

# 1 Funktionsumfang und Eingabeverarbeitung auf der Anwendungsebene

## 1.1 Entwicklung einer eigenen Scriptsprache zur Anwendungssteuerung

### 1.1.1 Theoretische Grundlagen formaler Sprachen

Wir als Menschen haben die gesprochenen Sprachen entwickelt, um uns untereinander zu verständigen und Informationen auszutauschen. Um Missverständnisse zu vermeiden, ist unsere Kommunikation durch strikte grammatikalische und semantische Regeln definiert. Es werden strukturelle Merkmale wie zum Beispiel die Reihenfolge einzelner Sprachteile festgelegt, außerdem gibt es einen begrenzten Wortschatz, indem jedes Wort eine bestimmte Bedeutung hat. Um auch Computern Informationen zu übermitteln, und eine maschinelle Verarbeitung dieser zu ermöglichen, benutzt man formale Sprachen. Diese gleichen von den grundlegenden strukturellen Ansprüchen den natürlichen Sprachen, verfügen also auch über eine Syntax und eine Semantik, definieren sich allerdings dadurch, dass diese für einen Rechner eindeutig verständlich sein müssen, und demnach kein Interpretationsfreiraum gelassen wird. Ziel ist es klare Anweisungen in einer für den Computer verständlichen Weise zu übermitteln.

Da Sprachen in der Regel durch das Aneinandereihen von einzelnen Sprachelementen über eine Unendlichkeit verfügen, nutzt man künstliche Grammatiken der Form  $G = (N, T, P, s)$ , um diese zu beschreiben. Eine Grammatik  $G$  dieser Art verfügt über eine Menge von nichtterminalen Symbolen  $N$ , oder auch Variablen genannt, welche im Verlauf der Wortbildung durch die Menge der Produktionsregeln  $P$  der Form  $U \rightarrow V$  durch eine Kombination aus terminalen Zeichen der Menge  $T$  ersetzt werden. Das Startsymbol  $s$  ist zwingend die erste Variable, welche durch eine Produktion der Form  $s \rightarrow V$  ersetzt wird. Ein Wort gehört einer Grammatik  $G$  an, wenn es nur noch aus terminalen Symbolen der Menge  $T_G$  besteht und ausgehend von dem Startsymbol  $S_G$  mit den Produktionen der Menge  $P_G$  gebildet werden kann.<sup>1</sup>

Je nachdem nach welcher Art die Produktionsregeln einer Grammatik aufgebaut sind, lässt sich eine formale Sprache nach dem Informatiker Noam Chomsky in vier Typen einteilen. Grundsätzlich gehört jede Sprache dem Typ 0 der allgemeinen Sprachen an. Eine Sprache gehört immer einem Typ  $X$  an, wenn alle Bedingungen der Typen  $\leq X$  für alle Produktionsregeln der Form  $U \rightarrow V$  erfüllt sind. Die Bedingung für eine kontext-sensitive Sprache des Typs 1 besagt, dass durch eine Produktionsregel das Wort nicht verkürzt werden kann, also  $|U| \leq |V|$ . Für eine kontextfreie Sprache des Typs 2 gilt, dass durch eine Produktionsregel nur jeweils eine Variable ersetzt werden kann, also  $|U| = 1$ . Eine Sprache ist eine reguläre Sprache des Typs 3, wenn ein nichtterminales Symbol mit einer Produktionsregel durch ein terminales Symbol und ein optionales nichtterminales Symbol ersetzt wird, jedoch nicht durch mehrere, also  $U \leq a|aB$ .<sup>2</sup>

<sup>1</sup>[https://www.uni-ulm.de/fileadmin/website\\_uni\\_ulm/iui.inst.040/FormaleMethoden\\_der\\_Informatik/Vorlesungsskripte/FMdI-06--2010-01-10--FormaleSprachenVorlesung.pdf](https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.040/FormaleMethoden_der_Informatik/Vorlesungsskripte/FMdI-06--2010-01-10--FormaleSprachenVorlesung.pdf) ; 19.11.2018

<sup>2</sup>I. Wegener; Theoretische Informatik; Kap.5

Verwendung	Zeichen	Verwendung	Zeichen
Definition	=	Aufzählung	,
Endezeichen	;	Alternative	—
Option	[...]	Optionale Wiederholung	{...}
Gruppierung	(....)	Anführungszeichen 1. Variante	"..."
Anführungszeichen 2. Variante	'...'	Kommentar	(*...*)
Spezielle Sequenz	?...?	Ausnahme	-

Um die Darstellung von Sprachen zu vereinheitlichen, wurde die Erweiterte-Backus-Nauer-Form, kurz EBNF entwickelt. Die oben gegebenen Zeichen sind Teil des EBNF Iso-Standards und durch sie lässt sich jede beliebige Formale Sprache beschreiben.

