozessor mag keine Funktionsaufrufe. Die verlangsamen den Ablauf. Also wenn wir welche schreiben, dann

sollten sie sich lohnen und wir sollten eine Funktion immer auch etwas tun lassen, was die CPU für eine Weile beschäftigt. Allzu zerstückelter Code ist auch schwer zu lesen.

Die CPU mag keine Divisionen. Alles was mit Division zu tun hat, ist teilweise hundert mal langsamer als eine Addition. Das selbe gilt für Modulo. Wenn wir also dividieren, sollten wir gut überlegen, ob man das auch anders lösen kann. Wenn wir mit Modulo arbeiten, gibt es manchmal eine wesentlich schnellere Lösung.



Die verwöhnte kleine Prinzessin, die CPU mag einfache Operationen, mag es, wenn Daten in Arrays liegen und dicht zusammen gehalten werden. Sie mag es, wenn Daten in verdaulichen Happen verarbeitet werden, anstatt in riesigen Stücken, die sie nicht schlucken kann. Artemis spickt denjenigen mit ihren Pfeilen, der konstante Werte innerhalb von Schleifen immer wieder neu berechnet, ebenfalls äugt ihr Bruder misstrauisch auf die triviale Schleife, die innerhalb einer kostspieligen Schleife (auch die Schleife selbst kostet Zeit!) nichts als eine einfache Addition ausführt. Dem dessen Schleife mehr Arbeit kostet als an Arbeit darin verrichtet wird, dem zürnen die Götter der Informatik!

Die Programmierer kämpfen um die Gunst der CPU und versuchen es der kleinen Prinzessin leicht zu machen. Das erzeugt schnelle und kurze, elegante Programme. Am Ende hat das schnelle Programm recht. Wir denken manchmal, dass wir eine besonders elegante Lösung gefunden hätten, aber erst wenn wir unser Programm gegen die Zeit messen, wissen wir, ob wir richtig liegen.

Daher ist

```
start=os.clock()
…
zeit=os.clock()-start
```

dieser kleine Kode für alle unsere Programme die gerechte und faire Messlatte, ob unser Kode etwas taugt, oder ob wir daran noch arbeiten müssen, nicht der Ausbilder, nicht der Lehrer, nicht der Vorgesetzte, die Eltern; wir haben einen gerechten Richter: die Realität selbst. Wir selbst sind das Maß und wir messen uns selbst, um über uns selbst hinaus zu wachsen. Die Welt des Programmierens ist die des Rationalen; es gibt ein Richtig und ein Falsch. Und an dieser Messlatte können wir wachsen, dieses `true` und `false` ist das einzige Werkzeug, mit der die Natur für Evolution sorgt, für Entwicklung, diese Methode ist für uns Informatiker die Methode, die Wissenschaft, die Technologie voran zu treiben und für ein besseres Leben für unsere Mitmenschen zu sorgen. Für ein besseres Leben, Nahrung für alle, Medizin und ein Kampf der Krankheit und Seuche, für Frieden und eine bessere Welt. In den letzten dreißig Jahren gab es in der Forschung und Technologie, in der Medizin, Geschichte, selbst in der Philosophie und Mathematik keinen Fortschritt mehr, der nicht durch Computer berechnet oder durch sie umgesetzt wurde.

Der Computer ist der Dietrich des menschlichen Wissens. In der Vergangenheit wurden große Entdeckungen vielleicht noch ohne das gemacht, in der Zukunft ist die künstliche Intelligenz und unsere Fähigkeit, sich dieser als Werkzeug zu bedienen der bestimmende Faktor. Der Computer ist zum wichtigsten Werkzeug der Menschheit geworden, wie das Feuer es einst für den Steinzeitmenschen gewesen ist. Die Kunst der Beherrschung oder zumindest das Verständnis bedeutet Leben oder Tod.

Das mag übermäßig optimistisch klingen, aber es läuft tatsächlich darauf hinaus. Nicht alles, was durch Technologie erreicht wurde war nur zum Vorteil der Menschheit, wie es mit jedem mächtigen Werkzeug ist, aber ohne Zweifel ist dies die richtige Richtung. Und auch wenn vielleicht nicht alle Menschen in ihrem Leben Programmierer werden oder Programmierer werden wollen, muss jeder erkennen, dass er fähig dazu ist, mit dieser Methode neues zu erschaffen, zum Schöpfer zu werden, der Zauberspruch der Technologie, der mit einem einfachen Wort, dem Programm, die Welt bewegen kann.

Die Geschichten, die Märchen über Magie und Zauberei, die sind wahr. Sie sind wahr geworden durch die Computer und unsere Zaubersprüche, die Programme. Es ist der Beweis der wissenschaftlichen Methode: es funktioniert.

Programmierung, das ist nicht nur im stillen Kämmerchen vor sich hinschreiben. Wir kommunizieren mit anderen, wir teilen unsere Erkenntnisse, wir „sharen“, wir geben und nehmen der Gemeinschaft, wir stehen immer auf den Schultern von Riesen und wir müssen begreifen, dass wir nicht allein sind. Dass da andere sind, die die selben Träume teilen, die selben Visionen haben, die selben Ziele zu erreichen versuchen. Und der Computer verbindet uns alle zu einem gewaltigen Welt umspannenden Netzwerk.

Wir können spielen mit Lua, mit http://Love2D.org , wir können Online-Games spielen bei http://roblox.com , wir können den Kode von anderen einsehen bei http://rosettacode.com und wir können von einfachen Skripten bis zur künstlichen Intelligenz, von Datamining aus Webseiten bis zum Cheaten in Spielen, Computermusik, Robotik oder Heimautomation alles umsetzen, was wir an Ideen haben.

Der Wille ist das einzige, was wir dafür aufbringen müssen. Wir können einen Unterschied machen. Wir als einzelne menschliche Wesen können einen Unterschied machen für die ganze Welt. Denn wir sind Magier geworden.

4.3 Andere Bibliotheken

Die math-Bibliothek versorgt uns mit mathematischen Funktionen, zum Beispiel für die Kryptologie ist math.random() sehr nützlich und mit random(x,y) können wir einen Zufallswert zwischen x und y erhalten. Wir starten diesen Generator mit randomseed(x), wobei die selbe Pseudo-Zufallszahlenfolge aus dem selben Startwert erzeugt wird. Wollen wir einmalige Werte, benutzen wir os.time() als Startwert oder eine Mischung aus time(), date() und clock().

Die bit32-Bibliothek liefert uns weitere nützliche Funktionen, wie etwa die bxor() Funktion, mit der wir außer dem uns bisher bekannten plus und minus eine interessante Verknüpfung von einem Text mit einem Passwort vornehmen können, die Exklusiv-Oder Funktion ist uns ja schon bekannt: diese arbeitet aber gleich mit 32 bit auf einmal statt mit nur einem Bit, also einem einfachen „true“ oder „false“-Wert..

Und schließlich haben wir die io-Bibliothek, die uns hilft Dateien zu lesen und zu schreiben und so Daten in unser Programm zu lesen oder zu speichern. Mehr dazu aber später, sobald wir es brauchen. Wir finden viel dazu auch im 3rd Edition Buch für Lua.

```
> =string.byte("A")
65
> =bit32.bxor(string.byte("A"), string.byte("B"))
3
> =bit32.bxor(3,string.byte("B"))
65
>
```

5 Kryptologie

Kryptologie ist ein Überbegriff über die Kunst des verschlüsselten Schreibens „Kryptographie“ sowie der Kunst, verschlüsselte Texte zu entschlüsseln „Kryptoanalyse“, es umfasst aber auch die Technik des geheimen, versteckten Schreibens, der Steganographie durch verbergen mit unsichtbarer Tinte, versteckten Markierungen oder geschickte Veränderungen im Text, um den Inhalt zu verschleiern.

5.1 Monoalphabetische Verschlüsselung

5.1.1 Cäsar Chiffre

Die erste Verschlüsselung, die wir kennengelernt haben, war der Cäsar Chiffre. Julius Cäsar hat damit seine Nachrichten verschlüsselt und auch wenn diese Methode denkbar einfach ist, wurde sie für Jahrhunderte angewendet. Für die Cäsar Verschlüsselung verschieben wir das Chiffre-Alphabet gegen das Text-Alphabet um 13 Buchstaben.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
NOPQRSTUVWXYZABCDEFGHIJKLM
```

Diese Art Verschlüsselung nennt sich monographische, monoalphabetische Verschlüsselung, weil jeweils ein Text-Buchstabe (monografisch) zu einem Schlüsselbuchstaben verschlüsselt wird und wir nur ein einziges Schlüsselalphabet dafür benutzen.

Außerdem ist der Cäsar Schlüssel auch noch invertierbar, also die Verschlüsselung ist gleichzeitig die Entschlüsselung. Verschlüsseln wir anders als mit Cäsar, auch „Rot13“ genannt, ist das nicht so. Im Falle von Rot3, wenn wir A gegen D austauschen, wird nicht wie oben A gegen N auch N gegen A ausgetauscht.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
```

Dieser Schlüssel ist also nicht invertierbar. Wir müssen für das Entschlüsseln von unten nach oben arbeiten, also die Schritte rückwärts ausführen. Es gibt zwei grundsätzliche Methoden, wie wir das in Lua umsetzen können. Die eine Methode ist, dass wir die Buchstaben in ihrer Position durchzählen. Also A wird 0, B wird 1 und so weiter und dann einfach 13 oder 3 oder welchen Schlüsselwert wir wählen dazu addieren.

