Technisches Gymnasium Ravensburg

Kommandozeilen-interpreter

Am Beispiel der Bourne-Again Shell

Johannes Sonn
05.12.2018
TG 13/4
Gleichwertige Feststellung von Schülerleistungen
Computer-Technik bei Frau Fichtner

Inhaltsverzeichnis

Was ist ein Kommandozeileninterpreter?

Einleitung

Um einen Computer effizient nutzen zu können, sollte man in irgendeiner Form ihm Informationen übermitteln und Information zurückerhalten können. Diese Funktion übernimmt eine sogenannte Schnittstelle. Es ist ein sehr fundamentales Problem in der Computergeschichte. Mittlerweile gibt es eine Vielzahl an Computer-Benutzerschnittstellen. Zum Beispiel gibt es sprachbasierte, zeichenorientierte, natürliche oder graphische Benutzerschnittstellen. Diese verschiedenen Varianten bauen auf die Kommandozeile auf.

In den frühen Jahren der Entwicklung von Computer-Systemen musste eine Benutzerschnittstelle her, um Computer-Systeme steuern zu können. Beispielsweise sollte man Funktionen des Betriebssystems oder anderer Programme steuern und überwachen können. Außerdem wäre es auch von Vorteil, wenn man die Steuerung der Programme automatisieren könnte.

Die wohl am häufigsten genutzten Varianten einer Computer-Benutzerschnittstelle stellen heutzutage Graphische Benutzeroberflächen (GUI, engl. graphical user interface) und Natürliche Benutzerschnittstellen (NUI, engl. natural user interface) dar. Während eine GUI häufig mit Maus und Tastatur bedient wird, werden NUIs beispielsweise mit einem Touchscreen bedient. Außerdem gibt es auch die Möglichkeit der Bedienung eines Computers über eine Kommandozeile (CLI, engl. command line interface). Diese Art der Bedienung geschieht normalerweise nur über eine Tastatur zur Eingabe und ein Ausgabegerät, wie ein Bildschirm und war für lange Zeit die am meisten verbreitete Variante der Interaktion mit Computern.

In dieser Ausarbeitung möchte ich das Grundprinzip von Kommandozeilen und Kommandozeilen-interpretern erläutern, deren Geschichte und Entwicklung aufzeigen, Beispiele von CLIs geben und anhand eines CLIs Beispiele zur Verwendung zeigen.

Geschichte

Zuse Z3

Zu Beginn der Computerentwicklung verwendete Konrad Zuse 1941 bei dem ersten programmierbaren digitalen Rechner Zuse Z3 eine Dezimaltatstatur als Ein- und ein Lampenfeld als Ausgabe. Die Z3 hatte 200 Byte Arbeitsspeicher und konnte lediglich zwischen neun Befehlen unterscheiden. Dennoch war sie bereits in der Lage mit Gleitkommazahlen rechnen. Programme wurden über Lochstreifen eingelesen, wohingegen für Zahleneingaben die Tastatur benötigt wurde. Dies ähnelte von der Funktion her sehr den ab dem Jahre 1969 verkauften Bildschirm-Terminals.

Fernschreiber

Damit in den 1960 Jahren Großrechner benutzt werden konnten, wurden vom eigentlichen Computer getrennte Benutzerschnittstellen verwendet. Diese wurden als Konsole oder Fernschreiber bezeichnet und bestanden aus einem Schreib- oder Lochstreifenschreibautomat. Ein Schreibautomat besteht aus einer elektrischen Schreibmaschine und manchmal auch einem Lochstreifen-Stanzer und -Leser. Die elektrische Schreibmaschine ermöglichte es Tastendrücke über ein Kabel herauszuführen, aber auch über ein Signal einzelne Typenhebel anzusteuern. Die Signale von der Konsole wurden an den Computer weitergeleitet und von einem Kommandozeileninterpreter interpretiert. Dadurch konnten Befehle an den Computer erteilt werden. Dies wird jedoch später genauer beschrieben. Durch die Verwendung des Lochstreifen-Lesers konnten im Vorhinein gestanzte Befehlssequenzen automatisiert abgearbeitet werden.

