Neue Modder braucht das Land

Workshop – Einführung in LUA

Manuel Leithner aka Face
SFM-Modding
Kontakt: sfm_f@web.de
Homepage: http://forum.landwirtschafts-simulator.de

Gliederung

Gliederung

Gliederung

Grundlagen

- Scriptsprache:

  *„Scriptsprachen sind **Programmiersprachen**, die vor allem für **kleine, überschaubare** Programmieraufgaben gedacht sind. Sie **verzichten** oft auf **bestimmte Sprachelemente**, deren Nutzen erst bei der Bearbeitung größerer Projekte zum Tragen kommen. So wird etwa in Skriptsprachen auf den Deklarationszwang von Variablen verzichtet – vorteilhaft zur schnellen Erstellung von **kleinen Programmen (Prototyping)**, bei großen hingegen von Nachteil, etwa wegen der fehlenden Überprüfungsmöglichkeit von Tippfehlern in Variablennamen.“*

- Merkmale:
  - implizit deklarierte Variablen
  - dynamische Funktionsnamen
  - dynamische Typisierung
  - automatische Speicherverwaltung/-bereinigung
  - werden in der Regel nicht compiliert, sondern interpretiert
    - GIANTS-Engine compiliert die Scripte beim Spielstart

LUA

- Design-Prinzipien:
  - Extensibility
    - Erweiterbarkeit des Programms durch weitere LUA-Script und/oder C-Code
    - Viele Bibliotheken (Libraries) vorhanden
  - Simplicity
    - Kleine und einfache Sprache
    - Einfach zu lernen
  - Efficiency
    - Eine der schnellsten interpretierten Scriptsprachen
  - Portability
    - Lauffähig auf allen Plattformen (Windows, Linux, Unix, Mac, Playstation, etc.)

Was benötigen wir?

- Programmieren (Texteditor):
  - Windows Texteditor
  - Notepad++
- Ausführen (Interpreter):
  - LUA 5.1 Interpreter-Console
  - Web-based Interpreter (http://www.lua.org/cgi-bin/demo)
- Entwicklungsumgebung (besser):
  - LuaEdit
  - SciTE
  - Programm oder Spiel das LUA unterstützt

Getting Started

- Hallo World:
  - Das wohl bekannteste Programm überhaupt
  - Jedes Buch über Programmiersprachen startet mit diesem Programm

```
print(„Hello World“);
```

- Erklärung:
  - *print(…):* Ist eine Funktion, die etwas ausgibt (darüber später mehr)
  - *„Hello World“:* Ist ein String (Zeichenkette)

  - dieser Befehl würde bei Ausführung auf der Console „Hello World“ ausgeben

Konventionen

- LUA-eigene Begriffe:
  - and
  - break
  - do
  - else
  - elseif
  - end
  - false
  - for
  - function
  - if
  - in
  - local
  - nil
  - not
  - or
  - repeat
  - return
  - then
  - true
  - until
  - while

Diese Wörter sind Teil der LUA-Sprache und sind deshalb reserviert. D.h. falls man eine Variable definiert, darf diese **nicht** z.B. „local“ heißen

Kommentare

- Funktion von Kommentaren:
  - Erklärung von Script-Abschnitten
  - Anmerkungen des Autors

Mehrzeiliges Kommentar:

```
_=[[
    print(„Hello World“);
    …
]]
```

Einzeiliges Kommentar:

```
-- print(„Hello World“);
```

- Was sollte kommentiert werden:
  - Eine LUA-Datei sollte min. nennen (Name, Sinn & Zweck, Autor, Datum letzte Änderung)

Gliederung

Datentypen

- Eigenschaften von Datentypen:
  - Geben an von welchem Typ ein Wert ist

- Datentypen in LUA:
  - LUA ist eine **dynamisch typisierte** Sprache
  - Es gibt **keine** Typ-Definitionen im Script selber
  - **ABER**: es gibt 8 sogenannte Basistypen
    - **nil** *(Nicht-Wert: Wert einer Variable vor Initialisierung)*
    - **boolean** *(Wahrheitswert: true / false)*
    - **number** *(Zahl: 1; -1; 1.00; 1e-3; 1e10, etc.)*
    - **string** *(Zeichenkette: „Hallo Welt“)*
    - **function** *(Funktion: print(param))*
    - **table** *(Datenstruktur: Tabelle zum Speichern von Werten)*
    - userdata
    - thread