Was tun wir aber, wenn wir über die 25 kommen, um wieder von vorn anzufangen?

Die zweite grundsätzliche Methode, die wir wählen können, ist mit einer Tabelle zu arbeiten. Entweder können wir eine eine Tabelle nehmen, in der jedes Array unsere gewünschten verschlüsselten Buchstaben besitzt oder wir können auch eine Hashtable benutzen, in der wir nicht mit der festen Position arbeiten, sondern wie in einem Kodebuch Buchstabe gegen Buchstabe vertauschen können.

Beide Methoden funktionieren, aber welche Methode ist besser? Was ist der Vorteil gegen die erste Methode der Addition? Und wie können wir wieder von vorn anfangen, sobald unsere Buchstaben samt Addition größer als 25 werden? Gibt es nur eine Methode? Falls es mehrere gibt, was sind die Vorteile der beiden Versionen?

Wir müssen das ausprobieren und die Zeit messen, die unser Programm braucht, um einen Test-Text zu übersetzen. Wir schreiben dafür eine Funktion, die als Eingabe den Ausgangstext hat und die Kodenummer oder eine Entschlüsselungstabelle. Als Rückgabe haben wir wieder einen String mit dem verschlüsselten Ergebnis.

Wenn wir ein nicht symetrisches Verfahren haben, brauchen wir entweder eine zweite Funktion, die entschlüsselt oder einen Parameter, der uns sagt dass Ver- oder Entschlüsselt werden soll. Was ist besser?

```
function caesar(text, key, decrypt)
      …
      return cipher
end
```

Oder sollten wir lieber eine Funktion encrypt() und decrypt() machen?

```
function caesar.encrypt(text, key)
      …
      return cipher
end
```

Wir programmieren unse erste Schlüsselfunktion!

5.1.2 Erst noch ein paar „Utilities“, Hilfsroutinen

```lua
function string.filter(input, pat, rep)
    pat = pat or "[%s%p%c]+"
    rep = rep or ""
    return input:gsub(pat, rep)
end
function string.filter(input, pat, rep)
    pat = pat or "[%s%p%c]+"
    rep = rep or ""
    return input:gsub(pat, rep)
end

function string.block(text, block, line)
    local block = block or 5
    local line = line or 60
    local len = 0
    local output = text:gsub(string.rep("%C", block), function(t)
        len = len + #t+1
        if len>line then
            len=0
            t = t.."\n"
        else
            t = t.." "
        end
        return t
    end)
    return output
end
```

Um unsere Texte schön ausgeben zu können, machen wir uns noch zwei kleine Hilfsfunktionen filter() und block(). Filter filtert aus dem Eingangs-String alle Zeichen heraus, die nicht ins Standard-Alphabet gehören, sowie alle Leerzeichen oder Zeilenvorschübe. Wir speichern dieses kleine Programm unter „filter.lua“.

```
> require "filter"
  true
> s="DER ANGRIFF IST IM MORGENGRAUEN!"
> print(s:filter():block())
  DERAN GRIFF ISTIM MORGE NGRAU EN
```

Block() formatiert den Ausgabestring in Zeilen von 60 Zeichen und formatiert sie in 5er-Blöcke, so wie es in der Kryptologie üblich ist. Wir importieren diese Funktion mit „require“.

Eine weitere String-Funktion könnte sich für uns als nützlich erweisen, die „shuffle“-Funktion, die einen String durchmischt. Wir schreiben diese Funktionen deshalb direkt der „string“-Bibliothek zu, damit wir nicht nur mit string.filter(text) arbeiten können, sondern auch direkt mit text:filter(). Wie wir bereits wissen, ist der „:“ nur syntaktischer Zucker für den Aufruf einer Funktion mit sich selbst als Argument. Das heißt `text:filter()` ist das selbe wie `text.filter(text)` oder `string.filter(text)`.

```
1  function string.shuffle(text)
2      local t={text:byte(1,-1)}
3      local random=math.random
4      for i=#t,1,-1 do
5          local rnd=random(i)
6          t[i], t[rnd] = t[rnd], t[i]
7      end
8      return string.char(unpack(t))
9  end
10
```

Wie würde die Shuffle (also „Mischen“) Funktion aus für eine reine Tabelle statt für einen String? Und wo würden wir sie hineinschreiben, um sie möglichst faul nutzen zu können? Natürlich dürfen wir nicht vergessen, die `math.randomseed()` vorher zu initialisieren, wenn wir einen echten Zufallswert haben wollen, eignet sich `os.time()*os.clock()` ganz gut dazu, ansonsten kriegen wir innerhalb einer Sekunde immer alle die selbe Verwürfelung bei der selben Initialisierung.

```
> require "shuffle"
  true
> s="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
> print(s:shuffle())
  SHFYDXGBJNZIACOUQPMETLVWKR
```

Wenn wir übrigens weiterhin auf der Konsole von Zerobrane arbeiten wollen und wir haben einmal ein „require“ geschrieben, verändern aber das Programm dazu, müssen wir den Require zurück nehmen, um die Routinen zu aktualisieren. Das ist nicht vorgesehen, aber wir können uns schnell noch ein weiteres „Utility“ schreiben, das das für uns macht. Das gehört eigentlich nicht zu den kryptographischen Problemen, aber so ist das eben, dass man als Informatiker von einem Problem ins andere stolpert. Das Problem zu lösen lenkt von unserem eigentlichen Ziel ab, daher hier nur schnell die Lösung, ohne tiefer darauf ein zu gehen, welche Innereien von Lua wir an dieser Stelle mit dem Hackmesser bearbeiten.

```
1  function unrequire(m)
2      package.loaded[m] = nil
3      _G[m] = nil
4  end
```

Wir stressen langsam die Möglichkeiten unserer lokalen Konsole und werden ab jetzt und in Zukunft möglichst gleich richtige Programme schreiben.

Ein weiteres Hilfsmittel ist für den Fall, dass wir nicht nur ASCII A-Z lesen wollen, sondern auch Sonderzeichen, also Unicode. In der Datei unicode.lua oder in unserem Proramm speichern wir.

```
function string.allunicode(text)
    return text:gmatch("([%z\1-\127\194-\244][\128-\191]*)")
end
```

Und wir würden diese kleine Hilfsfunktion benutzen nach einem require etwa. Stattdessen können wir natürlich auch auf die reichhaltigen Bibliotheken für Lua zurückgreifen, in diesem Falle https://github.com/Stepets/utf8.lua sowie die reichhaltigen Resourcen, die uns zur Verfügung stehen, zum Beispiel http://lua-users.org/wiki/LuaDirectory

```
require "unicode"
txt="Köhlerhütte"
for c in txt:allunicode() do
    print(c)
end
```

5.1.3 Beispiele von Cäsar

Es gibt einige Varianten zum Cäsar-Chiffre. Der klassische Rot13 kennt außerdem Rot1 bis Rot25, mit Rot0 würde natürlich keine Verschlüsselung stattfinden, ein sogenannter „Null Cipher“. Es gibt die Variante der Porta-Scheibe von 1563, die das Alphabet in Sonderzeichen gewandelt hatte, bekannt ist auch die Alberti-Scheibe nach Leon Battista Alberti.

Alle Chiffrier-Scheiben oder Stäbchen oder einfache Chiffrierschieber arbeiten mehr oder weniger nach dem Cäsar-Prinzip. Wir können das Chiffrieralphabet wie im „ROT“-Verfahren verschieben oder umkehren (rückwärts schreiben von Z…A), wir können es komplett Mischen, dafür haben wir bereits die „shuffle“ Funktion bereit liegen, oder wir können eine der anderen bekannten Methoden benutzen, die die Geheimschreiber aus der Geschichte ausgeknobelt hatten, um ihre Korrespondenz vor den „Black Chambres“, die frühen Kodebrecher-Geheimagenten, zu schützen.