Für die Verbindung zwischen Computer und Fernschreiber wurde die 40 mA serielle Schnittstelle / Stromschleife verwendet. Diese Schnittstelle erlaubte eine Kabellänge von bis zu mehreren hundert Metern. Die Reichweite konnte jedoch mithilfe des Telex-Netz um einiges erhöht werden. Das Telex-Netz (*Teleprinter Exchange*) wurde seit den 1930er Jahren verwendet, um Fernschreiber weltweit zu verbinden und wurde in den 1980er Jahren durch schnellere Übertragungsmedien verdrängt, da die Übertragungsgeschwindigkeit nur 50 Baud (50 Symbole pro Sekunde) betrug. Dies führte zu lediglich ca. 6,5 Zeichen pro Sekunde, da für jedes übertragene Zeichen 7,5 zusätzliche Steuersymbole übertragen werden mussten und die Geschwindigkeit an mechanische Drucker angepasst war. Teletex wurde größtenteils durch Fax, Teletex und E-Mail ersetzt. Dennoch gibt es noch vereinzelte Anwender des Telex-Standards, wie beispielsweise Banken. Da das Telex-Netz nicht mehr betrieben wird, werden heutzutage andere Übertragungsmedien verwendet. Beispielsweise gibt es i-Telex[1], das über TCP/IP-Verbindungen funktioniert und auf eine zentrale Teilnehmerliste angewiesen ist.
Damit Fernschreiber und Computer verschiedener Hersteller miteinander kommunizieren konnten, wurde 1963 die 7-Bit ASCII-Kodierung (*American Standard Code for Information Interchange*) eingesetzt. Währenddessen entwickelte IBM den EBCDIC (*Extended Binary Coded Decimal Interchange Code*), welcher auf die BCD-Kodierung basierte, 8 Bit pro Zeichen verwendete und sich nicht durchsetzen konnte, da es nicht mit ASCII kompatibel war und nur mit IBM Maschinen funktionierte. Für die Verwendung einer Zeichenkodierung mit Fernschreibern wurden besondere Steuerzeichen benötigt. Eines der 33 nicht druckbaren Zeichen ist beispielsweise der Line Feed (kurz: LF / *Zeilenvorschub*), welcher dem Fernschreiber mitteilt, dass er die nächste Zeile ansteuern soll.

Glas-Fernschreiber

Mit der steigenden Popularität von Röhrenbildschirmen in den späten 1960er Jahren, änderte sich die gängige Art der Ausgabe an Computerterminals von Druckern zu Bildschirmen. Die, auch als Datensichtgerät bezeichneten, Computerterminals bestanden nun aus einer Tastatur zu Eingabe und einem meist monochromen Röhrenbildschirm zur Ausgabe. Aufgrund des gläsernen Bildschirmes wurden diese Geräte auch als glass TTY (*glass teletype*) referenziert und wurden. Die Verwendung von öffentlichen Standards und auch proprietärer Standards (beispielsweise durch IBM) endete größtenteils mit der Einführung des LAN-Standards Ethernet. Heutzutage werden hauptsächlich Internetprotokoll (IP) basierte Protokolle, wie Telnet oder SSH, verwendet.

Durch die Ersetzung des Druckers als Ausgabe mit einem Bildschirm, konnten dargestellte Inhalte aktualisiert werden. Dies ermöglichte Bewegungen des Cursors (Einfügemarke) und andere Dinge, wie beispielsweise den Bildschirminhalt zu löschen.

Kommandozeile

Als Kommandozeile (CLI) wird die Textzeile bezeichnet, die, meist nach betätigen der Return-Taste, an einen Kommandointerpreter (siehe unten) weitergegeben wird. Sie ist ausschließlich auf Text basiert und erlaubt es nicht mit beispielsweise einer Computermaus zu steuerten. Zur Eingabe dieser Kommandozeile dient eine Tastatur, wie es bei Fernschreibern und moderneren Terminals der Fall ist. Kommandozeilen sind zum Beispiel bei der Steuerung eines Computers via Fernschreiber, der Steuerung einer CNC-Maschine (G-Code) oder bei Computerprogrammen innerhalb einer GUI (z.B. Matlab oder Debugger) zu finden.

Zu Beginn der Zeile wird standardmäßig eine Eingabeaufforderung (Prompt) angezeigt. Dieser Prompt zeigt grundlegende Informationen an. Beispielsweise wird der Name des Computers, der Name des Nutzers und das aktuelle Dateiverzeichnis angezeigt. Außerdem teilt der Prompt dem

[1] https://www.i-telex.net

Nutzer mit, dass er erneut einen Befehl eingeben kann. Neben den oben genannten Informationen kann man ebenso oftmals nur durch den Prompt erfahren, ob der Nutzer Systemadministrator (root) ist. Bei Nutzern mit root-Rechten wird standardmäßig eine Raute `#` am Ende der Eingabeaufforderung angezeigt, wohingegen bei nicht root-Nutzern ein Dollarzeichen `$` verwendet wird.

Kommandointerpreter

Nachdem ein Befehl über die Kommandozeile eingegeben und der Return-Taste beendet wurde, wird der eingegebene Text an einen Kommandointerpreter weitergeleitet. Die Hauptaufgabe eines Kommandointerpreters ist es zu versuchen Befehle aus einzelnen Textzeilen zu identifizieren und anschließend auszuführen. Diese Textzeilen erreichen den Kommandointerpreter über einen Standard-Datenstrom, welcher als *Standardeingabe* (`stdin`) bezeichnet wird. Während und nach der Ausführung eines Befehls, können über einen anderen Datenstrom namens *Standardausgabe* (`stdout`) Informationen zu einem laufenden Prozess oder die Ergebnisse eines Befehls ausgegeben werden. Sollte ein fehlerhafter Befehl eingegeben oder während dem Ausführen des Befehls ein Fehler auftreten, wird üblicherweise eine Fehlermeldung über die *Standardfehlerausgabe* (`stderr`) ausgegeben.

