

Sie sind hier: [FH Wedel](#) > [Mitarbeiter](#) > [Malte Heins](#) > [Übung: UNIX](#) > [Aufgaben](#) > [Aufgabe 3](#)

Aufgabe 3

Shell-Skripte, Parameter, Variablen, Schleifen und bedingte Anweisungen

Aufgabenstellung

In dieser Aufgabe sollt ihr ein Shell-Skript schreiben, welches einfache mathematische Berechnungen mit den folgenden Operationen durchführen kann:

- » Addition (ADD)
- » Subtraktion (SUB)
- » Multiplikation (MUL)
- » Ganzzahlige Division (DIV)
- » Rest der ganzzahligen Division (MOD)
- » Exponentiation (EXP)

Hilfeausgabe

Im einfachsten Fall ruft man das Programm jedoch nur mit dem Parameter `--help` (oder `-h`) auf, wodurch es den folgenden Hilfetext ausgibt:

```
Usage:
pfcalc.sh -h | pfcalc.sh --help

prints this help and exits

pfcalc.sh NUM1 NUM2 OP [NUM OP] ...

provides a simple calculator using a postfix notation. A call consists of
two numbers and an operation optionally followed by an arbitrary number
of number-operation pairs.

NUM1, NUM2 and NUM are treated as integer numbers.

NUM is treated in the same way as NUM2 whereas NUM1 in this case is the
result of the previous operation.

OP is one of:
  ADD - adds NUM1 and NUM2
  SUB - subtracts NUM2 from NUM1
  MUL - multiplies NUM1 and NUM2
  DIV - divides NUM1 by NUM2 and returns the integer result
  MOD - divides NUM1 by NUM2 and returns the integer remainder
  EXP - raises NUM1 to the power of NUM2

At the end of a successful call the history of all intermediate calculations
is printed out to stderr.
```

Eine Angabe von weiteren Parametern bei der Hilfeanfrage ist nicht erlaubt und somit als Fehlerfall zu werten.

Bedenkt bei der Implementierung, dass die Hilfe ein Bestandteil eures Skriptes ist – es ist also weder sinnvoll noch erlaubt den Hilfetext aus einer extra Datei zu laden.

Berechnung

Im normalen Betrieb, wenn nicht die Hilfe angefordert wird, soll das Programm alle per Parameter definierten Berechnungen durchführen. Dabei sollen die eingegebenen Zahlen nur Ganzzahlen sein. Eine zusätzliche Einschränkung besteht für die beiden Divisionsoperationen (DIV und MOD), hier muss der Divisor eine Ganzzahlen exkl. 0 sein, da eine Division durch 0 nicht zugelassen ist und die Exponentiation, bei der der Exponent lediglich eine positive Ganzzahl oder Null sein darf.

Ihr könnt davon ausgehen, dass euer Programm nur mit gültigen Zahlen aufgerufen wird - eine Überprüfung, ob die Parameter korrekte Zahlen sind, ist also nicht notwendig.

Die Eingabe erfolgt in Postfix-Notation und kann beliebig lang sein. Um eine Berechnung durchführen zu können, müssen mindestens zwei Zahlen und ein Operator angegeben werden (in der Reihenfolge "Zahl Zahl Operator"). Der Operator wird dann auf die beiden eingelesenen Zahlen angewendet.

Beispiel:

```
1 2 ADD
Ergebnis: 3
```

Optional folgen dann weitere Zahl-Operator-Paare. Dabei wird die weitere Berechnung stets aus dem Ergebnis des vorhergehenden Schrittes und der aktuellen Zahl mit der aktuellen Operation berechnet.

Beispiel:

```
1 2 ADD 4 ADD 2 ADD
Ergebnis: 9
```

Geschachtelte Berechnung nach dem Schema "1 2 ADD 3 4 ADD DIV", also $(1+2)/(3+4)$, sind mit der für diese Aufgabe vereinfachten Postfix-Notation **nicht** möglich.

Berechnungshistorie

Zusätzlich zur Berechnung eines Ausdrucks gibt das Programm im Falle einer erfolgreichen Auswertung eine Historie mit allen durchgeführten Berechnungen auf `stderr` (zusätzlich zum Berechnungsergebnis auf `stdout`) aus. Dabei steht jede durchgeführte Berechnung in einer eigenen Zeile und ist mit dem Präfix ">" versehen, danach folgt in der selben Zeile die ausgeführte Berechnung in Prefix-Notation. Dabei wird zunächst der Operator gefolgt von den beiden verrechneten Zahlen ausgegeben, die Teile sind untereinander durch Leerzeichen getrennt, am Ende der Zeile befindet sich kein Leerzeichen.