Ein einfaches Cäsar ist aber immer knackbar, egal wie viel Mühe wir uns geben mit dem Chiffrier-Alphabet. Hier ein Englischer Text eines bekannten Autors aus dem Jahre 1843.

```
53‡‡†305))6*;4826)4‡.)4‡);806*;48†8¶60))85;1‡(;:‡*8
†83(88)5*†;46(;88*96*?;8)*‡(;485);5*†2:*‡(;4956*2(5*
-)8¶8*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡1;48†85;4)
485†528806*81(‡9;48;(88;4(‡?34;48)4‡;161;:188;‡?;
```

Die normalen Lettern gegen andere auszutauschen, hat lange Tradition, ein typisches Beispiel ist das Freimaurer-Alphabet etwa, bietet aber keinerlei zusätzliche Sicherheit. Cäsar wurde noch 1915 von der Russischen Armee benutzt, da den militärischen Stäben ein schwierigeres Verfahren nicht zuzumuten wäre, Diplomatenstäbe sind oftmals ähnlich kompetent, dafür gibt es viele historische Beispiele. „Military Intelligence“ kann man daher mit Fug und Recht behaupten, sei ein Oxymoron, ein Widerspruch in sich. Ein weiterer Englischer Text.

```
AOLYL'Z H IPN THU DLHYPUN H DOPAL ZBPA HUK WHALUA
SLHAOLY ZOVLZ OL DHUAZ AV AHRL OPZ TVURLFZ AV ZLL AOL
RPKZ HA AOL GVV.
```

Ebenfalls ist das Spielzeug „Barbie-Typewriter“ überraschenderweise fähig, mit einem Cäsar-Chiffre Texte zu verschlüsseln, es gibt allerdings nur vier mögliche Kode-Einstellungen. Oder im Comic GirlGenius gibt es eine sehr interessante monoalphabetische Variante, die Kodes enthält, um die Chiffiermethode zu verändern. Es gibt immer noch haufenweise ungelöste Kryptogramme (verschlüsselte Texte) aus historischen Briefen, Schatzkarten und dergleichen, die auf Entschlüsselung warten. Viele von diesen Texten sind überraschend einfach verschlüsselt. Vielleicht gibt es sogar reale Schätze, die auf einen warten? Auf jeden Fall jedoch Ruhm.

Erfahrungsgemäß reden die Menschen mehr über das Entschlüsseln als sie es tatsächlich tun.

```
UDXDL DVLDN MDABS DXDAV BAMLD VKJKA DNJBZ DNENJ HEGJU DXDEJ HMDBL
VJTAX VLDBB LADMQ XNAYE AMEDA BMEDA BLDBJ HMLDB XJDBL DBJHY MSDNJ
EDUOX TVJBM QXTDH LDNEH BLBAQ XEUDA MMUDB DNPJB BENDY YDB
```

Pedro Calderón de la Barca y Barreda González de
Henao Ruiz de Blasco y Riaño,
(1600 - 1681), spanischer Dichter

5.1.4 Implementation des Cäsar-Algorithmus

Unser Ziel ist eine Funktion, die als Parameter einen Text-String hat und den „Key“, also den Verdrehungsgrad der Chiffrierscheibe. Die Funktion soll einen Chiffre-String zurück geben, den wir weiter verwenden wollen.

Für den Anfang benutzen wir ausschließlich Zeichen von A-Z, in Großschrift. Bei jeder Implementation fangen wir mit der einfachsten Variante an und Schritt für Schritt erweitern wir sie, um das gewünschte Ergebnis zu bekommen. Wir lassen daher vorerst Sonderzeichen und Leerzeichen oder andere Symbole weg und benutzen ausschließlich Lettern.

Als erstes müssen wir die einzelnen Zeichen aus dem String bekommen. Da jedes Zeichen in einem Computer auch eine Zahl ist, brauchen wir für Cäsar den String nicht in Zeichen zu zerschneiden, wir sind besser bedient mit dem ASCII-Code. Der ASCII Code ist auf einer Tabelle, siehe Wikipedia, definiert, so dass „A“ etwa der Zahl 65 entspricht, „Z“ der Zahl 90.

5.1.5 Cäsar mit Addition

Ein kleines Programm „caesar_uebung1.lua“ testet das für uns.

```lua
-- caesar_uebung1.lua
text="DERSCHATZLIEGTIMSILBERSEE"
-- wandle das erste Zeichen des Strings
print("1:", string.byte(text, 1))
-- diese Formulierung tut das selbe, aber einfacher
print("2:", text:byte(1))
--[[ in einer Schleife kriegen wir alle die for-Schleife läuft
von 1 bis zur Länge des Strings #text, wir benutzen hier
io.write(), was das selbe wie print() ist, aber ohne
Zeilenvorschübe]]
io.write("3:\t")
for i=1,#text do
    io.write(text:byte(i), " ")
end
print()
-- wir wandeln ASCII in A=0, B=1… Z=25
io.write("4:\t")
local A=("A"):byte() -- A=65
for i=1,#text do
    io.write(text:byte(i)-A, " ")
end
print()
-- wir addieren unseren Schlüssel
io.write("5:\t")
local key=13
for i=1,#text do
    io.write(text:byte(i)-A+key, " ")
end
print()
--[[ um die Werte "umzuklappen" benutzt man in der Regel die
"Modulo" Operation, das ist der Rest einer Division z.B.
10%3 -> 1 (10/3=3 Rest 1) Ein x%26 klappt die Werte also immer
auf 0…25
]]
io.write("6:\t")
for i=1,#text do
    io.write((text:byte(i)-A+key)%26, " ")
end
print()
```

Die Ausgabe von caesar_uebung1.lua lautet:

```
1:  68
2:  68
3:  68 69 82 83 67 72 65 84 90 76 73 69 71 84 73 77 83 73 76 66 69 82 83 69 69
4:  3 4 17 18 2 7 0 19 25 11 8 4 6 19 8 12 18 8 11 1 4 17 18 4 4
5:  16 17 30 31 15 20 13 32 38 24 21 17 19 32 21 25 31 21 24 14 17 30 31 17 17
6:  16 17 4 5 15 20 13 6 12 24 21 17 19 6 21 25 5 21 24 14 17 4 5 17 17
```

Wir haben also grundsätzlich unsere Formel `(text:byte(i)-A+key)%26` mit A=65 und key=13 in diesem Falle. Jetzt müssen wir die Zahlen wieder in Zeichen wandeln.

Die Modulo-Funktion ist prinzipiell eine Division und als solche ist sie relativ aufwändig, verglichen mit Addition, Subtraktion oder einer if-Abfrage, die jeweils nur etwa einen Zyklus kosten, eine Division kann bis zu 200 kosten, in diesem Falle aber wohl weniger. Es kommt auch stark auf den Prozessor an und die Art von Computer, die wir benutzen, nur sei hier angemerkt, dass bei Verwendung einer Division immer eine misstrauische Augenbraue nach oben zieht und ein spontan heraus geplatztes, tolldreistes: „das geht besser!“, hat meistens Recht und hat so manchem verdienten Programmierer die Schamesröte ins Gesicht getrieben, wenn ihn ein Anfänger mit dieser einfachen Erkenntnis deklassiert. Wir finden Modulo in den Referenzlösungen auf Rosettacode und in allen Unterlagen, die Schulen und Universitäten zur Verfügung stehen…

Wir kümmern uns später nochmal um die Optimierung. Das erste Ziel ist immer, dass das Programm überhaupt läuft, zumal es unser erstes ist. Programmierer sollten von Anfang an an guten Kode gewöhnt werden. Lieber kein Beispiel als ein schlechtes.

Der nächste Schritt ist natürlich aus unseren Zahlen wieder ASCII zu machen. Die string-Bibliotheksfunktion „char“ tut dies für uns.

```lua
local text="DERSCHATZLIEGTIMSILBERSEE"
print(text)
local A=("A"):byte() -- A=65
local key=13         -- "Standard" Cäsar
for i=1,#text do
    -- unsere Formel von Übung 1
    local c=(text:byte(i)-A+key)%26
    c=c+A    -- 0…25 -> 65…90 (ASCII)
    io.write(string.char(c))
end
print()
```

Unsere caesar_uebung2.lua gibt folgenden Text aus:

```
DERSCHATZLIEGTIMSILBERSEE
QREFPUNGMYVRTGVZFVYOREFRR
```

Und damit haben wir die Vorbereitungsschritte fast abgeschlossen. Es bleiben zwei Probleme, das erste ist, wir wollen unsere Zeichen nicht einfach auf den Bildschirm ausgeben, sondern wir wollen sie in einem neuen String speichern. Der zweite Schritt ist, dass wir unser Programm auch wiederverwenden können wollen und für Wiederverwendung von Kode benutzt man Funktionen, also die „function“.

Um eine Funktion benutzen zu können, muss sie vorher definiert werden, also Funktionen stehen immer am Anfang von Programmen, der „Arbeitskode“, der auf sie zugreift am Ende. Das ist in vielen Sprachen so, nicht in allen, aber es hat sich durchgesetzt es möglichst so zu halten, als Konvention, selbst wenn die Sprache es eigentlich nicht erzwingt.

Das erste Problem können wir mit dem Operator „..“ lösen. Das wäre eigentlich alles, gäbe es hier nicht ein kleines Problem. Für jedes Zeichen legt Lua (und alle anderen Hochsprachen) dafür einen neuen String an. Das ist langsam. Daher werden wir das erst einmal so machen, aber gleich darauf einen weitaus schnelleren Weg wählen.