### 1.1.2 Spezifikation unserer Scriptsprache anhand unserer Ansprüche an den Funktionsumfang

Für unser Programm haben wir uns grundlegend vorgenommen, Funktionen für das Versenden von Dateien, sowie Befehlen an einen zweiten PC bereitzustellen. Es erfordert also in gewissem Maßen eine Kommunikation zwischen dem Nutzer und dem Programm, um die Wünsche des Bedieners genauer zu spezifizieren und der Software zu übermitteln. Um dies zu bewerkstelligen entschieden wir uns, eine eigene formale Sprache zu entwickeln, mit dem Ziel Befehle an das Programm mit den entsprechenden Argumenten zu übermitteln, welche über ein Konsolenfenster eingegeben werden. Unsere Sprache sollte folgende Anweisungen beinhalten:

Befehl	Argumente
<code>send_file</code>	<code>ip_address file_name file_type</code>
<code>send_comm</code>	<code>ip_address comm_name [arguments]</code>
<code>get_file</code>	<code>ip_address file_name file_type</code>
<code>shutdown</code>	<code>ip_address</code>
<code>open</code>	<code>ip_address file_name file_type</code>

Die Befehle *send\_file* und *send\_comm* sind hierbei die allgemeinen Funktionen zum Senden einer Datei oder eines Befehls an die betriebssystemeigene Kommandozeile, jeglicher Art. Das Kommando *get\_file* ist identisch zu *send\_file* mit dem Unterschied, dass eine Datei von dem angefragten PC auf den anfragenden übertragen werden soll. *shutdown* und *open* sind spezielle versendete Anweisungen, zum Herunterfahren des Zielcomputers und Öffnen einer Datei, bei denen wir es aufgrund ihrer frequentierten Nutzung für sinnvoll hielten, sie eigenständig in unsere formale Sprache zu integrieren. Zwangsweise muss mit jeder getätigten Anweisung an das Programm eine IP-Adresse des Empfängercomputers der Form IPv4 oder IPv6 als Argument übergeben werden. Zusätzlich für Befehle, welche mit Dateien arbeiten sollen, müssen sowohl der Dateiname als auch das Dateiformat als Zeichenkette angefügt werden. Die Übermittlungsfunktion

von Anweisungen benötigt neben dem Anweisungsnamen als Zeichenkette auch optionale Argumente, mit welchem die Anweisung in der Kommandozeile ausgeführt werden soll.

Um unsere formale Sprache zu implementieren, entwickelten wir zuerst eine Darstellung der Erweiterten-Backus-Nauer-Form, diese ist im Anhang einzusehen. Eine solche Darstellung hat den Vorteil einer unkomplizierten Übertragung der Produktionsregeln in die Implementierung, welche die Nutzereingaben überprüft.

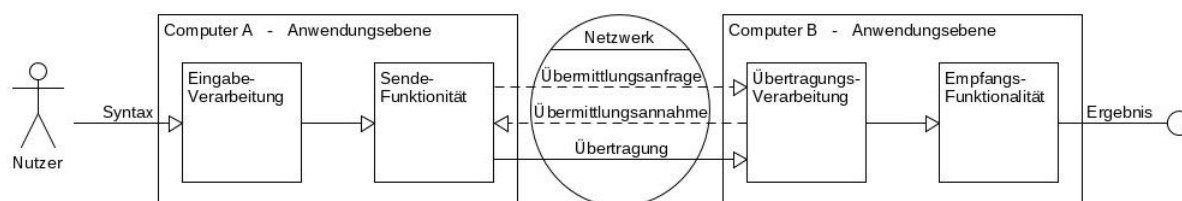
## 1.2 Implementierung der Anwendungssteuerung in der Scriptsprache Lua

### 1.2.1 Darstellung des grundlegenden Programmaufbaus und Programmablaufs

Grundlegende Aufgabe der Anwendungssteuerung ist es, die Eingaben die der Nutzer unserer Software tätigt, in Aktionen des Programms umzusetzen. Dazu werden zum einen auf dem Rechner der sendenden Person die nach der in Kapitel x.1 beschriebenen Syntax eingegebene Anweisung, in der Eingabeverarbeitung auf ihre Korrektheit überprüft, um anschließend aufgeschlüsselt zu werden und die gewünschte Sendefunktion zu initialisieren.

