– Lösung zur Praktikumsaufgabe 6 –

Thema: *Numerische Berechnungen im Shellskript, Brace Expansion*

1. Man kann den Test auf ein gültiges Operatorsymbol entweder mit einer Folge von `if`-Klauseln vornehmen, oder, wie in der hier angegebenen Lösung, elegant mit Hilfe der `case`-Anweisung der Bash.

Listing 1: Ein simpler Rechner

```
#!/bin/sh

if [ $# -ne 3 ]; then
    echo "Usage: $0 <op1> [ + | - | * | / ] <op2>"
    exit 1
fi

case "$2" in
    '+' | '-' | '*' | '/' )
  echo $((result = $1 $2 $3))
  exit 0;;
esac

echo Operation '$2' not allowed. Exit.
exit 2
```

2. a) Eine iterative Lösung könnte folgendermaßen aussehen:

Listing 2: Shellskript zur iterativen Berechnung der Fibonacci-Reihe

```
1  #!/bin/bash
2  # fib-iter.sh
3  # prints the first $iterations Fibonacci numbers according to
4  #   fib(0) = 1
5  #   fib(1) = 1
6  #   fib(n) = fib(n-1) + fib(n-2)   n >= 2
7
8  # as can be seen, the numbers wrap around at fib(92)
9
10 iterations=100
11 # initial setup
12 a=0
13 b=1
14 echo fib\(0\) = $a
15 echo fib\(1\) = $b
16
17 # main loop
18 for ((count=2; count < $iterations; count++))
19 do
20   let sum=a+b
21   echo fib\($count\) = $sum
22   let a=b
23   ((b=sum))
```

```
24 done
25
26 # epilogue
27 exit 0
```

Diskussion:

- Der gültige Zahlenbereich bei Integer-Variablen ist in der Bash ganz schön groß. Für fib(92) erhalten wir erstmalig einen *Integer-Overflow*. Dies hängt damit zusammen, dass die Bash mit der Integerbreite arbeitet, die dem Prozessor zur Verfügung steht, auf allen einigermaßen modernen Rechnern also 64 Bit.
- Eine Bereichsüberschreitung wird offenbar durch die Shell nicht erkannt.
- Zeilen 22 und 23 zeigen die beiden semantisch äquivalenten Möglichkeiten numerischer Ausdrücke (mit `let` oder doppelt geklammert).

b)* Eine gegebene Fibonacci-Zahl ermittelt man besser rekursiv:

Listing 3: Rekursive Ermittlung einer Fibonacci-Zahl

```
#!/bin/bash

# range of allowed parameters
MINFIB=0
MAXFIB=1000

function fib ()
{
    local ret

    # trivial cases
    if [ $1 -eq 0 ]; then
        return 0;
    elif [ $1 -eq 1 ]; then
        return 1
    fi

    # recursive calls
    fib $(($1-1))
    let ret=$?
    fib $(($1-2))
    return $((ret+$?))
}

# some parameter testing
if [ $# -ne 1 ];  then
    echo Usage: $0 \<number\>
    exit 1
fi
# Is $1 not an integer?
if [ $1 -eq $1 ] 2>/dev/null ; then
    :
```

```
else
    echo Integer expected.
    exit 2
fi
if [ $1 -lt 0 ] || [ $1 -gt $MAXFIB ]; then
    echo Parameter outside allowed range \($MINFIB\;$MAXFIB\).
    exit 3
fi

# main
fib $1
echo fib\($1\)=$?

#epilogue
exit 0
```

Fehlerbedingungen, die getestet werden:

i. Wurde die richtige Anzahl (hier: 1) Parameter übergeben?

ii. Wurde ein Integer als Parameter übergeben? Dieser Test ist programmiertechnisch nichttrivial, da es kein einfaches Sprachmittel innerhalb der bash dafür gibt.

    Zunächst wird $1 numerisch mit sich selbst verglichen. Handelt es sich um *keinen* numerischen Parameter (also eine Zeichenkette), dann schreibt die Bash eine Fehlermeldung nach `stderr`, die entsprechend nach `/dev/null` geleitet werden muss, und `test` liefert false zurück, was in den else-Zweig führt, in dem das Skript abgebrochen wird. Anderenfalls ist die Bedingung natürlich wahr; wir haben in diesem Fall aber nichts zu tun, daher das leere (`:`) Statement.

iii. Ist der Parameter innerhalb des erlaubten Wertebereiches? (Beispielsweise sind Fibonacci-Zahlen für negative Argumente nicht definiert)

Funktioniert damit unser Skript fehlerfrei? Protokollieren wir einmal einige Aufrufe:

```
~> ./fib-rek.sh Jehova
Integer expected.
~> ./fib-rek.sh 0
fib(0)=1
~> ./fib-rek.sh 5
fib(5)=8
~> ./fib-rek.sh .1
Integer expected.
~> ./fib-rek.sh -1
Parameter outside allowed range (0;1000).
~> ./fib-rek.sh 12
fib(12)=233
~> ./fib-rek.sh 13
fib(13)=121
```

Wie das? Die Reihe der Fibonacci-Zahlen ist doch streng monoton wachsend!