Beispiel:

```
1 2 ADD
Ausgabe: > ADD 1 2
```

Dies funktioniert analog auch über mehrere Verschachtelungsebenen:

Beispiel:

```
1 2 ADD 4 ADD -2 ADD
Ausgabe: > ADD 1 2
        > ADD 3 4
        > ADD 7 -2
```

Ausgabe

Im Falle einer erfolgreichen Berechnung wird das Ergebnis dieses Programmdurchlaufs auf `stdout` ausgegeben und die Berechnungshistorie auf `stderr`.

Euer Programm darf bei einem gültigen Aufruf während der Verarbeitung der Parameter keine weiteren Ausgaben machen. Es ist im Erfolgsfall mit einem *exit code* von 0 (siehe Hinweise) zu beenden.

Solltet ihr zum Debuggen zusätzliche Informationen ausgeben wollen, denkt daran diese vor dem Servertest auszukommentieren oder zu entfernen.

Fehlerbehandlung

Eine Fehlerbehandlung ist diesmal notwendig. Im Falle eines fehlerhaften Aufrufs soll sich euer Skript gemäß der gängigen Konvention mit einem *exit code* von > 0 beenden. Zudem sollen eine einzeilige Fehlermeldung gefolgt von dem Hilfetext zusammen auf `stderr` ausgegeben werden. Die einzeilige Fehlermeldung muss mit der Zeichenfolge "ERROR:" beginnen und anschließend eine aussagekräftige Fehlerbeschreibung enthalten.

Überläufe bei den Berechnungen müssen nicht abgefangen werden, zudem muss auch keine Überprüfung, ob gültige Zahlen eingegeben wurden, stattfinden. Abgesehen davon sind alle möglichen Fehlerfälle (unbekannte Operatoren, falsche Aufrufsyntax, Division durch Null, Exponent negativ) abzufangen und zu behandeln.

Hinweis: Wird nur die Hilfe angefordert (-h bzw. --help), soll der *exit code* des Skripts natürlich auch 0 sein, da dies ein gültiger Aufruf ist. In diesem Fall ist der Hilfetext auch auf `stdout` auszugeben!

Aufrufbeispiele

Beispiel 1:

```
./pfcalc.sh 1 2 ADD
```

ergibt auf `stdout`:

```
3
```

und auf `stderr`:

```
> ADD 1 2
```

Beispiel 2:

```
./pfcalc.sh -1 -2 ADD
```

ergibt auf `stdout`:

```
-3
```

und auf `stderr`:

```
> ADD -1 -2
```

Beispiel 3:

```
./pfcalc.sh 1 2 ADD 2 SUB 7 MUL 2 EXP
```

ergibt auf `stdout`:

```
49
```

und auf `stderr`:

```
> ADD 1 2
> SUB 3 2
> MUL 1 7
> EXP 7 2
```

Modularisierung

Verwendet in eurem Skript mindestens eine sinnvolle **Funktion**. Nutzt in der Funktion / den Funktionen gegebenenfalls Parameter als Eingabe und `stdout` um das / die Ergebnis(se) zurückzugeben.

Globale Variablen sind in Funktionen nicht erlaubt! Kennzeichnet gegebenenfalls benötigte lokale Variablen daher mit dem Schlüsselwort: `local` (siehe auch `man sh`).

sh vs. bash

In dieser Aufgabe ist die Benutzung der bash als Interpreter ausdrücklich untersagt, da wir dieses Skript möglichst POSIX-konform und portabel gestalten wollen.

Umfang der Lösung

Diese Aufgabe ist mit weniger als 150 Zeilen Code lösbar (ohne Kommentare und Leerzeilen). Lösungen, welche mehr als 250 Zeilen (ohne Kommentare und Leerzeilen) umfassen, werden wir daher nicht akzeptieren.

Sollte eure Lösung also entsprechend umfangreich sein, überprüft diese auf weitere Vereinfachungsmöglichkeiten.

Formatierung und Kommentare

Eine sinnvolle Formatierung und Einrückung des Quelltextes wird vorausgesetzt. Schlecht formatierte und somit unlesbare Programme werden wir nachbessern lassen.

Zudem erwarten wir, dass eure Lösung ausreichend kommentiert ist. Dazu gehört mindestens ein Modulkopf, welcher die Funktion des Skripts kurz beschreibt und den / die Autor(en) angibt; Funktionsköpfe, welche die Eingabeparameter und die Ausgabe der jeweiligen Funktion beschreiben (dies ist bei der Shell-Programmierung besonders wichtig, da hier die Parameter nicht auf den ersten Blick ersichtlich sind), und entsprechende Kommentare, so dass ein anderer Programmierer sich in eurem Quelltext zurechtfinden kann. Wie auch bei der mangelnden Formatierung werden wir Mängel in der Kommentierung nachbessern lassen.