    } Unwichtig für Modder

```
-- Funktion type(param) gibt den Datentyp einer Variable/Wertes  als String zurück:

print(type("Hello world"))      --> string
print(type(10.4*3))             --> number
print(type(print))              --> function
print(type(type))               --> function
print(type(true))               --> boolean
print(type(nil))                --> nil
print(type(type(X)))            --> string
```

Datentypen

- NIL:
  - Nicht-Wert. Wert einer Variable vor deren Initialisierung
- Boolean:
  - Wahrheitswert
  - true / false
- Number:
  - Zahl: Ganzzahl, Fließkommazahl
  - z.B.: 1; -1; 1.00; 1e-3; 1e10
- String:
  - Zeichenkette: „Hallo ich bin ein Text“ oder ‚Hallo ich bin ein Text‘
  - Escape-Sequenzen:
    - \n : Neue Zeile
    - \t : horizontaler Tab
    - \v : vertikaler Tab
    - \\ : Backslash
    - \“ : Anführungsstriche
    - \‘ : Hochkomma
    - \[oder \[: Öffnende oder schließende eckige Klammer

Gliederung

Operatoren

- Arithmetische Operatoren (einfache Mathematik):
  - Addition: +

```
print( 5 + 5 )    --> 10
```

  - Subtraktion: -

```
print( 5 - 5 )    --> 0
```

  - Multiplikation: *

```
print( 5 * 5 )    --> 25
```

  - Division: /

```
print( 10 / 5 )   --> 2
print( 10 / 0 )   --> 1.#INF        -> ERROR
```

  - Negation: -

```
print( - 5 )      --> -5
```

  - Potenz: ^

```
print( 2^8 )      --> 256
```

  - Modulo:

```
print( 5 % 4 )    --> 1
```

Operatoren

- Relationale Operatoren (Vergleich von Operatoren -> true/false):
  - Kleiner: <

```
print( 5 < 10 )   --> true
Print( 10 < 5 )   --> false
```

  - Größer: >

```
print( 10 > 5 )   --> true
print( 5 < 10 )   --> false
```

  - Kleiner-Gleich: <=

```
print( 5 <= 5 )   --> true
print( 6 <= 5 )   --> false
```

  - Größer-Gleich: >=

```
print( 10 >= 5 )  --> true
print( 5 >= 5 )   --> false
```

  - Gleich: ==

```
print( 5 == 5 )   --> true
print( 6 == 5 )   --> false
```

  - Ungleich: ~=

```
print( 6 ~= 5 )   --> true
print( 5 ~= 5 )   --> false
```

Operatoren

- Relationale Operatoren (Wissenswertes):
  - Gleich(==) und Ungleich(~=):
    - Wenn Werte/Variablen unterschiedlichen Types -> **Ungleich** -> false
    - **Nil** ist nur gleich **mit sich selber**: -> nil == nil -> true
    - Tables & Funktionen sind nur gleich wenn deren Referenzen identisch sind (später mehr)
  - Reihenfolge-Operatoren (<,> ,<= ,>=):
    - **Nur benutzbar bei 2 Nummern oder 2 Strings**
      - Bei 2 Nummern: normales logisches Vorgehen
      - Bei 2 Strings: alphabetische Reihenfolge

Operatoren

- Logische Operatoren (Verknüpfung von True/False):
  - AND: (alle Ausdrücke müssen TRUE sein)

```
print(  true and true )          -> true
print ( true and false )         -> false
print( 5 == 5 and 6 == 6)        -> true
Print( 5 == 5 and 6 < 5)         -> false
```

  - OR: (einer der Ausdrücke muss TRUE sein)

```
print(  true or false)           --> true
print(  false or false)          --> false
print( false or true)            --> true
print( 5 == 5 or 6 == 5)         --> true
print( 6 == 5 or 7 == 10)        --> false
```

  - NOT: (Falls Ausdruck false ist -> true)

```
print(  Not false )              --> true
print( not 5 ~= 5 )              --> true
print( not 5 == 5 )              --> false
```

Operatoren

- Logische Operatoren (Wissenswertes):
  - False und Nil wird als **FALSE** ausgewertet
  - Restliches wird als **TRUE** ausgewertet
  - Klammern von Ausdrucken ist möglich

```
print (  not ( 5 == 5 ) and 4 == 4)                 --> false

print( (( 4 > 2 ) and (10 > 9) ) or not  (5 == 5))  --> true
```

Operatoren

- Verkettung (Konkatenation):
  - Verknüpfung von Strings

```
print ( „Hello „ .. „World“)          --> Hello World

print( 0 .. 1 )                       --> 01

print ( 1 .. „ . LUA-Workshop“)       --> 1 . LUA-Workshop
```

  - Wichtig:
    - Bei der Verkettung von Strings werden komplett neue Strings erzeugt!