Die Antwort liegt im benutzten Mechanismus für die Übergabe von Resultaten der Funktion an die rufende Umgebung. Es kann mittels `return` nämlich nur ein Byte übergeben werden, was den Wertebereich [0;255] besitzt. Schlimmer noch, eine Wertebereichsüberschreitung (wie im obigen Protokoll) wird weder erkannt noch gemeldet. Es liegt also allein in der Verantwortung des Programmierers, auf den Resultatwertebereich zu achten bzw. gleich einen anderen Mechanismus zu verwenden.

Eine Variante wäre, globale Variablen zu nutzen, was aber im Interesse einer sauberen Programmstruktur besser unterbleiben sollte. Das folgende Listing demonstriert stattdessen die Kommunikation über `stdout`; die Plausibilitätstests für `$1` wurden der Übersichtlichkeit halber fortgelassen:

```
...
function fib ()
{
    # trivial case
    if [ $1 -eq 0 ] || [ $1 -eq 1 ]; then
        echo 1
        return
    fi

    # recursive calls
    echo $((`fib $(($1-1))`+`fib $(($1-2))`))
}
...
# main
echo fib\($1\)= `fib $1`

#epilogue
exit 0
```

Die Kodierung ist kompakter, wir benötigen auch keine lokale Variable mehr; die Übersichtlichkeit hat natürlich etwas gelitten.

3.* Zunächst das Shellskript:

Listing 4: Shellskript zur Berechnung des Siebes des Eratosthenes

```
#!/bin/bash
## Eratosthenes' Sieve

# upper limit to test for prime numbers
LIMIT=1000000

# init the array with 'false'
for (( c=0; $c < $LIMIT; c++ )); do a[$c]=1; done

let i=2
while let isquared=$i*$i; [ $isquared -le $LIMIT ]
do
```

```
    if [ ${a[$i]} -eq 1 ]
    # i is prime, eliminate its multiples, starting with i*i
    then
        for (( d=$i*$i; $d < $LIMIT; d=$d+$i )); do a[$d]=0; done
    fi
    let i++
done

# output results
echo -n 2
for (( c=3; $c < $LIMIT; c++ ))
do
    if [ ${a[$c]} -eq 1 ] ; then echo -n , $c ; fi
done

# epilogue
echo
exit 0
```

Und nun das C-Programm:

Listing 5: C-Implementierung des *Sieb des Eratosthenes*

```
/*
   Sieve of Eratosthenes
*/
#include <stdio.h>
#include <stdlib.h>

/* upper limit to test for prime numbers */
#define LIMIT 1000000

int main(int argc, char* argv[])
{
  int a[LIMIT+1];
  unsigned long i, d;

  /* init the array with 'false' */
  for (i=0; i < LIMIT; i++) {
    a[i] = 0;
  }
  i = 2;
  while (i*i <= LIMIT) {
    if (a[i] == 0) {
      /* i is prime, eliminate its multiples, starting with i*i */
      for (d = i*i; d <= LIMIT; d += i) {
        a[d] = 1;
      }
    }
    i++;
  }

  /* output results */
  printf(" 2");
  for (i=3; i <= LIMIT; i++) {
```

```
      if (a[i] == 0) {
        printf(", %li", i);
      }
    }
    printf("\n");
    exit(EXIT_SUCCESS);
}
```

Die folgende Tabelle enthält die mittels `time` ermittelten Laufzeiten in Abhängigkeit von N für ein System mit Intel Core 2 Duo 6400 mit 2.13 MHz Taktfrequenz und 2 GiB RAM unter Debian Linux, GCC 4.1.2.

LIMIT	100	1000	10 000	100 000	1 000 000
Shellskript	14 ms	91 ms	1.45 s	117 s	224 min
C-Programm	1 ms	2 ms	5 ms	32 ms	372 ms

Tabelle 1: Vergleich der Abarbeitungszeiten für das *Sieb des Eratosthenes* implementiert in C und als Shellskript.

Der Geschwindigkeitsvorteil des C-Programmes wächst mit der Größe des Problems und beläuft sich für $N = 10^6$ auf den Faktor 36000! Ein deutlicher Hinweis, Zahlenspielereien nicht in der Bash zu programmieren, sondern lieber einen Compiler einzusetzen.

4. a) aller Zeichenketten, die aus vier Kleinbuchstaben bestehen:

```
echo {a..z}{a..z}{a..z}{a..z}
```

b) aller Zeichenketten, die aus 1-3 Buchstaben bestehen

```
echo {{a..z},{A..Z}} {{a..z},{A..Z}}{{a..z},{A..Z}} \
     {{a..z},{A..Z}}{{a..z},{A..Z}}{{a..z},{A..Z}}
```

c) die Rechner isys101 . . . isys121, isys6, isys8, isys10 . . . isys29:

```
echo isys{6,8} isys{1,2}{0..9} isys1{0,1}{1..9} isys1 ↩
   {10,2{0,1}}
echo isys{6,8} isys{101..121} isys{10..29}
```

d)* die (abstrakten) Zeichenketten bb bbbb aa aabb aabbbb aaaa aaaabb aaaabbbb

```
echo {,aa,aaaa}{,bb,bbbb}
```

e)

```
bards, barns, barks, beds
```