Neben dem Erkennen einzelner Befehle und dem Ausführen der erkannten Befehle, werden ebenso optional angegeben Parameter entgegengenommen und beim Ausführen eines Befehls beachtet. Beispielsweise nimmt der Befehl `echo` etwas Text als Parameter ein und gibt ihn wieder über `stdout` aus.

Datenstrom-Umleitung

Wie bereits oben genannt, verwenden Kommandozeileninterpreter drei verschiedene Standarddatenströme zum Dateneinlesen und zur Daten- bzw. Fehlerausgabe mit den englischen Abkürzungen `stdin`, `stdout` und `stderr` (siehe Abbildung 1). Das Einlesen von Daten mithilfe von `stdin` geschieht standardmäßig über die Tastatur, wohingegen die Ausgabe von `stdout` und `stderr` über einen Monitor funktioniert. Die einzelnen Datenströme können jedoch umgeleitet werden, sodass beispielsweise die Ausgabeströme in eine Datei oder an einen Drucker weitergeleitet werden. Für die Umleitung eines von `stdout` und `stderr` wird eine schließende spitze Klammer (`>`) verwendet. Dies sieht dann wie folgt aus:

```
$ Befehl > Ausgabe.txt
$
```

In diesem Beispiel wird das Ergebnis von `Befehl`, das über `stdout` ausgegeben wurde, in die Datei `Ausgabe.txt` geschrieben. Solange der Befehl problemlos ausgeführt werden kann wird nichts auf der Konsole bis auf die nächste Eingabeaufforderung ausgegeben. Sollte es jedoch zu einem Fehler kommen, da beispielsweise der Befehl nicht korrekt eingegeben wurde, wird über `stderr` eine Fehlermeldung ausgegeben. `Stderr` wird standardmäßig auf dem Monitor ausgegeben und würde bei einem fehlerhaften Befehl ungefähr so aussehen:

```
$ FehlerhafterBefehl > Ausgabe.txt
Command 'FehlerhafterBefehl' not found.
$
```

In diesem Fall wird sich nichts in `Ausgabe.txt` befinden, da es zu einem Fehler kam und keine Ergebnisse über `stdout` ausgegeben werden konnten.

Wenn es vonnöten ist in ein Logfile ausschließlich Fehlermeldungen zu schreiben, kann man dies mit einer sehr kleinen Änderung erreichen. Dafür wird bei der Umleitung des Datenstroms, durch das Hinzufügen einer `2` unmittelbar vor der spitzen Klammer, der gewünschte Datenstrom spezifiziert. Die `2` kann ebenfalls mit beispielsweise `1` (für `stdout`) ersetzt werden. Ein Beispiel hierfür sieht wie folgt aus:

```
$ FehlerhafterBefehl 2> logfile.log
$
```

Nun wird nichts auf der Konsole ausgeben, sondern der Fehler von `stderr` in logfile.log `geschrieben`.

In den genannten Fällen der Datenstrom-Umleitung werden neue Dateien erstellt, falls die Dateien (`Ausgabe.txt` und `logfile.log`) noch nicht bereits existieren. Sollte eine Datei mit demselben Namen bereits bestehen, wird dessen Inhalt mit dem umgeleiteten Datenstrom überschrieben. Möchte man aber Daten zu einer Datei lediglich hinzufügen, kann man eine doppelte spitze Klammer verwenden. Am Beispiel des Logfiles würde dies so aussehen:

```
$ FehlerhafterBefehl 2>> logfile.log
$
```

Dies funktioniert sowohl, wie hier im Beispiel, mit `stderr` als auch mit `stdout`.

Der Standardeingabedatenstrom (`stdin`) kann ebenfalls umgeleitet werden. Hier wird jedoch viel eher die Quelle des Datenstroms festgelegt, als ein Datenstrom umgeleitet. Diese Art der Datenstrom-Umleitung wird nicht so häufig wie die anderen verwendet, aber kann doch hilfreich sein. Es gilt größtenteils dasselbe wie bei der Umleitung von `stdout` und `stderr`, allerdings ist zu beachten, dass eine öffnende spitze Klammer, anstelle einer schließenden Klammer verwendet werden muss. So kann beispielsweise der Inhalt einer Datei der `stdin`-Datenstrom für ein Programm sein. Unter Unix- und unixartigen Systemen gibt es ein Programm namens `cat` (Abkürzung für *concatenate*, engl.: verketten), das den Inhalt einer Datei oder der Standardeingabe ausgeben kann.

```
$ cat < Datei.txt
Das ist die erste Zeile der Datei
Das ist die zweite Zeile
Hier ist noch eine Zeile
Und eine vierte Zeile ist auch noch zu finden
$
```

In diesem Beispiel wird der Inhalt von `Datei.txt` als `stdin` an `cat` weitergeleitet und von `cat` via `stdout` auf der Konsole ausgegeben.