Operatoren

- Vorrang bzw. Priorität bzw. Auswertungsreihenfolge:
  - Absteigende Priorität:
    - Potenz: ^
    - Negation: not , -
    - Punktrechnung: * , /
    - Strichrechnung: + , -
    - Verkettung: ..
    - Relationale Op: < , > , <= , >= , ~= , ==
    - Und: and
    - Oder: or

a+i < b/2+1	<-->	(a+i) < ((b/2)+1)
5+x^2*8	<-->	5+((x^2)*8)
a < y and y <= z	<-->	(a < y) and (y <= z)
-x^2	<-->	-(x^2)
x^y^z	<-->	x^(y^z

Gliederung

Variablen

- Was ist eine Variable:

  *„In der Programmierung ist eine **Variable** im allgemeinsten Sinne einfach ein **Behälter** für **Rechnungsgrößen** („Werte“), die im Verlauf eines Rechenprozesses auftreten.“*

- Bestandteile:
  - Bezeichnung
    - Mischung aus Buchstaben (keine Umlaute/Sonderzeichen) und Zahlen und _
    - Name **muss** mit Buchstabe oder _ beginnen!
      - **FALSCH:** **2meinVariable**
    - Schreibweise: eineLangeVariableMitMehrerenWoertern
  - Wert
    - Vor Initialisierung hat Variable den Wert **nil**
  - Initialisierung einer Variable (Zuweisung eines Wertes):

```
x = 2;
a = 1+2;

print( x + a )                          --> 5

name = „SFM-Modding“;
print( name .. „ ist wahnsinnig toll“);  -> SFM-Modding ist wahnsinnig toll
```

Variablen

- Geltungsbereich (Scope):
  „Bereich in dem eine Variable gültig ist bzw. benutzt werden kann“

- Möglichkeiten:
  - Global-Scope
    - überall gültig/benutzbar
  - Local-Scope
    - nur im Block gültig, in dem sie definiert wurde
    - Block (Kontrollstruktur, Funktion -> dazu später mehr)

```
x = 2;          (Globale Variable)
local a = 3;    (Lokale Variable)
```

- Wichtig:
  - Immer Lokale-Variablen, wenn es möglich ist (Ist in den meisten Fällen machbar bei LS)
  - globale Variablen kenzeichnen: g_globaleVariablenName

Konstanten

- Konstanten:

  „Ähnlich wie Variablen, aber ihr Wert ist nach der Initialisierung nicht änderbar“

- Konstanten in LUA:
  - Nicht verfügbar
  - Jedoch möchte man manchmal solche Konstanten definieren
  - Konstante wird als Variable definiert, aber so benannt, dass andere Scripter sehen können, dass diese Variable eine Konstante darstellen soll.

```
ICH_BIN_EINE_KONSTANTE = 5;
```

Gliederung

Kontrollstrukturen

- Was ist eine Kontrollstruktur:

  *„**Kontrollstrukturen** werden in imperativen Programmiersprachen verwendet, um den **Ablauf** eines Computerprogramms **zu steuern**. Eine Kontrollstruktur gehört entweder zur Gruppe der **Verzweigungen** oder der **Schleifen**. Meist wird ihre **Ausführung** über **logische Ausdrücke** der **booleschen Algebra** beeinflusst.“*

- Kontrollstrukturen in LUA:
  - Verzweigungen
    - if … then … end;
    - if … then … else … end;
    - if … then … elseif … (elseif … else …) end;
  - Schleifen:
    - for … do … end;
    - while … do … end;
    - repeat … until …;

Verzweigungen

- Wann benötige ich eine Verzweigung:

  Tipp: „Wenn das gilt dann mache das, sonst mache das“

```
if   boolscher Ausdruck1  then
  _=[[ Kommentar:
     Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
  print(„ich werde ausgegeben wenn ‚boolscher Ausdruck1‘ == true ist“);

elseif   boolscher Ausdruck2    then
  _=[[ Kommentar:
     Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
 print(„ich werde ausgegeben wenn ‚boolscher Ausdruck2‘ == true ist“);
else
  _=[[ Kommentar:
     Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
 print(„ich werde ausgegeben wenn kein vorhergehender Fall true ist“);

end;
```

- Anmerkung:
  - elseif: kann beliebig oft vorkommen
  - else: darf entweder 1 mal oder gar nicht vorkommen
  - boolscher Ausdruck: true/false oder Operatoren (siehe Kapitel Operatoren)

Verzweigungen

- Beispiel:

```
local x = 10;

if   x > 15 then
        print(„x ist groesser als 15“);

elseif  x > 10 then
        print(„x ist groesser als 10“);

else
        print(„x ist kleiner-gleich 10“);          -- --> wird ausgegeben
end;
```

```
local x = 10;

if   x > 5 then
         if x > 7 then
                  print(„x ist groesser als 7“);
        else
                  print(„x ist groesser als 5 aber kleiner-gleich 7“);
        end;
end;
```

Schleifen

- Wann benötige ich eine Schleifen:
  - Im Normalfall um Tables auszulesen
  - Sprich: „Führe den Block aus, solange die Bedingung gilt“

- Arten von Schleifen:
  - Numerische For-Schleife

```
for i=0, 10 do
        print(i);                   -> 0,1,2,3,4,5,6,7,8,9,10
end;

for i=0, 10, 2 do
 _=[[ Kommentar:
      Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
        print(i);                   -> 0,2,4,6,8,10
end;
```

  - Generische For-Schleife:
    - Dazu später mehr ☺

Schleifen

- Arten von Schleifen:
  - While-Schleife:

```
while true do
        print(„ich bin eine Endlosschleife!“);     --> SCHLECHT!
end;
```

```
local x = 0;
while x < 10 do
        print(x);                    --> 0,1,2,3,4,5,6,7,8,9
        x = x + 1;
_=[[ Kommentar:
     Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
end;
```

  - Repeat-Schleife: (Einsetzbar wenn der Block auf jedenfalls 1mal ausgeführt werden soll!)

```
local line;
Repeat
        line = os.read();
until  line ~= „“

print(line);
```

Schleifen

- Schlüsselwort „break“:
  - Wird benutzt um aus einer Schleife frühzeitig herauszuspringen

```
local  x = 0;
while x < 100000 do
        if x == 1000 then
                break;                  --> Schleife wird verlassen sobald x == 1000 ist
        end;
        x = x + 1;
end;
```

Gliederung

Funktionen

- Was sind Funktionen:

  „Bezeichnung eines Programmierkonzeptes. Hauptmerkmal einer Funktion ist es, dass sie ein Resultat zurückliefert“

- Sinn und Zweck von Funktionen:
  - Auslagerung von Programm-Code
  - Wiederverwendung von Programm-Code
  - Verbesserung der Lesbarkeit von Programm-Code

- Rückgabewert (nicht zu verwechseln mit Ausgabewert):
  - Ein Wert
  - Kein Rückgabewert (Prozedur genannt in anderen Programmiersprachen)

- Funktionen in LUA:
  - Funktionen in LUA können einen oder keinen Rückgabewert haben
  - Sind ebenfalls Datentypen (nicht wie in richtigen Programmiersprachen)
  - Funktions-Namen sind dynamisch änderbar

Funktionen

- Definition einer Funktion

```
function funktionsName ( [param1, param2, param3,…] )
  _=[[ Kommentar:
     Das ist ein BLOCK (Siehe Local/Global-Scope/Geltungsbereich)
  ]]
  return WERT;                    --> eine Funktion muss kein Return Anweisung haben
end;
```

```
function multipliziere(x,y,z)

        local wert = x * y * z;     --> Die Lokale Variable wert ist nur innerhalb der Funktion benutzbar
        return wert;
end;

local x = multipliziere(2,5,10);
print(x);                           --> Ausgabe 100
```

```
function gibAus(text)
        print(text);
end;
function sageHallo()
        print(„Hallo“);
End;
gibAus(„Hallo ich bin ein neuer Text“);        --> Hallo ich bin ein neuer Text
sageHallo();                                   --> Hallo
```

