WBE-Praktikum 4

Funktionen

Aufgabe 1: Node REPL

Zunächst wieder ein paar Versuche auf der Node REPL:

```
> let factorial = (n) => n<=1 ? 1 : n * factorial(n-1)
> factorial(20)
> factorial.doc = "Fakultätsfunktion"
> factorial.created = "26.08.2021"
> factorial
> factorial(5, 10, 20)
> let param = (a, b, c) => [a, b, c]
> param(1)
> param(1, 2, 3, 4)
> let data = [10, 11, 12, 13, 14]
> param(42, ...data)
> let divmod = (m, n) => [Math.floor(m/n), m%n]
> let [div, rest] = divmod(17, 7)
> div + rest
> const prefix = (pre) => (text) => pre + text
> const mod = prefix("mod_")
> mod("getData")
```

Aufgabe 2: Funktion definieren

Schreiben Sie die Fakultätsfunktion aus Aufgabe 1 so um, dass die Fakultät iterativ bestimmt wird. Passen Sie das Ergebnis zudem so an, dass die Funktion auch mit *BigInt* arbeitet:

```
> factorial(10)
3628800
> factorial(50n)
30414093201713378043612608166064768844377641568960512000000000000n
```

Aufgabe 3: Schriftsysteme

In der Datei *scripts.js* finden Sie eine Sammlung von Informationen zu Schriftsystemen, welche in verschiedenen Regionen der Erde im Einsatz sind oder waren. Betrachten Sie den Aufbau der Daten. Welche Rollen spielen Objekte und Arrays bei der Repräsentation der Daten?

Sie können die Daten mit *require* in ein eigenes Script importieren. Probieren Sie es aus – angenommen *scripts.js* liegt im gleichen Verzeichnis wie Ihr Script:

```
require('./scripts.js')
console.log(SCRIPTS[0])
```

a. Implementieren Sie eine Funktion *oldAndLiving*, welche die Sammlung der Scripts als Parameter bekommt und die Namen aller Schriftsysteme, für welche *year < 0* und *living true* ist, als Array zurückgibt:

   ```
   console.log( oldAndLiving(SCRIPTS) )
   // [ 'Ethiopic', 'Greek', ...]
   ```

   Am besten setzen Sie die Funktion zunächst iterativ um (*for…of*-Schleife). Sobald das funktioniert, probieren Sie noch eine Variante mit *filter* und *map*.

b. Welche Zeichencodes in Unicode (gleich mehr dazu) die verschiedenen Schriftsysteme umfassen, ist im Attribut *ranges* angegeben: es ist ein Array von Codebereichen, jeder Bereich ist ebenfalls als Array angegeben mit unterer Grenze (inklusive) und oberer Grenze (exklusive).

   Implementieren Sie eine Funktion *numberOfCodes*, welche ein Schriftsystem als Parameter erhält und die Anzahl der für diese Schrift reservierten Zeichencodes liefert:

   ```
   console.log( numberOfCodes(SCRIPTS[3]) )
   // 1280
   ```

   Am besten setzen Sie auch diese Funktion zunächst iterativ um.

Ein etwas „funktionalerer" Ansatz: in der funktionalen Programmierung versucht man, Schleifen und Zuweisungen zu vermeiden. Sie können häufig einfach durch Funktionsaufrufe ersetzt werden. Da der Einsatz von Funktionen wie *reduce* etwas gewöhnungsbedürftig ist, soll hier eine entsprechende Lösung für Aufgabenteil b) angegeben werden. Der Inhalt der Funktion *numberOfCodes* schrumpft nun zu einem einzigen Funktionsaufruf, keine Schleife, keine Zuweisung mehr:

```
const numberOfCodes = ({ranges}) =>
  ranges.reduce((curr, [from, to]) => curr+to-from, 0)
```

Destrukturieren von Parametern: Im Beispiel wird der Funktion *numberOfCodes* ein Objekt als Argument übergeben, von Interesse ist hier aber nur das Attribut *ranges*. Daher wird dieses direkt in der Parameterliste angegeben. Ebenso in der Funktion mit Parameterliste *(curr, [from, to])*: anstelle

den Bereich als *range*-Parameter anzugeben und dann mit *range[0]* und *range[1]* zuzugreifen, werden die Grenzen hier direkt in Variablen *from* und *to* gefüllt.

Einschub: Unicode, UTF-8

Um besser zu verstehen, wozu diese Bereiche von Zeichencodes in den verschiedenen Schriftsystemen dienen und welche Rolle UTF-8 als Zeichencodierung spielt, soll dies in einem kleinen Exkurs hier erläutert werden. ASCII stellt nur 128 Codes zur Verfügung. Das reicht für viele Zwecke nicht aus, da viele ausserhalb des englischen Sprachraums verbreitete Zeichen fehlen. Abhilfe brachten zunächst die ISO-8859-Codierungen, welche die 128 Codes auf 256 Codes erweiterten und ASCII komplett enthielten. Vorteil: für jedes Zeichen wird weiterhin nur ein Byte benötigt. Nachteil: die 256 Codes reichen ebenfalls nicht aus, weshalb mehrere ISO-8859-Codierungen definiert wurden: westeuropäisch (ISO-8859-1, Latin-1), zentraleuropäisch (ISO-8859-2), kyrillisch (ISO-8859-5), usw. Die Konsequenz ist, dass ohne korrekte Einstellung der Codierung Zeichen falsch angezeigt werden, was auch nicht sehr befriedigend ist.