Eine Kombination aus Ausgabe- und Eingabeumleitung könnte so aussehen:

```
$ Befehl < Eingabe-Datei.txt >> Ausgabe-Datei.txt
$
```

Sollte man beide Ausgabedatenströme (`stdout` und `stderr`) in beispielsweise eine Datei umleiten, könnte man `stdout` in eine Datei und `stderr` (2) in `stdout` (1) umleiten, sodass beide Datenströme in `stdout` vereint in die Datei umgeleitet werden.

```
$ Befehl > Ausgabe.txt 2>&1
$
```

Piping

Neben dem Umleiten von Datenströmen in Dateien oder ähnliches, kann ebenso der Ausgabe-Datenstrom eines Programms in den Eingabe-Datenstrom umgeleitet werden. Damit ist es möglich Ergebnisse eines Programms zunächst zu filtern oder weiterzuverarbeiten bevor sie letztendlich auf

der Konsole ausgegeben oder in einer Datei gespeichert werden. So kann beispielsweise ein Logfile nach speziellen Ereignissen anhand von Schlüsselwörtern durchsucht werden.

```
$ cat logfile.log | grep SEVERE
$
```

Um zwei Befehle mit einer *anonymen Pipe* (engl.: wortwörtlich Rohrleitung) zu verbinden, wird das sogenannte Pipe-Symbol (`|`) verwendet. In diesem Fall werden mithilfe von `grep` ausschließlich Zeilen, in welchen das Wort `SEVERE` gefunden wurde, also nur schwerwiegende Fehler, ausgegeben.

Außerdem kann über den Befehl `mkfifo` ein FIFO erstellt werden, der als Pipe dienen kann und von mehreren Prozessen unterschiedlichen Ursprungs verwendet werden kann. Es können sogar Prozesse von einem anderen Computer auf diese Pipe zugreifen, solange sie autorisiert sind und den Namen der Pipe wissen. Da FIFO-Pipes Teil des Dateisystems sind, bleiben sie nach ihrer Verwendung bestehen.

```
$ mkfifo fifo_pipe
$ cat Datei.txt > fifo_pipe &
[1] 13022
$ grep zweite < fifo_pipe
Das ist die zweite Zeile
[1]+  Done                    cat Datei.txt > test
$
```

In dem oben gezeigten Beispiel wird eine FIFO-Pipe namens `fifo_pipe` erstellt und anschließend mit dem Ergebnis von `cat Datei.txt` gefüllt. Mit dem dritten Befehl wird der Inhalt von `fifo_pipe` in `grep zweite` umgeleitet und somit nur die Zeile mit dem Schlüsselwort „zweite“ ausgegeben und der Prozess mit der PID `13022` beendet.

Neben den bereits erläuterten Pipes gibt es auch eine Alternative, um Datenströme zwischen Prozessen umzuleiten. Allerdings können mit dieser Variante nur `stdout` und `stderr` umgeleitet werden.

```
$ Befehl1 $(Befehl2)
$
```

Hier werden `stdout` und `stderr` von Befehl2 als Argument für Befehl1 übergeben.

Parameter

Da Programme, die innerhalb einer Konsole verwendet werden, selten die Möglichkeit bieten interaktiv gesteuert zu werden, werden oftmals bereits beim Aufruf des Programms Parameter angegeben, die den Ablauf des Programms beeinflussen. Die Parameter können Optionen oder Argumente sein.

Optionen

Optionen werden auch Flags genannt und unter Unix- und Linux-Systemen mit einem einfachen (`-`) oder doppelten Bindestrich (`--`) und unter Windows mit einem Schrägstrich (`/`) gekennzeichnet. Manche Optionen eines Programms fordern bei Verwendung einen Wert, der unmittelbar der Flag folgt.

```
$ Befehl --flag
$ Befehl -f
$ Befehl --option /Pfad/zu/einer/Datei
$ Befehl -o /Pfad/zu/einer/Datei
$
```

Flags können oft in einer langen Schreibweise (hier `--flag`) oder abgekürzt (hier `-f`) angegeben werden. Dabei ist zu beachten, dass bei der längeren Schreibweise ein doppelter Bindestrich, wohingegen bei Abkürzungen ein einfacher Bindestrich verwendet wird. Außerdem ist in dem obigen Beispiel die Angabe eines Wertes (in diesem Fall ein Dateipfad) gezeigt. Ein Beispiel anhand von dem Befehl `mv` (move: verschieben, umbenennen) sieht wie folgt aus:

```
$ mv source.txt target.txt --verbose
Renamed ‘source.txt’ -> ‘target.txt’
$ mv source.txt target.txt -v
Renamed ‘source.txt’ -> ‘target.txt’
$ mv source1.txt source2.txt --target-directory=/path/to/target
$ mv source1.txt source2.txt -t=/path/to/target
$
```

Die ersten beiden Befehle dieses Beispiels bewirken das gleiche, indem sie `source.txt` zu `target.txt` umbenennen und ausführlich ausgeben was passiert. Die anderen beiden Befehle verschieben dahingegen `source1.txt` und `source2.txt` in das angegebene `target_directoy` und geben dabei keine weiteren Informationen über `stdout` aus. Bei Unix- und Unix-ähnlichen Systemen wird bei Flags sowie bei Befehlen zwischen Groß- und Kleinschreibung unterschieden. So haben beispielsweise die Flags `-a` und `-A` eine unterschiedliche Bedeutung. Wenn mehrere Flags angegeben werden, können diese nach einem einfachen Bindestrich ohne Leerzeichen aufgelistet werden.

