HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

# Einarbeitung in WebAssembly anhand eigener Beispielprogramme

Fakultät für Elektrotechnik und Informatik
der RWU

**Projektarbeit**

vorgelegt von

**Christian Jehle**

am

28. Februar 2021

**Erstprüfer:** Prof. Marius Hofmeister

# Inhaltsverzeichnis


**1 Einleitung** **5**

**2 Grundlagen** **8**
2.1 Vorstellung von WebAssembly . . . . . 8
2.2 Designziele von WebAssembly . . . . . 9
2.3 WebAssembly kompilieren . . . . . 10
2.4 Emscripten Compiler Toolchain . . . . . 11
2.4.1 Installation der Emscripten SDK . . . . . 11
2.4.2 Kompilieren von Programmen mit Emscripten . . . . . 11
2.4.3 Emscripten Speichermodell . . . . . 14
2.5 Potenziale und Beispiele für WebAssembly-Anwendungen . . . . . 18

**3 Zielsetzung** **21**

**4 Programmierung mit WebAssembly** **22**
4.1 Module Objekt . . . . . 22
4.2 Funktionsaufrufe . . . . . 23
4.2.1 C-Funktionen in Javascript aufrufen . . . . . 23
4.2.2 Javascript Funktionen in C aufrufen . . . . . 25
4.3 Arbeiten mit komplexen Datentypen . . . . . 27
4.4 Zugriff aus das Dateissytem mit WebAssembly . . . . . 28
4.5 Debugging WebAssembly . . . . . 30

**5 Einarbeitung von WebAssembly anhand eigener Beispiele** **32**
5.1 Anwendung: Hello World . . . . . 32
5.1.1 Ziel und Aufbau der Anwendung . . . . . 32
5.1.2 Umsetzung . . . . . 32
5.1.3 Ergebnis . . . . . 33
5.2 Anwendung: Fibonacci Folge . . . . . 33
5.2.1 Ziel und Aufbau der Anwendung . . . . . 33
5.2.2 Umsetzung . . . . . 35
5.2.3 Ergebnis . . . . . 38
5.3 Anwendung: Einfacher Taschenrechner . . . . . 39
5.3.1 Ziel und Aufbau der Anwendung . . . . . 39
5.3.2 Umsetzung . . . . . 40
5.3.3 Ergebnis . . . . . 41

5.4 Anwendung: Palindrom Test . . . . . 42
5.4.1 Ziel und Aufbau der Anwendung . . . . . 42
5.4.2 Umsetzung . . . . . 43
5.4.3 Ergebnis . . . . . 46
5.5 Anwendung: Bearbeiten von Textdateien im Browser . . . . . 47
5.5.1 Ziel und Aufbau der Anwendung . . . . . 47
5.5.2 Umsetzung . . . . . 47
5.5.3 Ergebnis . . . . . 53

**6 Bewertung** **57**
6.1 Bewertung des Konzepts von WebAssembly . . . . . 57
6.2 Bewertung des Compilers . . . . . 58
6.3 Bewertung der Programmierung mit WebAssembly . . . . . 60

**7 Zusammenfassung** **62**

**8 Ausblick** **63**

# Abbildungsverzeichnis


1.1 Häufigkeit der Internetnutzung [3] . . . . . 5
1.2 Häufigkeit der Internetnutzung nach Altersgruppen [4] . . . . . 6

2.1 Benchmarkt-Test von Nick Fitzgerald bzgl. Scala.JS unter Javascript-Engines und WebAssembly [47] . . . . . 19

4.1 Debugging Verzeichnisse im Browser . . . . . 31

5.1 Aufbau HelloWorld . . . . . 32
5.2 Ergebnis HelloWorld . . . . . 34
5.3 Aufbau Fibonacci . . . . . 34
5.4 Ergebnis Fibonacci . . . . . 38
5.5 Aufbau HelloWorld . . . . . 39
5.6 Ergebnis Taschenrechner . . . . . 42
5.7 Aufbau HelloWorld . . . . . 43
5.8 Ergebnis echtes Palindrom . . . . . 46
5.9 Ergebnis falsches Palindrom . . . . . 46
5.10 Aufbau HelloWorld . . . . . 47
5.11 Ergebnis Textbearbeitung . . . . . 53
5.12 Ergebnis Textbearbeitung, auswählen einer Datei . . . . . 54
5.13 Ergebnis Textbearbeitung, editieren einer Datei . . . . . 55
5.14 Ergebnis Textbearbeitung, speichern einer Datei . . . . . 56

6.1 Fehlermeldung im Browser . . . . . 59

# 1 Einleitung

Ob Audio/Video-Bearbeitung oder tägliche Büroarbeiten - immer mehr Bereiche, die bisher von nativen Anwendungen dominiert wurden, wechseln nun in das Internet. Insbesondere konnte man dies im vergangenen Jahr beobachten. Durch Kontakt- und Ausgangsbeschränkungen wurden Homeoffice und E-Learning immer gefragter und somit auch unterstützende Anwendungen wie bspw. Zoom, Skype, BigBlueButton oder auch Googles Onlineanwendungen zur Bearbeitung von Dokumenten, Tabellen, Präsentationen und Formularen.

Auch den nachfolgenden Statistiken (Abb 1.1 und Abb 1.2) des statistischen Bundesamtes ist zu entnehmen, dass die Anzahl der Internetnutzer sowie die Häufigkeit der Nutzung steigen. Im Besonderen lässt sich erkennen, dass bspw. mehr als 90 % der erfassten Internetnutzer jeden bzw. fast jeden Tag online sind [3]. Auch in der nächsten Statistik aus Abb. 2 [4] zeigt sich, dass nahezu 100 % der Personen aus den Altersgruppen zwischen 10 und 64 das Internet nutzen.