Aber erst Implementation, dann Optimierung:

```lua
-- caesar_uebung3.lua
function caesar(text, key)
    local A=("A"):byte() -- A=65
    local cipher=""
    for i=1,#text do
        local c=(text:byte(i)-A+key)%26
        c=c+A   -- 0…25 -> 65…90 (ASCII)
        cipher=cipher..string.char(c)
    end
    return cipher
end

local text="DERSCHATZLIEGTIMSILBERSEE"
print(text)
--[[ key 13, A->N N->A heißt "invultorisch"
also umkehrbar. Alle anderen Schlüssel sind nicht
umkehrbar, abgesehen vom Null-Schlüssel 0, der
den unverschlüsselten Originaltext erhält z.B.
key=1 A->B aber B->C wir müssten hier key=-1
nehmen.
]]
local key=13 -- Variablen nur einmal deklarieren!
local encrypt=caesar(text, key)
print(encrypt)
local decrypt=caesar(encrypt, key)
print(decrypt)
-- nicht invultorischer Schlüssel
print("Nicht invultorisch")
key=1        -- nicht mehr deklariert!
print(text)
encrypt=caesar(text, key)
print(encrypt)
decrypt=caesar(encrypt, key)
print(decrypt)
print("Aber mit -key")
decrypt=caesar(encrypt, -key)
print(decrypt)
```

Die Ausgabe von caesar_uebung3.lua ist

```
DERSCHATZLIEGTIMSILBERSEE
QREFPUNGMYVRTGVZFVYOREFRR
DERSCHATZLIEGTIMSILBERSEE
Nicht invultorisch
DERSCHATZLIEGTIMSILBERSEE
EFSTDIBUAMJFHUJNTJMCFSTFF
FGTUEJCVBNKGIVKOUKNDGTUGG
Aber mit -key
DERSCHATZLIEGTIMSILBERSEE
```

Dass Cäsar in der Regel nicht invultorisch ist, ist logisch. Der Cäsar Chiffre ist im Prinzip eine einfache Addition. Addition ist nicht invultorisch. Also 4+5=9, 9+5=14. Cäsar wird nur dann invultorisch, wenn der Schlüssel gerade key=-key über den Modulo des Zeichenvorrats vom verwendeten Alphabet ist.

Eine kleine Anmerkung zu unserem Alphabet. Die alten Römer hatten ein 20 Zeichen Alphabet, kein 26 Zeichen Alphabet, wie wir es kennen, sondern einen geringeren Zeichenvorrat. Der originale Cäsar-Schlüssel müsste also 10, nicht 13 gewesen sein.

```
      Original Römisch:
20: ABCDEFGHILMNOPQRSTVZ
      Ab 1600 kommen in Europa dann KWXY dazu, mit K einem
      scharfen (Germanischen) C, ein W als Ligatur eines
      weichen Doppel-VV, X als Kürzel für CS und Y als eine
      spezielle Form des I
24: ABCDEFGHIKLMNOPQRSTVWXYZ
      Im 18ten Jahrhundert kam das U dazu
25: ABCDEFGHIKLMNOPQRSTUVWXYZ
      Erst um 1900 kam dann noch das J dazu
26: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Die Reihenfolge des Alphabets war auch nicht immer die selbe, zum Beispiel um 1466 auf der Alberti-Scheibe und bei Trimetius, der uns nochmal begegnen wird, stand das W für das „Omega“, das letzte Zeichen des Alphabets. Das W ist in dieser Zeit das letzte Zeichen im Alphabet, nach Z, weil es auch erst grade neu dazu gekommen war.

Will man alte Chiffren entziffern aus der Piratenzeit, sind diese Kleinigkeiten recht nützlich. Es liegen tatsächlich noch Schätze dort draußen, die heute viele Millionen wert wären. Ebenfalls kann man sich überlegen, wenn man einen eigenen Kode erfindet, ob man nicht ein reduziertes Alphabet benutzen will, um den üblichen Methoden der Analyse der Buchstabenhäufigkeit ein kleines Schnippchen zu schlagen.

Außerdem, wenn wir genau drüber nachdenken, merken wir, dass wir so eine Cäsar-Verschlüsselung mit unserer ASCII-Additions-Methode nicht implementieren können. Da müssen wir zu anderen Methoden greifen.

Aber erst einmal das Problem des „..“.

Lua legt für jedes dieser Schritte zwei Strings an, erst den `string.char(c)` und dann erzeugt er einen neuen aus diesem und `cipher` und weist dies dann wiederum `cipher` zu.

Es gibt einen einfachen Weg, das zu optimieren. Man speichert alle erzeugten char() in einer neuen Tabelle und fügt die Tabelle am Ende mit einem einzelnen Befehl zu einem String zusammen. Klingt einfach? Ist es auch!

```lua
-- caesar_uebung4.lua
function caesar(text, key)
    local char=string.char
    local A=("A"):byte() -- A=65
    local cipher={}
    for i=1,#text do
        cipher[i]=char((text:byte(i)-A+key)%26+A)
    end
    return table.concat(cipher)
end

local text="DERSCHATZLIEGTIMSILBERSEE"
print(text)
local key=13
local encrypt=caesar(text, key)
print(encrypt)
local decrypt=caesar(encrypt, key)
print(decrypt)
```

Wir haben hier die lokale Variable „c“ wegoptimiert und das ganze in eine Zeile gepackt. Außerdem haben wir „string.char“ in eine lokale Variable „char“ gepackt, weil diese sehr häufig aufgerufen wird. Und lokale Variablen sind etwa dreimal schneller als globale. Das gilt natürlich auch für Funktionen, die nichts weiter als eine Form von Variablen sind (`caesar=function(text, key)`…)

Ob Lua schneller wird, wenn man die Dinge zusammen in eine Zeile packt? Schwer zu sagen, durch das JIT compiling vermutlich nicht. Aber man packt zusammenhängende Formeln zusammen, wenn sie ein sinnvolles Paket ergeben und wenn nicht viel gewonnen würde dadurch, dass man sie über mehrere Zeilen verteilt und in Einzelschritte.

Diese Entscheidung ist mehr persönlicher Stil, aber bevor man ihn sich angewöhnt, besser vorher einen Performance-Test ausführen! Es gibt nichts schlimmeres als schlechte Angewohnheiten, die harmlos aussehen und dann fatale Folgen haben. Bei der Sprache Python zum Beispiel macht es einen gewaltigen Unterschied, bei nicht JIT Lua ebenfalls und bei den meisten bekannten Hochsprachen, die nicht JIT kompilieren.

Der `table.concat` läßt sich im übrigen in der Lua Sprach-Referenz 5.2 nachschlagen. Ein Studium der durchaus übersichtlichen Standard-Bibliotheken ist in jedem Falle dringend anzuraten, wenn man selbst mehr mit Lua machen möchte.

Es gibt eine weitere Methode, die man in der „caesar_fast“ Lösung auf Rosettacode finden kann. Aber Programmen von Leuten, deren Tiefgang man nicht kennt sollte man grundsätzlich mit Skepsis begegnen. In diesem Falle ist die vorgestellte Lösung fatal. Die sehr elegante Lösung dort lautet:

```
local res_t = { text:byte(1,-1) }
```

Und tatsächlich funktioniert diese Methode ausgezeichnet und ohne, dass wir eine for-Schleife benutzen. Wir kopieren aus unserem Text alle Zeichen vom ersten (1) bis zum letzten (-1) als Bytes in eine neue Table „{…}“ und das war‘s! Großartig! Was ist das Problem?

Das Problem dieses Programmierers ist, dass er übersehen hat, dass solche Konstruktionen eine limitierte Menge von Parametern hat. Also ab einer bestimmten Größe von unserem „text“ gibt es eine Fehlermeldung, die wir nur mit Mühe nach vollziehen können.

```
> text=("A"):rep(8000); local res={ text:byte(1,-1) }
> text=("A"):rep(8001); local res={ text:byte(1,-1) }
[string "text=("A"):rep(8001); local res={ text:byte(1..."]:1: string slice too long
  [C]: in function 'byte'
```

Das heißt manchmal ist der geradlinige Weg der bessere und wir können den Lösungen auf irgendwelchen Webseiten im Internet grundsätzlich nicht trauen. Die Methode funktioniert bis zu Textlängen von 8000 Zeichen. Danach versagt sie. Und diese 8000 können auch noch von System zu System und von Architektur zu Architektur unterschiedlich sein.

Das ist gefährlich und solcherlei Art „Bugs“ sind überall in unserer Infrastruktur. In dem Autopiloten vom Airbus bis zur Sicherheit von Webseiten, mit einem grandiosen Zoo von solchen Bugs in Form von Windows 10, was zu einem wimmelnden Heer von Schadware geführt hat und einer absolut begeisterten und fleißigen Ansammlung von Cyberkriminellen, die jeder ihr kleines Ding auf der Plattform drehen.

Also: Finger weg von den allzu coolen Lösungen. Ein bisschen konservativ bleiben beim Programmieren ist gut, wenn es um Sicherheit geht sogar absolutes Muss. In der Kryptologie sind Leute extrem konservativ und nehmen lieber eine Routine, die sich seit jahrzehnten bewährt hat anstatt eine kleine, coole, neue, bessere, elegantere Verschlüsselungsmethode (10x langsamer übrigens), ich sag mal „Elliptic Curve Cryptography“ und „NSA Backdoor“ in einem Satz.

Ich will damit niemanden davon abhalten, auf Rosettacode suchen zu gehen, im Gegenteil! Aber auch namhafte Programmierer bauen Mist. Und jeder Anfänger, wenn er drauf achtet, findet diesen Mist. Beispielsweise schaut doch einfach mal nach „Modulo“ Anwendungen. Nicht jeder Modulo ist schlecht, aber jeder sollte Alarmglocken schrillen lassen und Überprüfung verlangen.