Funktionen

- Das Zauberwort „return“:
  - Letzte Statement einer Funktion
  - Beendet eine Funktion
  - Definiert den Rückgabewert
  - Kann auch blank definiert werden

```
function meineFunktion(eineZahl)
        if eineZahl == 5 then
                return;
        elseif eineZahle == 10 then
                return eineZahl;
        else
                return eineZahl * eineZahl;
        end;
end;

print(meineFunktion(5));                --> ?
print(meineFunktion(10));               --> ?
print(meineFunktion(7));                --> ?
meineFunktion(15);                      --> ?
```

Exkurs: Iteratives vs. rekursives Vorgehen

- Iteration:
  - mehrmaliges Ausführen einer Aktion
  - Mitzählen der aktuellen Anzahl der Schleifendurchläufe
  - Sprachkonstrukte: for, while, repeat

  - Vorteile: - schneller
    - weniger Speicherbedarf

  - Nachteile: - unschön
    - teils unübersichtlich

- Rekursion:
  - mehrmaliges Ausführen einer Aktion
  - Kein Mitzählen, sondern „Aktion erklärt sich durch sich selbst“
  - Sprachkonstrukte: if, rekursiver Aufruf

  - Vorteile: - einfacher
    - schöner
    - übersichtlicher

  - Nachteile: - langsamer
    - benötigt mehr Ressourcen (Speicher)

Beispiel: Fakultät

- Iterativ:

```
function fractorial(value)

        if value >= 0 then
                local sum = 1;
                for i=1, value do
                    sum = sum * i;
                end;
                return sum;
        else
                print(„Error: Negative Input“);
                return nil;
        end;
end;

print(meineFunktion(5));                    --> ?
```

Beispiel: Fakultät

- Rekursiv:

```
function fractorial(value)

        if value >= 0 then
                if value ~= 0 then
                    return value * fractorial(value-1);
                else
                    return 1;
                end;
        else
                print(„Error: Negative Input“);
                return nil;
        end;
end;

print(meineFunktion(5));                    --> ?
```

Fragen / Feedback

- Gibt es noch Fragen?
- Was sollte besser gemacht/geändert werden?
- Welche Inhalte würden euch interessieren oder fehlen noch?

Übungen im Anschluss

- Aufgabe 1 (Rekursion / Iteration):
  - Erstelle 2 Funktionen, welche die N-te Fibonacci-Zahl berechnen
  - 1mal rekursiv und 1mal interativ
  - N soll als Parameter übergeben werden.
  - Definition der Fibonacci-Zahlen:
    - Negative Zahlen nicht erlaubt
    - N = 0 -> 0
    - N = 1 -> 1
    - N > 1 -> Fib(n-1) + Fib(n-2)
  - Folge sieht somit so aus:
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1.597, …

Übungen bis zur nächsten Veranstaltung

- Aufgabe 1 (Grundlagen LUA):
  - Erstelle folgende Funktionen (Lösungsweg beliebig wählbar):
  - Funktionen für:
    - Addition, Subtraktion, Multiplikation, Ganzzahl-Division, Modulo
    - In den Funktionen selber darf NUR das Plus-Zeichen benutzt werden.
    - Wenn die Multiplikations-Funktion fertiggestellt ist, darf diese Funktion verwendet werden, sofern sie benötigt wird.
    - Wert 1.Parameter > Wert 2.Parameter

- Aufgabe 2 (Grundlagen LUA):
  - Implementiere folgende Funktion, sodass bei Ausführung Folgendes auf der Console erscheint:

    Abblendlicht: an , Fernlicht: aus
    Abblendlicht: aus, Fernlicht: an
    Abblendlicht: aus, Fernlicht: aus

- Aufgabe 3 (Kommentare):
  - Füge bei beiden Funktionen sinnvolle Kommentare ein!

Übungen bis zur nächsten Veranstaltung

```
function licht()
    --In diesem Block kannst du dein Script unterbringen

    --bis hier her
    print("Abblendlicht:"..abblendlicht.." - Fernlicht:"..fernlicht);
end;

--
abblendlicht = "aus";
fernlicht = "aus";

for i=1,3 do
    licht();
end;
```

- Einsendeschluss: Samstag, 14.März 2010 – 24.00 Uhr
- Nächster Termin: Sonntag, 15.März 2010 – 20.00 Uhr