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/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.

Noch ein Hinweis: Verwenden Sie Variablen (*let*) nur dann, wenn sich ein Wert tatsächlich ändert. Ansonsten sind Konstanten (*const*) vorzuziehen.

Und schliesslich noch ein 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: Zahlen potenzieren (Abgabe)

Selbstverständlich ist es für Sie keine grosse Herausforderung, eine *power*-Funktion zu implementieren. Hier geht es vor allem darum, sich mit den speziellen Eigenschaften von JavaScript im Vergleich zu anderen Sprachen vertraut zu machen. Im Unterricht haben wir bereits eine Funktion *power* angesehen, welche für ganze Zahlen ≥0 die Potenz b^n mit Hilfe einer Schleife berechnet. Da wir n Schleifendurchgänge haben, ist die Laufzeit dieser Variante linear (O(n)).

Nun sehen wir uns die Möglichkeiten an, die Funktion rekursiv zu implementieren (natürlich ohne den seit ES 2016 vordefinierten **-Operator zu verwenden):

a) Die Funktion kann wie folgt rekursiv spezifiziert werden:

$b^n = b \cdot b^{n-1}$
$b^0 = 1$

Das wäre einfach. Es soll aber gleich noch eine Optimierung eingebaut werden, um durch mehrfaches Quadrieren schneller zum Ergebnis zu kommen, basierend auf folgendem Zusammenhang:

$b^n = (b^{n/2})^2$ wenn n gerade ist
$b^n = b \cdot b^{n-1}$ wenn n ungerade ist
$b^0 = 1$

Implementieren Sie *power* nun auf diese Weise. Um eine logarithmische Laufzeit (O(log n)) zu erhalten, sollten Sie beim Quadrieren nicht zwei rekursive Aufrufe machen… ☺

b) Wenn die übergebenen Argumente nicht den Erwartungen entsprechen, können verschiedene Probleme auftreten, da JavaScript mit seinem dynamischen Typenkonzept keine Überprüfung der Parametertypen vornimmt. Ergänzen Sie Ihr Script um eine Funktion *assert* und verwenden Sie diese in der Funktion *power*, so dass Fehler in den übergebenen Argumenten *AssertionErrors* auslösen.[2] Mit `Number.isInteger(n)` können Sie überprüfen, ob n eine ganze Zahl ist.

```
function assert(condition, message) {
    if (!condition) throw new Error(message || "Assertion failed")
}
```

Aufruf:

```
> power(3, 3)
27
> power(3, 3.5)    // !! AssertionError
> power(3, -3)     // !! AssertionError
```

[2] Node.js enthält ein Modul *assert* mit einer gleichnamigen Funktion. Um sicherzustellen, dass es keine Probleme mit dem Abgabeserver gibt, auf dessen *node*-Implementierung *assert* möglicherweise nicht zur Verfügung steht, soll hier eine eigene Funktion verwendet werden. Ausserdem ist am Beispiel zu sehen, wie in JavaScript ein Fehler geworfen wird und mit || ein Default-Wert gesetzt werden kann.

c) Erweitern Sie Ihre *power*-Funktion so: wenn beide Argumente vom Typ *"bigint"* sind, soll mit beliebig grossen Zahlen gerechnet werden können und *"bigint"* zurückgegeben werden:

```
> power(3, 3)
27
> power(3n, 3n)
27n
> power(12345678901234567890123456789 0n,2n)
152415787532388367504953515625361987875019051998750190521 00n
```

Abgabe

Geben Sie die letzte Version der Funktion *power* ab.

Abgabeserver:	https://radar.zhaw.ch/python/UploadAndCheck.html
Name Praktikum:	WBE1
Dateiname:	power.js
Funktionsname:	power
Export im Script:	`module.exports = { power }`

Hinweis zum Abgabeserver

Die Zusammenarbeit des Jasmine-Test-Tools und Node.js auf dem Linux-Server hat sich als nicht ganz problemlos herausgestellt. Daher läuft für die Tests momentan keine aktuelle Node.js-Version und es muss davon ausgegangen werden, dass die neueren JavaScript-Features nicht unterstützt werden, zum Beispiel der ??-Operator.

Aufgabe 4: Fibonacci-Funktion einmal anders (fakultativ)

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

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

Diese Funktion haben Sie sicher schon in anderen Fächern implementiert. Wenn man die Funktion rekursiv umsetzt, lässt sich die Definition sehr einfach in Programmcode umsetzen. Da die meisten Aufrufe zwei weitere Funktionsaufrufe nach sich ziehen ist klar, dass der Aufrufstack für grössere Zahlen schnell wächst und ein Stack-Overflow resultieren kann. Auch eine iterative Implementierung mit linearer Laufzeit ist relativ einfach möglich. Wenn Sie noch keine Fibonacci-Funktion in JavaScript implementiert haben, ist das natürlich eine gute Übung…

Hier soll aber ein anderer Ansatz verfolgt werden. Die n-te Fibonacci-Zahl ist die nächste ganze Zahl zu diesem Funktionswert:

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

Implementieren Sie eine Fibonacci-Funktion mit Hilfe dieser Formel.
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?

Soweit so gut, wir können aber noch tiefer einsteigen… Die Formel oben hat die n-te Fibonacci-Zahl nur näherungsweise bestimmt. Wir können sie aber auch exakt angeben:

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

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

Expertenaufgabe für die mathematisch Interessierten

Beweisen Sie, dass diese Formel die n-te Fibonacci-Zahl gemäss der Definition am Anfang der Aufgabe 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.

Hinweis zur Zeitmessung

Um verschiedene Implementierungen einer Funktion (etwa der Fibonacci-Funktion) zu vergleichen, kann es hilfreich sein, die Ausführungszeit messen zu können. Dazu kann zum Beispiel der Timer der Console API verwendet werden:

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

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/