```lua
-- caesar_uebung5.lua
function caesar_modulo(text, key)
    local char=string.char
    local A=("A"):byte() -- A=65
    local cipher={}
    for i=1,#text do
        cipher[i]=char((text:byte(i)-A+key)%26+A)
    end
    return table.concat(cipher)
end
function caesar_if(text, key)
    key=key%26  -- key ist immer positiv
    local char=string.char
    local A=("A"):byte() -- A=65
    local cipher={}
    for i=1,#text do
        local c=(text:byte(i)-A+key)
        if c>25 then c=c-26 end
        cipher[i]=char(c+A)
    end
    return table.concat(cipher)
end

local key=13
local text=("ABCDEFGHIJKLMNOPQRSTUVWXYZ"):rep(1e6)
local clock=os.clock
local start=clock()
local a=caesar_if(text, key)
print("IF Variante:", clock()-start, "sec")
start=clock()
local b=caesar_modulo(text, key)
print("Modulo Variante:", clock()-start, "sec")
```

Die Überprüfung in diesem Falle musste wirklich radikal ausfallen. Wir arbeiten mit 26 Megabyte an Test-Text und Lua knackt diese Mengen in wenigen Sekunden.

```
IF Variante:      3.615892    sec
Modulo Variante:  4.112171    sec
```

Wir sehen trotzdem, wie unsere if-Variante glatte 13% schneller ist, wobei die Modulo-Funktion offenbar im Verhältnis zu dem Rest der Operationen nicht so stark ins Gewicht fällt. Es gibt jedoch Situationen, in denen die Unterschiede deutlich gravierender ausfallen. Der eine Modulo am Beginn der if-Variante spielt im übrigen keine Rolle, da es nur einmal aufgerufen wird. Zu den restlichen 26 Millionen spielt das keine Rolle.

Stellen wir fest: unsere Vermutung war also richtig! If ist schneller als Modulo, wenn aber die übrigen Operationen so viel Zeit fressen, dass die Mehrkosten von Modulo keine Rolle spielen, können wir das auch gern verwenden. Was für Modulo gilt, gilt natürlich für alle anderen Funktionen und Verbesserungen unserer Routine.

Wir haben bisher, zum Beispiel, nicht überprüft, ob unser table.concat wirklich schneller als ein „..“ ist. Probiert das aber gern selbst. Aber mein Rat ist der, sich dabei nicht nur einen Tee zu kochen, sondern nochmal los zu gehen, um sich einen neuen zu kaufen, ihn aufzubrühen, zu trinken, in das Museum für Völkerkunde zu fahren, dort an einer Teezeromonie teil zu nehmen zurück zu kommen und dann vielleicht… vielleicht ist er fertig.

Das Problem mit der String-Variante ist nämlich, dass sie in quadratischer Laufzeit arbeitet. Während sie bei 26tausend Zeichen nur 10 mal langsamer läuft, ist es bei der nächsten 10er Potenz 100 mal langsamer, 10000 mal langsamer, usw. Mein Versuch mit 1e5 Alphabeten läuft seit einer Viertelstunde und ich warte nicht mehr ab[1], wann er denn fertig wird… Vielleicht ist die Laufzeit auch exponential? Ihr könnt das selbst entscheiden, ob es sinnvoll ist, `table.concat` statt dessen zu verwenden.

1 Nur 54 Minuten dann doch, also 8600 mal langsamer. Für kleine Strings wirklich okay, aber… das sind so die kleinen Dinge, die gemeint sind mit „als sich die Schulweisheit erträumt“.

Meistens liegt es nicht an der Sprache, wenn ein Programmierer Probleme mit der Geschwindigkeit seiner Programme bekommt, sondern mit seiner Art diese anzuwenden. Die richtigen Algorithmen sind jedenfalls ein wesentlicher Faktor, oftmals wichtiger als eine starke CPU oder eine „bessere“ Sprache. Das alles erlernt man nur aus Erfahrung, also machen, selbst Hand anlegen und probieren. Drüber lesen reicht nicht, kritisches, wissenschaftliches Vorgehen muss man praktizieren und nicht nur studieren.

5.1.6 Cäsar mit Tabellen: Substitution

Wir haben bereits festgestellt, dass unsere Implementation mit Addition, so „wie man es macht“ und es an der Schule und Universitäten gelehrt wird, starke Grenzen gesetzt sind. Wir können nicht ohne weiteres kleinere Alphabete benutzen, wir können die Alphabete nicht durchwürfeln oder rückwärts sortieren, wir können sie nicht austauschen und mit Unicode-Zeichen können wir schon gar nicht arbeiten.

Vielleicht ist die Version über Tabellen nicht ganz so schnell wie die Berechnung aber wir werden sehen, wieviel uns das kostet. Lua, wie im Teil über die tables schon besprochen, besitzt die sehr mächtige Hash-Table Funktionalität. Das heißt Lua kann assoziieren. So gesehen ist eine Folgerung von „A“→ „N“ eine Assoziation.

Wie sähe eine solche Tabelle aus? Wir müssten sie nämlich aus unseren Chiffrier-Alphabeten generieren und richtig nett wäre es, wenn wir statt der ASCII Zeichen einfach Unicode lesen könnten aus unserem String. Dazu haben wir unter Utilities schon eine kleine Routine, auf die wir natürlich zurück greifen werden.

Die Hash-Tabelle bauen wir erstmal händisch, man nennt diese Form der Buchstabenaustauschung auch „Substitution“. In der ersten Variante erledigen wir das so wie wir es kennen, in der zweiten greifen wir auf die string-Funktion „gsub“ zurück, was für „globale Substitution“ steht. Dafür geht dieses globale Suchen und Ersetzen durch den String und ersetzt jedes Zeichen gegen das in der Tabelle, dem dritten Argument.

```lua
-- caesar_uebung7.lua
function caesar_table(text, key)
    cipher={}
    for i=1, #text do
        cipher[i]=key[text:sub(i,i)]
    end
    return table.concat(cipher)
end

function caesar_gsub(text, key)
    return string.gsub(text, ".", key)
end

function caesar_gsub_unicode(text, key)
    return string.gsub(text, "([%z\1-\127\194-\244][\128-\191]*)", key)
end

local tab={
    A="N", B="O", C="P", D="Q", E="R", F="S",
    G="T", H="U", I="V", J="W", K="X", L="Y", M="Z",
    N="A", O="B", P="C", Q="D", R="E", S="F",
    T="G", U="H", V="I", W="J", X="K", Y="L", Z="M"}
local text=("ABCDEFGHIJKLMNOPQRSTUVWXYZ"):rep(1e6)
local clock=os.clock
local start=clock()
local a=caesar_table(text, tab)
print("       Table Variante:", clock()-start, "sec")
print(a:sub(1,52))

clock=os.clock
start=clock()
local b=caesar_gsub(text, tab)
print("        Gsub Variante:", clock()-start, "sec")
print(b:sub(1,52))

clock=os.clock
start=clock()
local c=caesar_gsub_unicode(text, tab)
print("Unicode-Gsub Variante:", clock()-start, "sec")
print(c:sub(1,52))
```

Das Ergebnis dieses Tests ist folgendes:

```
        Table Variante:  1.324761   sec
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
         Gsub Variante:  1.714772   sec
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
Unicode-Gsub Variante:  3.30194    sec
NOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLM
```

Wir stellen also fest, dass unsere händische Variante die schnellste ist, sie ist sogar schneller als unsere schnellste berechnete „IF“ Variante, die dauerte 3.6 Sekunden, wir sind also tatsächlich fast dreimal so schnell!

Die Hash-Table von Lua ist, wie sich bestätigt, ein sehr mächtiges Werkzeug. Es wird dabei auf eine sehr effiziente Weise eine komplexe Funktion ausgeführt, das sehr großes Potential hat. Und nebenbei hat das unsere ersten sechs Testroutinen damit deklassiert.

Die `string.gsub` Varianten sind beide etwas langsamer und während sich das für das Lesen eines einzelnen Zeichens, mittels „.“, kaum lohnt, bekommen wir aber mit der Fähigkeit Unicode zu lesen ein mächtiges Werkzeug in die Hand, das diesen Preis wohl wert ist. Nebenbei können wir nun übrigens Leerzeichen, Satzzeichen und anderes verwenden, da alles, was nicht in der Tabelle des Zeichenvorrat-Alphabets enthalten ist einfach im Original erhalten bleibt.

Wie das Pattern „([%z\1-\127\194-\244][\128-\191]*)“ genau funktioniert, kann man in der 5.2 Referenz zum Thema Pattern nachschlagen. Wie das Pattern für Unicode zustande kommt, das wir hier verwenden, dafür müsste man untersuchen, wie UTF-8 genau aufgebaut ist und wie es funktioniert, dann wird es klar wie dieses Pattern arbeitet. Wir gehen da erstmal nicht näher drauf ein und benutzen es einfach.

Wir haben hiermit übrigens die (fehlerhaften) Referenz-Implementationen auf Rosettacode geschlagen. Soviel zu großartigen Beispielen aus dem Netz und wie sehr wir ihnen vertrauen sollten.

Fassen wir zusammen: wir haben eine sehr schnelle, sehr flexible Lösung für unser Substitutions-Problem. Wir können mit reduzierten Alphabeten arbeiten und unsere Tabellen würden auch Wort-Ersetzungen durchführen, wenn wir die Pattern dafür anpassen würden.

Was wir noch brauchen, wäre eine automatische Methode, mit der wir diese sehr unhandlich zu schreibenden Tabellen automatisch erstellen können, außerdem brauchen wir noch einen Verschiebe-Key. Also eine Zahl, mit der wir das Cipher-Alphabet gegen das Zeichenvorrats-Alphabet verschieben können.

Gehen wir am besten Schritt für Schritt vor, um das zu erreichen. Programmierung ist der Prozess der schrittweisen Verbesserung und nicht des einmaligen Geniestreiches. Ich hoffe, dass der Prozess und wie wir mental an das Problem heran gehen verständlich ist.

5.1.7 Cäsar Tabellen bilden

Das Problem unsere Substitution mit Unicode zu machen ist deutlich komplexer als es den Anschein hat. Daher greifen wir jetzt auf eine Bibliothek zu. In der Programmierung muss man nicht alles selbst können, es gibt immer jemanden draußen in der Welt, der weiter ist, erfahrener im Programmieren und der uns vielleicht diesen Kode zur Verfügung stellt, wenn wir ein wenig danach suchen.

Wir kümmern uns, wenn wir eine Bibiothek benutzen nur darum, wie wir die Funktionen benutzen, nicht wie sie im Detail implementiert wurden. Im Prinzip haben wir alles gelernt, was dafür notwendig ist, die Details der Implementation gehen aber über den Rahmen dieses kleinen Kurses, was niemanden davon abhalten soll, bei der Bibliothek die Haube zu lüften und zu sehen, wie der Motor funktioniert.

Ich habe die Bibliothek kurzerhand „ccrypt.lua“ genannt, also „Classic Crypt“. Diese Bibliothek wird im Laufe des Kurses weiter wachsen, aber wir haben schon mal ein paar schöne Funktionen. Der Einfachheit klappe ich die Funktionen im Editor zusammen, so dass man nur noch die Namen sieht. Dann folgen Bespiele, wie wir damit arbeiten und wir fangen an, damit ein bisschen Spaß zu haben.