Argumente

Ein anderer Teil von Kommandozeilenparameter sind sogenannte Argumente. Sie geben dem aufgerufenen Programm meistens grundlegende Informationen, wie auf welchen Pfad das Programm angewendet werden soll und sind oftmals optional. So nehmen beispielsweise die Befehle `ls` (list directory content) und `rm` (remove file or directories) einen Pfad zu einem Ordner oder einer Datei ein.

```
$ ls /tmp/
tmpfile
$ rm /tmp/tmpfile
$
```

Nach dem Aufrufen von `ls /tmp/`, wird der Inhalt von dem Ordner `/tmp/` ausgegeben. In diesem Fall befindet sich nur die Datei `tmpfile` in diesem Ordner. Anschließend wird diese Datei, durch angeben ihres Pfades beim Aufruf von `rm`, gelöscht.

Es gibt Befehle, die keine Parameter akzeptieren (z.B. `clear`[2]) oder keine Parameter benötigen, aber akzeptieren (z.B. `pwd`[3]). Dahingegen gibt es Befehle, wie `rm` oder `cp`, die unbedingt ein oder mehrere Argumente benötigen.

Wildcards

Damit man beispielsweise nicht die Pfade zu mehreren ähnlichen Dateien angeben muss, kann man mit Wildcards arbeiten. Wildcard sind Platzhalter für beliebig viele Zeichen und wird in der `bash` mit einem Asterisk (`*`) dargestellt. So kann man mit dem Befehl `rm *.html` alle HTML-Dateien in dem aktuellen Verzeichnis löschen. Möchte man die zu löschenden HTML-Dateien genauer spezifizieren, damit nicht andere HTML-Dateien ausversehen gelöscht werden, so kann man ein Fragezeichen als Platzhalter verwenden. Sind beispielsweise die Dateien `index.html`, `datei1.html`, `datei2.html`,

[2] https://www.computerhope.com/unix/uclear.htm
[3] https://www.computerhope.com/unix/upwd.htm

`datei2.orig.html` und `datei3.html` in einem Verzeichnis und es sollen nur die Dateien, die mit „datei“ beginnen und auf „.html“ enden, außer der `datei2.orig.html`, gelöscht werden, könnte man den Befehl rm `datei?.html` verwenden. Wenn nur datei1.txt und datei2.txt gelöscht werden sollen, kann man dies in mehreren verschiedenen einzeiligen Befehlen erzielen. Bei der Verwendung von `datei[1-2].txt` oder `datei[1..2].txt`, werden alle Dateien gelöscht die zwischen „datei“ und „.txt“ eine Zahl zwischen 1 und 2 im Dateinamen haben. Mit `datei[12].txt` oder `datei{1,2}.txt` werden alle Dateien mit einer 1 oder 2 an der entsprechenden Stelle im Dateinahmen referenziert. Mit einem Ausrufezeichen kann eine Wildcard negiert werden und alle Dateien, die dem eigentlichen Muster entsprechen, werden exkludiert (z.B. `datei[!2].txt` exkludiert die `datei2.txt`). Um Wildcards zu vermeiden, wenn man ein Zeichen, dass eine Wildcard definieren würde, benutzen möchte, werden Backslashs (`\`) verwendet.[4]

Befehle logisch verknüpfen

Um die Benutzung einer Kommandozeile zu beschleunigen, bieten sie oft die Möglichkeit Befehle logisch zu verknüpfen. Dies kann ebenso sehr hilfreich beim Scripting sein. So kann bei erfolgreichem oder gescheitertem Ausführen eines Befehls ein anderer ausgeführt werden.

```
$ compile_program && run_program || show_log
$
```

Beispielsweise kann damit, nach erfolgreichem Kompilieren eines Programms, dieses direkt ausgeführt werden oder bei gescheitertem Kompilieren oder Ausführen des Programms, das Log mit der entsprechenden Fehlermeldung angezeigt werden. Eine logische UND-Verknüpfung wird durch `&&` zwischen den beiden Befehlen erreicht, wohingegen bei einer ODER-Verknüpfung `||` verwendet wird.