Die Lösung ist Unicode, ein „internationaler Standard, in dem langfristig für jedes sinnvolle Schriftzeichen oder Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wird" (Wikipedia). Mittlerweile sind mehr als 140'000 Codes definiert. Ursprünglich waren zwei Bytes pro Zeichen vorgesehen (UTF-2), was natürlich heute nicht mehr reicht, um alle Codes darzustellen. Eine Variante ist UTF-16, das UTF-2 erweitert, so dass bei Bedarf vier statt zwei Bytes zur Repräsentation verwendet werden.

JavaScript Strings

JavaScript repräsentiert Strings intern mit UTF-16 (und nicht mehr wie früher mit UTF-2), das heisst jedes Zeichen mit zwei Bytes, in Ausnahmefällen werden vier Bytes verwendet. Wichtig: Rechnen Sie damit, dass mit Vier-Byte-Zeichen nicht alles funktioniert wie erwartet. Ein paar Beispiele:

```
> '英国的'.length
3                                   // soweit gut
> "🌹".length
2                                   // das sind wohl zwei Bytes
> "🌹".charCodeAt(0)
55356                               // charCode liefert hier einen falschen Code
> "🌹".codePointAt(0)
127801                              // codePoint (neuer) funktioniert korrekt
> String.fromCodePoint(55356);
'?'
> String.fromCodePoint(127801);
'🌹'
```

UTF-8 als externe Repräsentation

Vorherrschende Codierung für Unicode-Zeichen ist heute UTF-8. Das wird auf den meisten Webseiten verwendet und ist normalerweise in Texteditoren voreingestellt. In UTF-8 werden die Unicode-Zeichencodes auf ein bis vier Bytes verteilt:

Codes mit bis zu 7 Bits:	`0xxxxxxx`	(1 Byte)
Codes mit 8 bis 11 Bits:	`110xxxxx 10xxxxxx`	(2 Bytes)
Codes mit 12 bis 16 Bits:	`1110xxxx 10xxxxxx 10xxxxxx`	(3 Bytes)
Codes mit 17 bis 21 Bits:	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	(4 Bytes)

ASCII-Zeichen werden also auch in UTF-8 mit einem Byte, seltenere Zeichen mit bis zu vier Bytes dargestellt. Vorteil: kompakt. Nachteil: Abbildung Zeichen ↔ Bytes komplizierter.

Aufgabe 3: Schriftsysteme erkennen

Mit dem Wissen über Unicode und die Zeichencodierungen können wir in einer Fortsetzung von Aufgabe 2 noch ein paar weitere Funktionen implementieren.

a. Implementieren Sie eine Funktion *scriptOfSample*, welche für ein Zeichen (String der Länge 1) den Namen des Schriftsystems ermittelt:

```
console.log( scriptOfSample("A", SCRIPTS) )      // "Latin"
console.log( scriptOfSample("英", SCRIPTS) )     // "Han"
console.log( scriptOfSample("я", SCRIPTS) )      // "Cyrillic"
```

Für nicht erkannte Zeichen, zum Beispiel Satzzeichen, soll "unknown" resultieren.

b. Implementieren Sie eine Funktion *scriptsInString*, welche für einen String zählt, wie oft jedes Schriftsystem vorkommt, und das Ergebnis in Form eines Objekts zurückgibt:

```
console.log( scriptsInString('英国的狗说 "JavaScript", "тяв"', SCRIPTS) )
// { Han: 5, unknown: 7, Latin: 10, Cyrillic: 3 }
```

c. Angenommen Sie kopieren aus einer E-Mail diesen String:
`https://postfinance.ch`

Zum Test wenden Sie Ihre Funktion *scriptsInString* darauf an und erhalten dieses Ergebnis:
`{ Latin: 17, unknown: 4, Cyrillic: 1 }`

Warum sollte Sie das skeptisch machen?

d. Fakultativ: Wenn Sie eine Datei mit Textcodierung UTF-8 anlegen, welche nur das Zeichen 🌹 enthält, und diese dann in einem Hex-Editor untersuchen, sehen Sie diese Bytefolge:
`F0 9F 8C B9`
Oben haben wir gesehen, dass das Zeichen den Code `127801` hat. Können Sie erklären, wie daraus die angegebene Bytefolge zustande kommt?

Wenn in Domainnamen Zeichen ausserhalb des ASCII-Zeichenumfangs vorkommen, wird eine Codierung namens *Punycode* (https://en.wikipedia.org/wiki/Punycode) verwendet, um Unicode in ASCII abzubilden. So sind auch Domainnamen wie dieser möglich:

`http://☃.net`