```
1     -- ccrypt.lua
2     local Unicode="([%z\1-\127\194-\244][\128-\191]*)"
3   ⊞ function string.utf8all(text)
7   ⊞ function string.utf8len(str)
11  ⊞ function string.filter(input, pat, rep)
17  ⊞ function string.block(text, block, line)
34  ⊞ function string.shuffle(text)
47  ⊞ function string.subst_table(alphabet, cipher, key)
67  ⊞ function string.substitute(text, sub_table, pattern)
```

Wir haben unsere Substitutions-Funktion „substitute“, die auf einen String angewendet werden kann mit einer Substitutions-Tabelle ähnlich wie den, den wir gelernt haben. Die Sache mit dem Unicode macht das etwas komplizierter, aber das läuft alles unter der Haube und braucht uns nicht zu kümmern.

Die Substitutions-Tabellen erzeugen wir mit subst_table, wobei wir sie über ein beliebiges Alphabet erzeugen, mit einem „Additions-Key“ als key-Argument und einem möglicherweise anderem Ziel-Alphabet „cipher“.

Mit shuffle können wir ein Alphabet vermischen, das ist nützlich, wenn wir eben nicht wie bei Cäsar auf die selbe Reihenfolge sondern eine zufällige Reihenfolge substituieren wollen, etwa „ABCD…“ → „DABC…“ oder „ABCD…“ → „🙀🙁🙂🙅“

filter und block sind Hilferoutinen, mit denen wir die Ausgabe netter gestalten können und utf8all sowie utf8len sind Hilfsroutinen, mit denen wir durch eine Menge Unicode-Zeichen blättern können oder die tatsächliche Anzahl der Unicode-Zeichen bekommen. Wir wissen ja, dass bei ASCII jedes Zeichen ein Byte ist, aber bei Unicode kann ein Zeichen aus bis zu vier Bytes bestehen, das „Ä“ beispielsweise besteht aus zwei Bytes.

Fangen wir mit etwas einfachen an, ein klassischer Cäsar, diesmal aber mit Leerzeichen und Satzzeichen, die unsere neuen Routinen nämlich unangetastet lassen, wenn wir sie nicht heraus filtern. Die neuen Programme beginnen alle mit „mono“, da es sich um monographische monoalphabetische Schiffrierungen handeln, also ein Zeichen wird in ein anderes getauscht, ohne dabei das Alphabet zu wechseln.

5.1.8 Klassisch Cäsar

```lua
-- mono_caesar.lua
require "ccrypt"

local text=[[Er stand auf seines Daches Zinnen,
Er schaute mit vergnügten Sinnen
Auf das beherrschte Samos hin.
«Dies alles ist mir untertänig»,
Begann er zu Ägyptens König,
«Gestehe, dass ich glücklich bin.»
]]
local alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜß"
local key=13
-- wir wandeln erstmal unseren Text in Großbuchstaben
text=text:upper()
--[[ aber die Sonderzeichen wurden noch nicht gewandelt,
wir wandeln die kleinen Zeichen zu großen also
nur wandeln, nicht verschieben]]
toupper_tab=("äöü"):subst_table("ÄÖÜ")
text=text:substitute(toupper_tab)
print(text) -- Originaltext
print(alphabet, "Key="..key, "\n")
--[[ jetzt verschlüsseln wir ihn mit klassischem Cäsar
wenn wir kein cipher-alphabet angeben, nimmt er das
]]
local enc_key=alphabet:subst_table(alphabet, key)
local encrypted=text:substitute(enc_key)
print(encrypted) -- verschlüsselter Text
--[[ zum Entschlüsseln kehren wir den key um und tauschen
das Alphabet gegen das Cipheralphabet aus (in diesem Falle ist
es das selbe]]
local dec_key=alphabet:subst_table(alphabet, -key)
local decrypted=encrypted:substitute(dec_key)
print(decrypted) -- entschlüsselter Text
```

Die Ausgabe ist diese:

```
ER STAND AUF SEINES DACHES ZINNEN,
ER SCHAUTE MIT VERGNÜGTEN SINNEN
AUF DAS BEHERRSCHTE SAMOS HIN.
«DIES ALLES IST MIR UNTERTÄNIG»,
BEGANN ER ZU ÄGYPTENS KÖNIG,
«GESTEHE, DASS ICH GLÜCKLICH BIN.»

ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜß Key=13

RA BCNÄQ NDS BRVÄRB QNPURB IVÄÄRÄ,
RA BPUNDCR ZVC ERATÄLTCRÄ BVÄÄRÄ
NDS QNB ORURAABPUCR BNZÖB UVÄ.
«QVRB NYYRB VBC ZVA DÄCRACJÄVT»,
ORTNÄÄ RA ID JTHÜCRÄB XKÄVT,
«TRBCRUR, QNBB VPU TYLPXYVPU OVÄ.»

ER STAND AUF SEINES DACHES ZINNEN,
ER SCHAUTE MIT VERGNÜGTEN SINNEN
AUF DAS BEHERRSCHTE SAMOS HIN.
«DIES ALLES IST MIR UNTERTÄNIG»,
BEGANN ER ZU ÄGYPTENS KÖNIG,
«GESTEHE, DASS ICH GLÜCKLICH BIN.»
```

Das ist nett, aber nicht alles, was wir mit unserer neuen Substitution tun können. Wir hatten von der Reduktion des Alphabets gesprochen, auf 20 Zeichen?

5.1.9 Reduktion des Alphabets

```
-- mono_20letter.lua
require "ccrypt"
local text=[[ER STAND AUF SEINES DACHES ZINNEN,
local alphabet="ABCDEFGHIJKLMNOPQRSTUVYZ"
local   cipher="ABCDEFGHIICLMNOPQRSTVVIZ"
local enc_key=alphabet:subst_table(cipher) -- default key=0
enc_key.X="CS" enc_key.ß="SZ" enc_key.W="VV"
enc_key.Ä="AE" enc_key.Ö="OE" enc_key.Ü="UE"
local encrypted=text:substitute(enc_key)
print(encrypted)
```

```
ER STAND AVF SEINES DACHES ZINNEN,
ER SCHAVTE MIT VERGNUEGTEN SINNEN
AVF DAS BEHERRSCHTE SAMOS HIN.
«DIES ALLES IST MIR VNTERTAENIG»,
BEGANN ER ZV AEGIPTENS COENIG,
«GESTEHE, DASS ICH GLUECCLICH BIN.»
```

Wir können also unsere Substitutions-Tabelle auch mit eigenen Erweiterungen versorgen, nachdem wir sie erstmal grundsätzlich erstellt haben. Die automatische Tabelle kennt nur eine ein Buchstabe zu einem Zuordnung, aber der Substitution ist das egal.

5.1.10 Unicode: beliebige Kodierer

Wir können natürlich auch von einem Alphabet in ein beliebiges anderes wandeln, etwa Runen oder Hiroglyphen oder Emoji-Zeichen.