Nachdem die Daten mithilfe des Netzwerkprotokolls übermittelt wurden, werden sie von der Übertragungsverarbeitung, auf dem Empfängercomputer weiter bearbeitet. Je nach spezifiziertem Typ der Übermittlung, wird entweder die enthaltene Datei auf Fehler überprüft und anschließend endgültig gespeichert, oder die auszuführende Anweisung wird an die betriebssystemeigene Kommandozeile weitergegeben und ausgeführt.

Um keine ungewollten Übertragungen von potenziell schädlichen Dateien oder Anweisungen zu ermöglichen, wird für die Ausführung jeder Übermittlungsanfrage die Bestätigung des Empfängers benötigt.



### 1.2.2 Ansprüche an die Implementierung und die daraus resultierende Wahl der Programmiersprache

Wie in Kapitel 3.2.1 erläutert, wird auf der Anwendungsebene unseres Programmes, also dem Teil welcher ausschließlich auf einem Rechner läuft, die grundlegende Funktionalität implementiert. Da unser selbstentwickeltes Netzwerkprotokoll, welches die eigentliche Kommunikation zwischen 2 Computern bewirkt, relativ universell auf verschiedenartige Daten anwendbar ist, stellt dieses im Bezug auf den Funktionsumfang der Software nicht den limitierenden Faktor da. Um das Protokoll eventuell auch im Zeitraum nach

der eigentlichen Arbeit ausnutzen zu können, und nicht durch einen festgelegten Satz von Befehlen beschränkt zu werden, entschieden wir uns dazu diese Softwareebene in einer Scriptsprache zu implementieren. Diese Klasse von Programmiersprachen kennzeichnet sich dadurch aus, dass der vom Menschen lesbare Quelltext eines Programmes erst bei seiner Ausführung in Anweisungen übersetzt werden, welche für den Computer ausführbar sind. Im Gegensatz zu kompilierten Sprachen hat dies den Vorteil, dass Programmcode unkompliziert korrigiert und modifiziert werden kann.

In diesem Zusammenhang ergibt sich außerdem eine leichte Wartbarkeit, also die Möglichkeit eventuell auftretende Fehler zu beseitigen, als ein weiterer erfüllter Anspruch. Da wir stets zu dritt an unserer Software programmiert haben, sollte es jedem von uns mit möglichst geringem Aufwand möglich sein, Programmteile zu verbessern, auch wenn es sich nicht um den ursprünglichen Autor handelt.

Den Vorteil von kompilierten Programmiersprachen, dass sie sich durch eine höhere Leistungsfähigkeit besonders bei komplexeren Problemen auszeichnen, haben wir uns durch die Implementierung unserer Netzwerkteils in der Sprache C++ zu nutzen gemacht. Dies stellt den Anspruch, dass die gewählte Scriptsprache problemlos in Software der Sprache C++ einzubetten ist, und keine Probleme bei der Kommunikation zwischen Programmen der zwei Sprachen auftreten.

Da wir als Gruppenmitglieder unterschiedliche Betriebssysteme auf unseren Arbeitsrechnern benutzen, entstand als Nebenprodukt die Anforderung, dass das finale Programmsystem unabhängig sein muss, also ohne erheblichen Aufwand auf neue Betriebssysteme portierbar ist. Unser Fokus lag dabei für die Arbeit auf Computersystemen und wir haben mobile Geräte vorerst außen vor gelassen. Zu guter Letzt ist der Zeitraum der Seminarfacharbeit auch nur auf andertalb Jahre begrenzt, weshalb die gewählte Sprache mit einem geringem Lernaufwand benutzbar sein muss.

Als Kompromiss zwischen allen beschriebenen Ansprüchen entschieden wir uns für die Programmiersprache Lua. Da sie in reinem C geschrieben ist, weißt sie eine uneingeschränkte Integrationmöglichkeit unseres Netzwerkprotokolls vor, und lässt sich außerdem auf die verbreitetsten Betriebssystemarchitekturen wie Windows und Unix portieren.<sup>3</sup> Ein positiver Nebeneffekt dieser Wahl ist die hohe Leistungsfähigkeit von Lua, verglichen zu anderen weit verbreiteten Scriptsprachen. Der Sprachaufbau orientiert sich sehr nah an der Englischen Sprache, was ein schnelles Erlernen ermöglichte.