Variablen

Um Daten speichern und zu einem anderen Zeitpunkt wieder verwenden zu können, werden Variablen verwendet. Es können verschiedene Datentypen gespeichert werden, allerdings werden meistens String-Variablen (Zeichenketten) verwendet. Variablennamen können aus Buchstaben, Sonderzeichen und Zahlen bestehen, aber dürfen nie mit einer Zahl beginnen. Die Syntax zum Setzen und Abfragen kann von CLI zu CLI unterschiedlich sein. In Windows `cmd.exe` wird eine Variable mit dem Namen `MEINE_VARIABLE` mit dem Befehl `SET MEINE_VARIABLE=Inhalt` deklariert (wenn sie noch nicht bereits existiert) und mit dem String „Inhalt“ initialisiert. Zum referenzieren der Variable, werden vor und nach dem Variablennamen ein Prozentzeichen (`%`) hinzugefügt. Das Ausgeben des Inhalts von `MEINE_VARIABLE` wird mit dem Befehl `ECHO %MEINE_VARIABLE%` erreicht. Im Gegensatz zu `cmd.exe` wird in der `bash` eine Variable deklariert und initialisiert, indem unmittelbar nach dem Variablenname ein Gleichheitszeichen und der Wert, der gespeichert werden soll, angegeben werden. Um die Variable zu referenzieren, wird vor dem Variablenname ein Dollarzeichen (`$`) hinzugefügt. Das Beispiel mit `MEINE_VARIABLE` sieht, dann wie folgt aus:

```
$ MEINE_VARIABLE=Inhalt
$ echo $MEINE_VARIABLE
Inhalt
$ echo $meine_variable
```

[4] http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm

```
$
```

Anhand dieses Beispiels ist zu erkennen, dass die bash zwischen Groß- und Kleinschreibung unterscheidet. Beim Referenzieren der Variable `meine_variable`, wird an den Befehl ein leerer String übergeben und es wird nur der standardmäßige Zeilenumbruch und kein Fehler ausgegeben. Dahingegen unterscheidet `cmd.exe` nicht zwischen Groß- und Kleinschreibung und gibt beim referenzieren einer unbekannten Variablen einfache ihr Name zurück:

```
C:\> echo %UNBERKANNTE_VARIABLE%
%UNBEKANNTE_VARIABLE%
C:\>
```

Variablentypen

In der bash können Variablen in vier verschiedene Typen unterteilt werden. Diese Typen sind Strings, Integers (ganze Zahlen), Konstanten und Arrays. Der Typ einer Variablen kann beim Deklarieren mit dem Schlüsselwort `declare` definiert werden.

```
$ declare string_var_1=Inhalt
$ declare string_var_2=05
$ declare -i integer_var_1=5
$ declare -i integer_var_2=05
$ declare -i integer_var_3=024
$ echo $string_var_1
Inhalt
$ echo $string_var_2
05
$ echo $integer_var_1
5
$ echo $integer_var_2
5
$ echo $integer_var_3
20
$
```

In dem obigen Beispiel kann man sehen, dass Variablen standardmäßig als String gespeichert werden, aber der Datentyp mit der Flag `-i` zu Integer geändert werden kann. Sollte der Wert, der in der Integer-Variable gespeichert werden soll, mit einer Null beginnen, wird versucht den Wert als Integer zu interpretieren, was nicht immer wie gewollt funktioniert.

Umgebungsvariablen

Umgebungsvariablen werden auch globale Variablen genannt und können durch ihren Wert das Verhalten von Programmen beeinflussen. Sie sind von allen Kommandozeileninterpretern eines Systems erreichbar und werden dynamisch geändert. Mit den Befehlen `printenv` oder `env`, können diese Variablen unter bash eingesehen werden. Dahingegen können in `cmd.exe` die Umgebungsvariablen über den Befehl `SET` aufgelistet werden. Umgebungsvariablen beinhalten Werte, wie den Usernamen, Pfade unter denen ausführbare Dateien zu finden sind (`PATH`), welcher Kommandozeileninterpreter verwendet wird (`SHELL`) und in welchem Verzeichnis der Nutzer sich befindet (`PWD`).

```
$ env
USER=root
HOME=/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
SHELL=/bin/zsh
PWD=/root
$
```

Dieses Beispiel von Werten; der Umgebungsvariablen eines Ubuntu-Systems; ist nicht vollständig und zeigt, dass die `zsh` als Kommandozeileninterpreter verwendet wird und der User `root` heißt.

Bei den vorigen Beispielen gelten die deklarierten Variablen nur in dem aktuellen Prozess und allen Kindprozessen. Damit Variablen in einer kompletten Session (Sitzung) gelten und zur Umgebungsvariable werden, müssen sie exportiert werden. Das heißt, dass sie den anderen Prozessen des Systems bekannt gemacht werden. Die Syntax, um das zu erreichen, ist zwischen einzelnen Linux-CLIs unterschiedlich. In der Bourne-Shell (`sh`) und ähnlichen CLIs wird zuerst eine Variable deklariert und mit einem Wert initialisiert und anschließend mit dem Befehl `export` bekannt gemacht (`export variablen_name`). Bei der Korn-Shell[5][6] (`ksh`), `bash` und ähnlichen CLIs kann dies verkürzt werden, indem bereits bei der Deklarierung vor dem Variablennamen ein „`export`“ hinzugefügt wird. Andere CLIs (z.B. `csh`) verwenden den Befehl `setenv`. Sollen Umgebungsvariablen über eine Session hinaus geltend gemacht werden, so müssen sie in bestimmten Konfigurationsdateien spezifiziert werden.

