WBE-Praktikum 2

JavaScript Grundlagen

Aufgabe 1: Node REPL

Öffnen Sie Node.js durch Eingabe von node im Terminal. Dadurch wird Node.js im interaktiven Modus gestartet, das heisst jeder eingegebene Ausdruck wird direkt evaluiert und das Ergebnis ausgegeben. Sie befinden sich in der REPL (Read Eval Print Loop).

Um sich mit der Node-REPL vertraut zu machen, geben Sie am besten ein paar Anweisungen und Ausdrücke ein:[1]

```
> .help
> let x = 7
> x * x - x
> _
> _ % 10
> 0.1 + 0.2
> 1 / 0
> 2n ** 128n
> typof 0.5
> typeof "wbe"
> "wbe"[0]
> `half of 100 is ${100 / 2}`
> (x => x * x)(7)
> typeof(x => x * x)
> console.log(2 ** 6 > 100)
> console.clear()
```

In den Slides hat es noch einige Anregungen, was Sie noch ausprobieren könnten.

Dine Funktion *console.log* (eigentlich Methode *log* des *console*-Objekts) kann für Ausgaben eines Scripts verwendet werden. Hier einige weitere Funktionen der Console API: https://nodejs.org/dist/latest-v12.x/docs/api/console.html

[1] Das ">" ist das Prompt-Zeichen von Node.js, das geben Sie natürlich nicht mit ein ☺

Aufgabe 2: Script starten

Betrachten Sie das Script *hello-world.js* – es enthält einen kleinen Webserver, der auf jede Anfrage den Text `'Hello, World!\n'` liefert. Anstatt den Text einfach mit *console.log* auf der Konsole auszugeben, ist das sozusagen das etwas anspruchsvollere *Hello World* von Node.js.

```
const http = require('http')
const hostname = '127.0.0.1'
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello, World!\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

Starten Sie das Script mit

```
node hello-world.js
```

Sie sollten die Seite nun unter folgender Adresse im Browser öffnen können:

```
http://127.0.0.1:3000/
```

Hinweis: Der Aufruf *http.createServer* erhält als Argument eine Funktion übergeben. Ebenso wird dem *server.listen* als drittes Argument eine Funktion übergeben. In JavaScript ist es durchaus üblich, Funktionen als Argumente zu übergeben.

Wichtiger Hinweis: Wenn Sie mit Servern auf Ihrem Notebook experimentieren ist eine korrekt funktionierende Firewall wichtig. Der betreffende Port sollte nicht von aussen zugänglich sein.

Aufgabe 3: Primzahlfunktion

Schreiben Sie eine Funktion *isPrime*, welche feststellt, ob eine übergebene Zahl eine Primzahl ist:

```
> isPrime(15)
false
> isPrime(17)
true
```

Spielt es eine Rolle, ob sie *true* bzw. *false* oder etwa 1 bzw. 0 (wie man es in C machen würde) zurückgeben? Schreiben Sie die Funktion in einer JavaScript-Datei, inklusive einiger mit *console.log* ausgegebener Test-Cases.

Aufgabe 4: Fibonacci-Funktion

Die Funktion, welche die n-te Fibonacci-Zahl berechnet, ist folgendermassen definiert.

$$F_n = \begin{cases} 0 & \text{für} \quad n = 0 \\ 1 & \text{für} \quad n = 1 \\ F_{n-1} + F_{n-2} & \text{für} \quad n > 1. \end{cases}$$

Mit dieser Funktion sollen nun ein paar Experimente gemacht werden.

a) Implementieren Sie eine Funktion *fibo*, welche die n-te Fibonacci-Zahl rekursiv gemäss obiger Definition berechnet. Wenn man die Funktion rekursiv umsetzt, lässt sich die Definition sehr einfach in Programmcode umsetzen. Da die meisten *fibo*-Aufrufe zwei weitere Funktionsaufrufe nach sich ziehen ist klar, dass der Aufrufstack aber für grössere Zahlen schnell wächst und ein Stack-Overflow resultieren kann. Fügen Sie Ihrem Script ein paar Testaufrufe (Ausgabe mit *console.log*) hinzu.

   Zum Vergleich könnten wir die n-te Fibonacci-Zahl auch iterativ implementieren (fakultative Übung). Wir wollen hier aber noch einen anderen Weg gehen:

b) Die n-te Fibonacci-Zahl ist die nächste ganze Zahl zu diesem Funktionswert:

   $f(n) = G^n/\sqrt{5}$ mit $G = (1 + \sqrt{5})/2$

   Implementieren Sie die Fibonacci-Funktion mit Hilfe dieser Formel und vergleichen Sie die Ergebnisse mit der Variante aus Teil a).

   Tipp: Verwenden Sie die Funktionen *Math.round* und *Math.sqrt*, mehr Informationen hier: https://devdocs.io/javascript/global_objects/math

   Wissen Sie, unter welchem Namen die hier verwendete Konstante G auch noch bekannt ist?

c) Vergleichen Sie den Zeitaufwand für die Berechnung der Fibonacci-Zahl mit den beiden Funktionen für n = 30. Dazu können Sie den Timer der Console API verwenden:

```
console.time("fibo")       /* Start der Zeitmessung */
...                        /* Berechnung durchführen, ohne I/O */
console.timeEnd("fibo")    /* Ende der Zeitmessung, Ausgabe der Zeit */
...                        /* Ergebnis ausgeben */
```

d) Wenn wir die iterative Variante hinzunehmen, haben wir nun drei Varianten, die n-te Fibonacci-Zahl zu implementieren, jede mit bestimmten Stärken und Schwächen:
– rekursiv
– iterativ
– mit der Näherungsformel

Mit *BigInt* (neu in ES2020) können wir beliebig grosse Ganzzahlen exakt darstellen. Wie berechnen wir am besten die n-te Fibonacci-Zahl für grössere Zahlen, also wenn das Ergebnis ausserhalb der Ganzzahldarstellung von *Number* ist? Implementierung fakultativ.

Aufgabe 5: Etwas Mathematik (fakultativ)

Die Formel in Aufgabe 4 b) hat die n-te Fibonacci-Zahl nur näherungsweise bestimmt. Wir können sie aber auch exakt angeben:

$$f(n) = (G^n - H^n) / \sqrt{5} \qquad \text{mit} \quad G = (1 + \sqrt{5})/2 \qquad \text{und} \quad H = (1 - \sqrt{5})/2$$

Warum kann man den zweiten Teil mit H für die näherungsweise Berechnung der Fibonacci-Zahl weglassen und man kommt trotzdem zum richtigen Ergebnis?

Noch etwas für die mathematisch Interessierten:

Beweisen Sie, dass diese Formel die n-te Fibonacci-Zahl gemäss der Definition in Aufgabe 4 bestimmt. Dazu müssen Sie zeigen, dass die Formel für n=0 und n=1 stimmt, und, angenommen die Formel stimmt für n und n+1, dass sie dann auch für n+2 stimmt.

Bemerkung zum Schluss

Hier noch ein Zitat aus Eloquent JavaScript zum Thema Effizienz:

«Worrying about efficiency can be a distraction. It's yet another factor that complicates program design, and when you're doing something that's already difficult, that extra thing to worry about can be paralyzing.»

«Therefore, always start by writing something that's correct and easy to understand. If you're worried that it's too slow—which it usually isn't since most code simply isn't executed often enough to take any significant amount of time—you can measure afterward and improve it if necessary.»

https://eloquentjavascript.net/