Benennung

Nennt euer Skript "pfcalc.sh", damit die automatischen Tests laufen und achtet darauf, dass es auch als ausführbare Datei eingecheckt ist.

Erlaubte Tools

In dieser Aufgabe sind weiterhin fortgeschrittene Tools wie etwa perl, sed, awk und ruby nicht erlaubt und auch nicht vonnöten.

Testen

Auch bei dieser Aufgabe werden wir eure Lösungen automatisiert mit dem Testserver überprüfen. Ihr könnt diesen auch nach wie vor selbst nutzen um sicherzustellen, dass ihr alle Anforderungen richtig gelesen und umgesetzt habt. Beachtet hierzu die [Allgemeinen Hinweise zum Testen](#).

Um euch den Einstieg in diese Aufgabe und das etwas komplexere Testen mit Arnold zu erleichtern, geben wir eine kleine [Beispieltestbench](#) mit.

Schreibt euch aber unbedingt auch eigene Tests um sicherzustellen, dass ihr alle in der Aufgabenstellung beschriebenen Anforderungen auch umgesetzt habt. Bei der Abnahme werden wir prüfen, ob ihr sinnvolle Tests geschrieben habt und diese erklären könnt. Außerdem erwarten wir, dass ihr selbst testet (und dies gegebenenfalls auch nachweisen könnt) bevor wir uns bei Problemen eure Lösung vorab anschauen um mit euch auf Fehlersuche zu gehen.

Bei der Fehlersuche bietet es sich zudem an, die aus der Vorlesung bekannten Debug-Optionen der Shell zu nutzen.

Hinweis zum Servertest

Der Servertest wird in einem Großteil der Testfälle die Ausgaben auf stdout und stderr unabhängig voneinander testen. D.h. es ist möglich Teilpunkte zu erzielen, wenn nur die Berechnung oder nur die Ausgabe der Berechnungshistorie korrekt funktioniert.

Beachtet bitte, dass genau wie beim lokalen Testen mit arnold und diff auch jeglicher Whitespace relevant ist. Gerade "trailing spaces" können dazu führen, dass ein Test fehlschlägt, obwohl augenscheinlich das richtige Ergebnis ausgegeben wird.

Nicht-funktionale Mängel


Nachbesserungsfähige nicht-funktionale Mängel:

Nachbesserungsgrund	Abzug Prozentpunkte bei Nichterfüllung
Repository / Ordnerstruktur schlecht gepflegt	5
Interpreterangabe fehlt	5
Uneinheitliche Formatierung oder Sprachwahl für Bezeichner oder Kommentierung	5
Unangemessene Bezeichner (z.B. hinsichtlich ihrer Länge oder Aussagekraft)	5
Mangelnde Codehygiene (z.B. toter Code)	5
Mangelhafte Formatierung / schlecht lesbarer Code (z.B. Einrückung, Tabs und Leerzeichen gemischt)	10
Mangelhafte Dokumentation (z.B. Funktionsköpfe / Inline-Komentierung)	10
Unnötige Codeverdopplung	10
Verwendung von unstrukturierter Programmierung (z.B. unangemessen viele returns oder exits)	10
Mangelnde Effizienz / umständliche Lösung (z.B. unnötige temporäre Dateien / Variablen)	20
Nicht vorhandene Testfälle	20

Nachbesserungsgrund	Abzug Prozentpunkte bei Nichterfüllung
Umständliche / zu lange Lösung	30
Fehlende / Unpassende (unzureichende / übermäßige) Modularisierung	30
Nicht aufgabenkonforme Implementierung	30

Downloads

Das [Archiv](#) zu dieser Aufgabe enthält einen kleinen Beispielttest für Arnold sowie den Hilfetext als Kopiervorlage und Vergleichsdatei zum Testen.

Die [Folien](#)  zur Aufgabenvorstellung (die ja in diesem Semester leider nicht stattfinden kann), geben einen groben Überblick über die Aufgabenstellung und sollen dem besseren Einfinden in die Aufgabenstellung dienen. Sie sind nicht als weitere Spezifikation gedacht und auch nicht so zu verstehen - im Zweifelsfall gelten immer die Angaben, so wie sie hier in der Aufgabenstellung stehen.

Beachtet bitte unbedingt auch das [FAQ](#) zu den Abläufen in den Übungen.

Viel Erfolg!

letzte Änderung: 04.06.2020 13:20