Bash

Die GNU Bourne-Again Shell (`bash`) ist eine Shell (Kommandointerpreter) des GNU Betriebssystems, stammt von UNIXs Bourne Shell (`sh`) ab und ist in vielen Unix-basierten Systemen zu finden, da sie oft als Standard-Shell festgelegt ist. Sie wurde anfangs von Brian Fox entwickelt und weißt Eigenschaften von anderen Shells, wie von der Korn Shell (`ksh`) oder C Shell (`csh`) auf. Beispielsweise wurden die Kommandohistorie übernommen und sie ist größtenteils kompatibel mit ihren Vorgängern. Ihr Name ist eine Anspielung auf Stephen Bournes Shell (`sh`) im Sinne von *wieder/erneut eine Bourne Shell* oder *wiedergeborene Shell*, da ihr Name lautsynonym zu *borne again* (engl. wiedergeboren) ist. Die bash wird zurzeit von Chet Ramey weiterentwickelt. Aufgrund den vielen Gemeinsamkeiten mit anderen Shells ist sie relativ universell und vereint die positiven Aspekte anderer Shells.

Die Konfiguration der bash kann in systemweiten Konfigurationsdateien vorgenommen werden. Allerdings kann jeder Benutzer durch benutzerspezifische Konfigurationsdateien (im Home-Verzeichnis) die Bash anpassen. Diese Konfigurationen können beispielsweise das Aussehen der Prompt bestimmen.

Scripting/Programmierung

Als Kommandointerpreter kann die Shell Kommandos von der Kommandozeile, aber auch von (Skript-)Dateien interpretieren. Dies erlaubt es routinemäßig vordefinierte Kommandoabfolgen auszuführen. Außerdem können auch bedingte Verzweigungen, Schleifen und Funktionen verwendet werden.

Hello World

```
#!/bin/bash
clear
string_var=“Hello World“
echo $string_var
```

Das obere kann in eine Datei namens `hello_world` gespeichert werden und mit dem Befehl `bash hello_world` aufgerufen werden. Die erste Zeile in diesem Skript definiert mit welchem Programm es auszuführen ist und wenn der Nutzer das Recht dazu hat, erlaubt es das Skript direkt auszuführen. Dafür muss nur der Pfad zu dem Skript als Befehl eingegeben werden (`./hello_world`). Sobald das

[5] http://www.kornshell.org/
[6] https://github.com/att/ast

Skript ausgeführt wird und die zweite Zeile erreicht wurde, wird der Inhalt des Bildschirms gelöscht und die Variable `string_var` mit „Hello World“ initialisiert. Abschließend wird die eben initialisierte Variable ausgegeben. Aufgrund des Leerzeichens zwischen „Hello“ und „World“ werden Anführungszeichen benötigt, sonst würde „World“ als Befehl interpretiert werden.

Parameter und User-Input

Parameter, die beim Aufruf eines Skripts angegeben werden, können mit den Variablen `$1` bis `$n` referenziert werden. Die Variable `$0` beinhält den Namen / Pfad des Skripts. Zusätzliche Informationen können mit dem Befehl `read` vom User / von `stdin` eingelesen werden.

```
#!/bin/bash
echo “$0 is running”
echo “What is your name?“
read name
echo “Hello, $name”
```

Nach Ausführen dieses Skripts wird ausgegeben, dass es läuft und nach Erfragen und Einlesen des Namens des Nutzers, wird der Nutzer mit Namen gegrüßt.

Funktionen und Schleifen

```
#!/bin/bash
function greet {
    echo “What is your name?”
    read name
    echo “Hello, $name”
}
function fibonacci {
    letztes=0
    vorletztes=1
    for i in {1..10}
    do
        # $((..)) is needed so it gets interpreted as arithmetic expression rather than
        # concatenation
        neues=$(($letztes + $vorletztes))
        vorletztes=$letztes
        letztes=$neues
        echo $neues
    done
}
greet
fibonacci
```

In diesem Beispiel sieht man, dass Funktionen mit dem Schlüsselwort `function` beginnen, mit geschweiften Klammern eingegrenzt werden und mit nur ihrem Namen aufgerufen werden. Die Einrückung in Funktionen und Schleifen ist optional, aber hilft der Lesbarkeit. Mit for-Schleifen kann durch Listen iteriert werden. So kann man wie in dem Beispiel durch alle ganzen Zahlen von 1 bis 10, eine bestimmte Liste (z.B. `for datei in datei1 datei2 datei3`) oder Ergebnisse eines Befehls (z.B. `for file in $(ls)`) iterieren.