```
-- mono_runes.lua
require "ccrypt"
local text=[[ER STAND AUF SEINES DACHES ZINNEN,
ER SCHAUTE MIT VERGNÜGTEN SINNEN]]
local alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜß.,"
local    runes="ᚨᛒᚲᛞᛖᚠᚷᚺᛇᛃᛕᛚᛗᛜᛟᛈᛩᚱᛊᚦᚢᚡᚹᚣᛉᚤᛡᚼᛒ·+"
local enc_key=alphabet:subst_table(runes) -- default key=0
local encrypted=text:substitute(enc_key)
print("\n"..encrypted)
local dec_key=runes:subst_table(alphabet)
print(encrypted:substitute(dec_key))
```

```
ᛖᚱ ᛊᚦᚨᛜᛞ ᚨᚢᚠ ᛊᛖᛇᛜᛖᛊ ᛞᚨᚲᚺᛖᛊ ᛉᛇᛜᛜᛖᛜ+
ᛖᚱ ᛊᚲᚺᚨᚢᚦᛖ ᛗᛇᚦ ᚡᛖᚱᚷᛜᚤᚷᚦᛖᛜ ᛊᛇᛜᛜᛖᛜ
ER STAND AUF SEINES DACHES ZINNEN,
ER SCHAUTE MIT VERGNÜGTEN SINNEN
```

Gehen wir das selbe allgemein an mit einigen weiteren Beispielen. Die Liste ist beliebig erweiterbar und veränderbar, dafür muss man sich nur die Unicode-Tabelle vornehmen und interessante Zeichen heraus suchen. Leider ändert das nichts an der Sicherheit unseres Kodes.

```lua
require "ccrypt"
local alpha="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ÄÖÜä
local cards="[illegible]
local emoji="[illegible]
local hieroglyph="[illegible]
local alchem="[illegible]

local len=alpha:utf8len()
cards=cards:shuffle()
emoji=emoji:shuffle()
hieroglyph=hieroglyph:shuffle()
alchem=alchem:shuffle()

-- Schneide die Hieroglyphen und Alchemie auf die Länge des Alphabets
hieroglyph=hieroglyph:match("("..ccrypt.Unicode:rep(len).."）")
alchem=alchem:match("("..ccrypt.Unicode:rep(len)..")")

text=[[Durch diese hohle Gasse muss er kommen. Es führt kein andrer We
Hinterhalt diesen Abend. Wird später. Mit Abendbrot nicht auf mich war
- Wilhelm Tell]]
print(text)
text=text:filter()
local enc=alpha:subst_table(cards)
local cipher=text:substitute(enc)
print(cipher:block(5, 40), "\n")

enc=alpha:subst_table(emoji)
cipher=text:substitute(enc)
print(cipher:block(5, 40), "\n")

enc=alpha:subst_table(hieroglyph)
cipher=text:substitute(enc)
print(cipher:block(5, 40), "\n")

enc=alpha:subst_table(alchem)
cipher=text:substitute(enc)
print(cipher:block(5, 40), "\n")
-- decipher
enc=alchem:subst_table(alpha)
cipher=cipher:substitute(enc)
print(cipher:block(5, 40), "\n")
```

```
Durch diese hohle Gasse muss er kommen. Es führt kein andrer Weg nach Küssnacht.
Hinterhalt diesen Abend. Wird später. Mit Abendbrot nicht auf mich warten, Mutter.
- Wilhelm Tell
```

```
Durch diese hohle Gasse musse rkomm enEsf
ührtk einan drerW egnac hKüss nacht Hinte
rhalt diese nAben dWird späte rMitA bendb
rotni chtau fmich warte nMutt erWil helmT
ell
```

Wie wir sehen, ist es kein großes Problem einen beliebigen Kode zu erzeugen, beliebige Symbole zu zu weisen. Selbstverständlich gehen Morsezeichen oder irgend etwas anderes, eine Substitution erzeugt genau genommen nur ein anderes Symbol für den verwendeten Buchstaben. Wir kommen gleich drauf zurück, wenn wir den Cäsar Chiffre knacken gehen.

Vorher noch eine kurze Überlegung: Unicode kann nicht nur mit Zeichen drucken, es gibt auch eine ganze Anzahl Zeichen, mit denen man andere modifzieren kann. Das werden wir nochmal probieren, wie das aussieht. Es ändert prinzipiell nichts an dem Prozess, aber wenn man versteht, wie die monoalphabetische Substitution funktioniert, sind sie am Ende dann alle gleich.

5.1.11 Modifikation von Unicode Zeichen

```
-- mono_modify.lua
require "ccrypt"

local alpha="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
local CARONBELOW="\xCC\xAC" -- n̬
local RINGABOVE="\xCC\x87" -- ṅ
local X="\xCD\x93" -- n͓
local LOWLINE="\xCC\xB1" -- ṉ
local APICAL="\xCC\xBA" -- n̺
local cipher=CARONBELOW.."bcd"..RINGABOVE.."fgh"..
    X.."jklmn"..LOWLINE.."pqrst"..APICAL.."vwxyz"

text=[[Durch diese hohle Gasse muss er kommen.
    Es fuehrt kein andrer Weg nach Kuessnacht.]]
key=0
text=text:upper():filter()
local enc=alpha:subst_table(cipher, key)
encrypt=text:substitute(enc)
print(encrypt, "\n")
```

```
d̺rchḋ͓ṡẖhl̇g̬sṡm̺sṡrḵmṁṅsḟ̺hrtk͓̇n̬nḋṙẇgn̬chk̺̇ssn̬cht
```

6 Zahlensysteme

Als kleine Zwischenübung nehmen wir uns kurz einmal vor, wie die typischen Zahlen in der Informatik geschrieben werden. Nicht Dezimal, sondern Binär, Oktal und vor allem Hexadezimal. Das heißt während im Dezimalsystem zehn Ziffern zur Verfügung stehen, nämlich 0 bis 9, sind es im Binären- oder Dualsystem zwei, 0 und 1 und im Oktalen natürlich acht, also 0 bis 7.

Dezimal	Binär 0b	Hexadezimal 0x	Oktal 0o
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Wie wir sehen drückt eine Hexadezimalzahl genau vier Bit aus, also ein halbes Byte, was auch als „Nibble“ bezeichnet wird, ein „Knabbern“ als ein kleiner Biss. Mit zwei Hexadezimalzahlen lassen sich also 8 Bit schreiben. Die Oktalzahlen finden ebenfalls Verwendung, da sie immer drei Bit zu einer Ziffer zusammen fassen, aber die Zeiten, als unsere Computer noch mit 6 Bit Zeichen statt 8 arbeiteten sind vorbei und so kommt das Oktalsystem etwas aus der Mode.

Wir haben uns für das Hexadezimalsystem natürlich die Buchstaben A-F ausgeliehen für die fehlenden Ziffern (10), (11), (12), (13), (14) und (15) und das ist natürlich sinnvoll. Ansonsten müssten wir statt 0xAA immer 0x(10)(10) schreiben, denn 0x1010 wäre eine vierstellige Zahl, also 4112.

6.1 Wie rechnen wir die Zahlen in ein anderes System um?

Ganz einfach. Die Stellen jedes Zahlensystems sind die Basis in der Potenz. Zum Beispiel für das 10er System ist die erste Stelle die 1er, also 10^0. Die zweite Stelle sind die 10er, also 10^1. Die dritte 10^2, 10^3 und so weiter. 10^2 sind natürlich 10x10 also 100. 1er, 10er, 100er und so weiter.

Mit den anderen Zahlensystemen ist es genauso. Binär sind die Stellen 1, 2, 4, 8, 16, 32, 64,128,256,512,1024,2048, 4096. Für das Hexadezimalsystem 1, 16, 256, 4096,…

Man schreibt in Lua ein „0x“ vor eine Zahl, um auszudrücken, dass es sich um eine Hexadezimalzahl oder einfacher eine Hexzahl handelt. 0xAA ist also umgerechnet 10+16*10=170. Hexadezimal 0x1000 ist dezimal 4096, 0xC000 wären dann 12*4096 und so weiter. Wir können das auch in Lua machen mit der `string.format()`-Funktion.

Ebenfalls bietet die bit.* Library einige Funktionen, um mit Hex- und Binärzahlen um zu gehen, hier die zahlensysteme_uebung1.lua:

```
1   --Zahlensysteme in Lua
2   a=0xaa
3   b=192
4   c=11
5   print(a, b, c)
6   -- wir können die Ausgabe formatieren
7   print(string.format("%02X %02X %02X", a, b, c))
8   print(string.format("%x %x %x", a, b, c))
```

Ausgabe:

```
170 192 11
AA C0 0B
aa c0 b
```

Wie wir sehen, können wir die Stellen kontrollieren oder ob eine führende Null gedruckt wird. Mehr dazu im Lua-Buch oder auf der eben verlinkten Seite zu `string.format`.

Die Aus- und Eingabe geht auch über die bit-Library mittels **`print(bit.tohex(123456))`** zum Beispiel. Aus- und Eingabe als Binärformat geht mit dem von uns verwendeten LuaJIT 2.0.4 leider nicht, aber die Nachfolgeversion LuaJIT 2.1.0beta3, die man sich auch von der LuaJIT Seite herunter laden kann, bietet solche Funktionen an (ab beta1).

Hier geht sowas wie **`a=0b101101`** zum Beispiel, um eine Zahl binär einzugeben. Aber wir können Zahlen mit **`a=0o1234567`** Oktal eingeben und mit **`print(format.string('%o', a))`** ausgeben. Das 0o, also Null-kleines ‚o‘ leitet hier die Oktalzahl ein.

6.2 Logische Operationen

Wir schieben an dieser Stelle auch noch schnell die logischen Operationen ein, wir hatten schon einmal die XOR oder EOR, also das Exklusiv-Oder, das sich in Lua berechnet mit:

```
function xor(a, b)
      return a or b and not (a and b)
end
```

Das letzte (a and b) haben wir in Klammern gesetzt, um sicher zu stellen, dass das „not“ sich auf den gesamten Term und nicht nur auf das a bezieht. **`not false and false`** ergibt nämlich **`false`**, während **`not (false and false)`** dann **`true`** ergibt. Probiert das gern selbst einmal aus. Die Klammern sind wichtig!

Die logischen Funktionen AND und OR kann man in einer Tabelle mit zwei Eingangsvariablen a und b so darstellen:

A	B	A and B	A or B	Not A	Not B	xor(A, B)
False	False	False	False	True	True	False
True	False	False	True	False	True	True
False	True	False	True	True	False	True
True	True	True	True	False	False	False

Die XOR Funktion ist hier sehr interessant. Wir könnten mit ihrer Hilfe eine Verschlüsselung vornehmen (Vernam-Chiffre).

Verschlüsseln wir mit Hilfe der xor Funktion eine Folge von Wahrheitswerten A=11001 mit einem Code B=10101, dann bekommen wir 01100. Den Code können wir danach wieder mit B entschlüsseln, indem wir das XOR wiederholen:

A	11001
B	10101
A xor B = C	01100
C xor B = A	11001

An dieser Stelle hilft uns Lua wieder mit der bit-Library.

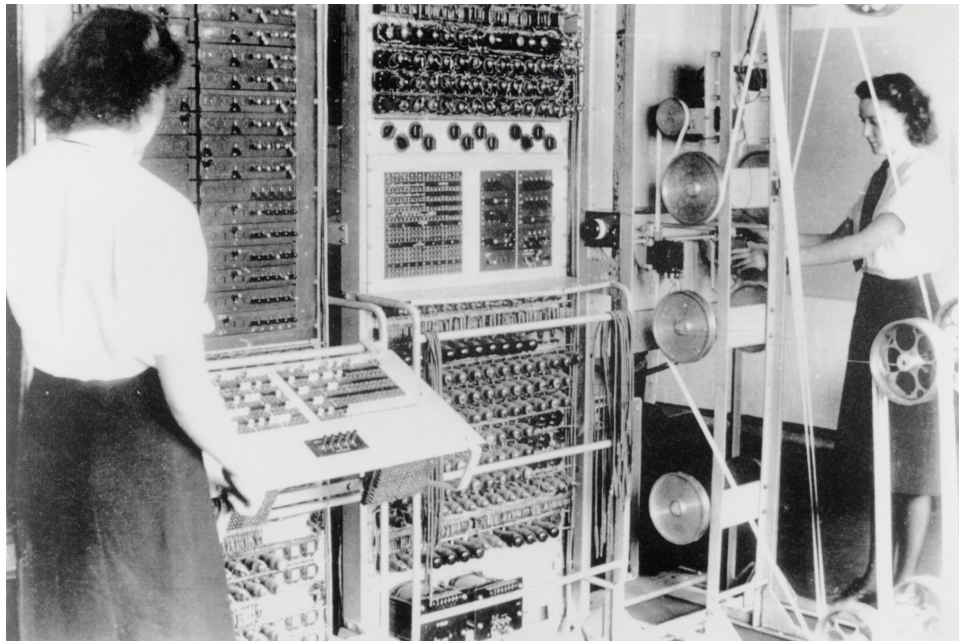
```
1   -- zahlensysteme_uebung2.lua
2   a="W" -- Klartext
3   ax=string.byte(a)-65
4   print(string.format("*%s = %02x", a, ax))
5   b="C" -- Schlüssel
6   bx=string.byte(b)-65
7   print(string.format(" %s = %02x", b, bx))
8   cx=bit.bxor(ax,bx) -- verschlüsseln
9   print(string.format(" %s = %02x", string.char(cx+65), cx))
10  dx=bit.bxor(cx,bx) -- entschlüsseln
11  print(string.format("*%s = %02x", string.char(dx+65), dx))
```

Ausgabe:

```
*W = 16
 C = 02
 U = 14
*W = 16
```

Wie wir sehen wird aus dem W ein U und wieder ein W. Alle modernen Krypto-Algorithmen arbeiten mit XOR Funktion, der einzige Unterschied der Algorithmen ist, wie sie den Cipher-Code erzeugen. Das Elegante an dieser Methode ist, dass man das hin und herverschlüsseln mit einer einzigen Operation vollziehen kann. Die Addition und Subtraktion in diesem Falle ist an sich völlig überflüssig, da der chiffrierte Text ohnehin unlesbar sein soll und nur aus Bytes besteht.

Eines der bekanntesten Maschinen, die diese Methode benutzt hat, ist die Lorenz-Schlüsselmaschine SZ42, von der wir eine Abbildung auf der Frontseite dieses Buches finden; ihr Cipher-Code war schwach, so konnte die Verschlüsselung in Bletchley Park, in der Nähe von London, von den Kryptologen dort mit Hilfe des Colosssus geknackt werden.



Es war jedoch nicht die Operation XOR schwach, sondern die Art, wie der Chiffrierschlüssel erzeugt wurde. Die Mechanischen Geräte dieser Zeit waren bestimmten Grenzen unterworfen. Während der Colossus für eine solche Nachricht noch acht Stunden brauchte, schafft es ein heutiger Computer in einer halben Minute, vermutlich sogar in Bruchteilen von Sekunden, wenn man sich etwas Mühe gibt, praktisch in Echtzeit.

Doch wie auch wir Schlüssel knacken können, wollen wir im Folgenden untersuchen.

7 Kryptoanalyse

7.1 Cäsar Chiffre

Wie brechen wir diesen Code? Natürlich. Wir können einfach alle 26 möglichen Cäsar-Einstellungen ausprobieren, sie untereinander ausdrucken mit dem Computer und unser Gehirn sieht auf den ersten Blick, welche Zeile die richtige ist.

Warum können wir das? Und wie können wir Cäsar knacken, wenn wir auf ein symbolisches Alphabet abbilden oder ein zufällig vermischtes?

Und hier kommen wir auf den spannenden Punkt, auf den wir schon am Anfang des Buches gestoßen sind: „Kode ist überall“. Buchstäblich. Und im folgenden Abschnitt nehmen wir uns das Skalpell unseres Verstandes und unsere Bildung (heißt Wikipedia), um dieses „überall“ zu zeigen. Und warum das Verständnis des nächsten Schrittes nicht nur wichtig ist für ein bisschen Wortspielerei, das einfache Entschlüsseln einer Nachricht, sondern warum wir mit dieser Methode im Stande sind viele, wenn nicht gar alle Probleme der Menschheit zu knacken. Denn alles ist Kode. Alles ist verschlüsselt auf eine gewisse Weise und wartet nur darauf, dass jemand daher kommt und es entschlüsselt. Was ist $E=mc^2$? Die entschlüsselte Formel der Energie. Haben wir den Schlüssel, steht uns die Welt offen.

Aber genug davon. Fangen wir mal ganz harmlos an. Wir nehmen einen beliebigen Text, zum Beispiel einen beliebigen Artikel aus Wikipedia und zählen, wie viele Buchstaben es gibt und wie viele Buchstabenpaare. Buchstabenpaare können wir auch weiter auffassen, als Gruppen von zwei, drei, vier oder mehr Buchstaben, wir nennen diese Gruppen üblicherweise Buchstaben-Tupel.

Und dann stellen wir eine überraschende Sache fest: sie sind nicht gleich verteilt. Also einige Buchstaben sind sehr viel häufiger vorhanden als andere. Aber wir behaupten das nicht einfach, wir machen es statt dessen. Wir sind Wissenschaftler, wir stützen unsere Behauptungen auf Zahlen, Beweise.

Wir haben dazu in unserer Bibliothek ein paar weitere Funktionen, die wichtigste lautet `string.count_tuples()`, wobei wir unseren Text und die Anzahl Buchstaben angeben und dafür eine Tabelle zurück bekommen mit der Anzahl der gezählten Tupel. Im String „ABCDA“ wäre dann A=2 und BC und D jeweils=1, ein Bigramm, also ein 2er Tupel AB, eins BC, CD DA käme ebenfalls vor, ein Trigramm, also ein 3er Tupel wäre in diesem Falle ABC, BCD und CDA und so weiter.

8 Literaturliste

Friedrich L. Bauer: *Entzifferte Geheimnisse. Methoden und Maximen der Kryptologie.* 3., überarbeitete und erweiterte Auflage. Springer, Berlin u. a. 2000, ISBN 3-540-67931-6.

Albrecht Beutelspacher *Geheimsprachen. Geschichte und Techniken*, 5. aktualisierte Auflage, 2012 ISBN 978 3-406-49046-0