### **1.2.3 Erläuterung wesentlicher Elemente der Implementierung**

Eine zentrale Rolle in der Umsetzung unserer Eingabeverarbeitung spielt die Interpreter Funktion. Ihre Aufgabe besteht darin, die von dem Nutzer in Form einer Zeichenkette gelieferten Eingabe, in eine für das Programm verständliche Anweisung zu übertragen. Sie ruft außerdem die Funktionen auf, welche die gewünschte Funktionalität implementieren, und übergibt ihnen die eingegebenen Parameter zur weiterführenden Überprüfung.

---

<sup>3</sup>[www.lua.org/about.html](http://www.lua.org/about.html), 17.11.2018

### Listing 1: Interpreter Funktion

```
function interpret_input(ui_input)
    local content = split_input(ui_input)
    local commands = {
        ["send_file"]=0,
        ["send_comm"]=0,
        ["get"]=0,
        ["open"]=0,
        ["shutdown"]=0}
    if commands[content[1]]~=NIL then
        local result = _G[content[1]](content)
        print(result)
    else
        local name = debug.getinfo(1, "n").name.." : "
        local subject = string.format("%q",content[1])
        print("ERROR: "..name..subject.." - unbekannter Befehl!")
    end
end
```

Die Variable *content* ist eine Liste der eingegebenen Wörter, sie wird durch die Funktion *split\_input* erstellt, welche die rohe eingegebene Zeichenkette an den Leerzeichen teilt, und demnach wortweise abspeichert. In der Liste *commands* wird der Befehlssatz gespeichert, hierbei wird dem Namen von jedem verfügbaren Befehl ein Wert zugeordnet, es spielt keine Rolle welcher Wert es ist, wichtig ist nur, dass die Variablen initialisiert sind und nicht keinen zugeordneten Wert haben. In der folgenden *If – Else*-Bedingung wird überprüft, ob der eingegebene Befehl teil des Befehlssatzes ist, sprich ob in der Liste *commands* ein Wert für den gewünschten Befehl vorliegt. Ist dies nicht der Fall, gibt die Funktion einen Fehler der Form *ERROR : interpret\_content : EINGABE-unbekannterBefehl!* aus, und es wird keine Übertragung eingeleitet. Wenn die eingegebene Anweisung Teil des Befehlssatzes ist, so wird die entsprechende Funktion mit den übergebenen Parametern aufgerufen, und das Ergebnis in der Variable *result* gespeichert. Der Aufruf geschieht durch die Lua-Interne Funktion *.G[STRING](ARGUMENTS)*, welche eine Zeichenkette als Input fordert, und eine Funktion dieses Namens mit den gewünschten Argumenten *ARGUMENTS* aufruft.

Die gewählte Form der Implementierung, dass der Befehlssatz in einer Liste bestehend aus den Namen der zugehörigen Funktionen, gespeichert ist, ermöglicht eine unkomplizierte Erweiterung der grundlegenden Befehle durch neue Funktionalität, welche möglicherweise spezifischere Ansprüche an das Protokoll fordert. Beispiel hierfür ist die Implementierung von Streaming, die einen ständigen Erhalt der Netzwerkverbindung zwischen den zwei Rechnern benötigen würde.

Die eigentlichen Funktionen sind alle nach einem ähnlichen Schema aufgebaut. Es gibt zwei verschachtelte *If – Else*-Bedingungen die erfüllt werden müssen, um die eigentliche Übertragung auszulösen. In der ersten wird überprüft, ob die Anzahl der eingegebenen

Parameter mit der für den Befehl benötigten übereinstimmt. Ist dies der Fall, so wird eine erste Übertragung mit dem Zweck der Autorisierung gestartet. Es wird eine zu bestätigende Anfrage an den Empfänger-PC geschickt, welche standardmäßig nach 10 Sekunden ausläuft. Wenn eine Bestätigung in besagtem Zeitraum stattfindet, so wird die eigentliche Übertragung initialisiert. Schlägt eine der Bedingungen fehl, so wird der Vorgang abgebrochen und eine Fehlermeldung entsprechen der oben dargestellten Form ausgegeben.

Listing 2: Beispielhafte Sende Funktion

```
function send_NAME(args)
local name = debug.getinfo(1, "n").name
  if get_length(args)==INTEGER then
    local ARG1 = args[2]
    ...
    local ARGN = args[N+1]
    if authenticate(ip) then
      local blank = "c_sendNAME(ARG1, ..., ARGN)"
    else
      local type = " - Uebertragung wurde abgelehnt"
      print("ERROR: "..name..type)
    end
  else
    local type = " - Argumentenzahl unpassend"
    print("ERROR: "..name..type)
  end
end
end
```