Neben der for-Schleife gibt es noch die while- und die until-Schleife:

```
while [ <Bedingung> ]
do
    <Befehl>
done
```

```
until [ <Bedingung> ]
do
    <Befehl>
done
```

Die while- und until-Schleifen unterscheiden sich darin, dass bei der while-Schleife die Befehle (`<Befehl>`) ausgeführt werden, solange die Bedingung (`<Bedingung>`) als *wahr* interpretiert wird (z.B. 1 ist) und bei der until-Schleife die Befehle solange ausgeführt werden bis die Bedingung als *falsch* (z.B. 0) interpretiert wird.

Nützliche Befehle

`pwd`	Gibt den Namen des aktuellen Verzeichnisses aus (*print working directory*)
`ls`	Listet den Inhalt eines Verzeichnisses auf
`cd`	Wechselt in das als Argument angegebene Verzeichnis. Sollte kein Argument angegeben sein, wird in das Home-Verzeichnis gewechselt. (*change directory*)
`man`	Stellt eine interaktive Anleitung zu einem als Argument angegebenen Befehl zur Verfügung. Viele Befehle verfügen über die `-help`-Flag und geben bei Verwendung dieser Flag ebenfalls eine kurze Erklärung aus. (manuals)
`cat`	Konkateniert (zusammenfügen) die als Argumente angegebenen Dateien und gibt das Ergebnis aus. (*concatenate*)
`mkdir`	Erstellt ein neues Verzeichnis mit dem als Argument angegebenen Namen (*make directories*)
`echo`	Gibt eine als Argument angegebene Textzeile aus. Kann mit Datenstromumleitung zum Bearbeiten von Dateien helfen.
`touch`	Aktualisiert den Zeitstempel einer Datei. Sollte die als Argument angegebene Datei nicht existieren, wird sie erstellt.
`rm`	Löscht die als Argument angegebene Datei. (*remove*)
`rmdir`	Löscht das als Argument angegebene Verzeichnis. Das Verzeichnis muss hierfür bereits leer sein. (*remove directory*)

Grafiken

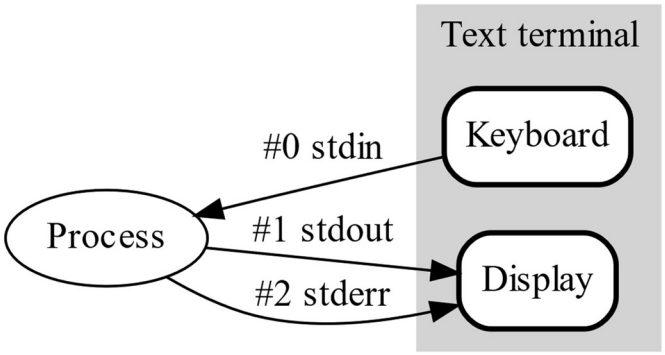

Abbildung 1 Die drei Standard-Datenströme

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig angefertigt und alle verwendeten Quellen mit Datum des letzten Aufrufs aufgelistet habe.

Vogt, 05.12.2018, Johannes Sonn

Quellverzeichnis

Websites

https://de.wikipedia.org/wiki/Kommandozeileninterpreter (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Zuse_Z3 (16:45 05.12.2018)
http://www.konrad-zuse.net/konrad-zuse/erfindungen/der-rechner-z3/seite01.html (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Standard-Datenströme (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Kommandozeile (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Command-line_interface (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Shell_(computing) (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Systemkonsole (16:45 05.12.2018)
https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Telex#Teilnehmeranschaltung (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Fernschreiber (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Schreibautomat (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Terminal_(Computer) (16:45 05.12.2018)
https://www.i-telex.net/grundlagen/ (16:45 05.12.2018)
http://tldp.org/LDP/abs/html/io-redirection.html (16:45 05.12.2018)
http://www.linfo.org/redirection.html (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Redirection_(computing) (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Pipe_(Informatik) (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Standard_streams (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Pipes_und_Filter (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Unix-Shell (16:45 05.12.2018)
https://www.shell-tips.com/2006/11/04/using-bash-wildcards/ (16:45 05.12.2018)
http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm (16:45 05.12.2018)
https://tiswww.case.edu/php/chet/bash/bashref.html#What-is-Bash_003f (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Bash_(Unix_shell) (16:45 05.12.2018)
http://www.linfo.org/create_shell_1.html (16:45 05.12.2018)
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-5.html (16:45 05.12.2018)
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-7.html (16:45 05.12.2018)
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-8.html (16:45 05.12.2018)
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-10.html (16:45 05.12.2018)
https://en.wikipedia.org/wiki/Environment_variable (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Variable_(Programmierung) (16:45 05.12.2018)
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_10_01.html (16:45 05.12.2018)
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html (16:45 05.12.2018)
https://de.wikipedia.org/wiki/Umgebungsvariable (16:45 05.12.2018)
http://environmentvariables.org/ (16:45 05.12.2018)

Grafiken

https://github.com/odb/official-bash-logo (16:45 05.12.2018)
https://upload.wikimedia.org/wikipedia/commons/7/70/Stdstreams-notitle.svg (16:45 05.12.2018)