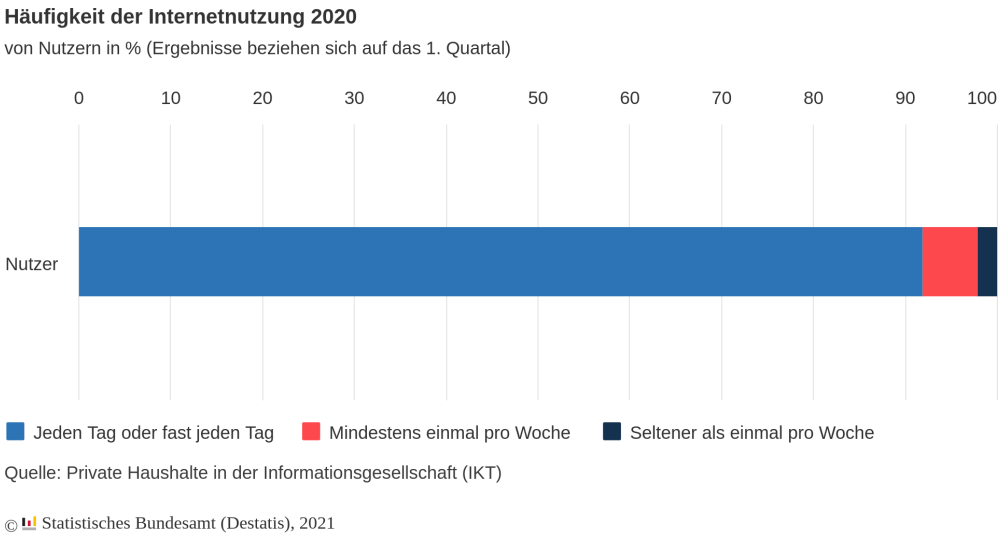


Abbildung 1.1: Häufigkeit der Internetnutzung [3]

Für Unternehmen kann es daher vorteilhaft, sein ihre Anwendungen neben der nativen Installation auch als Webanwendung auf den Markt zu bringen. Die technologischen Mittel dafür werden von modernen Browsern und ihren Javascript Engines bereitgestellt. Ein Beispiel für die Leistungsfähigkeit moderner Browser und der Programmiersprache Javascript

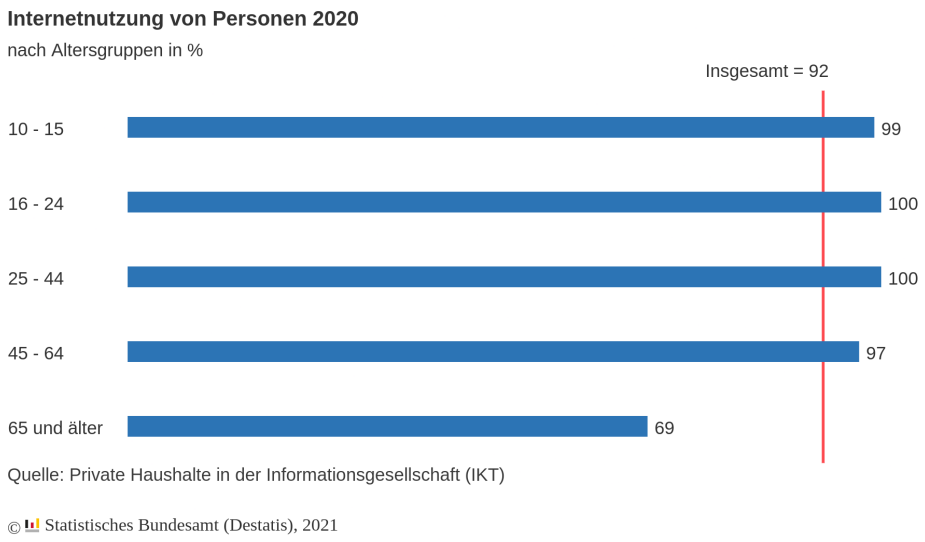


Abbildung 1.2: Häufigkeit der Internetnutzung nach Altersgruppen [4]

wurde in der Vergangenheit von dem Spielehersteller Epic erbracht. Epic hatte es sich zum Ziel gesetzt, ihre in C und C++ geschriebene Spiele-Engine, genannt Unreal Engine, in einem Browser ausführen zu können[49]. Dazu wurde die Codebasis der Software in Javascript übersetzt und konnte so erfolgreich inklusive eines Demospiels auf einem Browser ausgeführt werden.[10]

Trotz der dadurch aufgezeigten Möglichkeiten, die moderne Browser bieten, gibt es dennoch Limitierungen, die Entwickler in ihrer Arbeit und letztlich auch die fertige Anwendung einschränken. Faktoren, welche die Entwicklung von Webanwendungen einschränken, sind bspw., dass Javascript bisher die einzige Programmiersprache war, die von Browsern unterstützt wird. Da nur eine einzige Sprache unterstützt wird, gibt es zwangsläufig auch weniger Bibliotheken für verschiedene bzw. auch speziellere Probleme im Vergleich zur kumulierten Menge an Bibliotheken aus mehreren Programmiersprachen mit unterschiedlichen Schwerpunkten . Als Konsequenz bleibt oft nur die Möglichkeit eine eigene Lösung zu programmieren, was wiederum zusätzlich Zeit und Ressourcen beansprucht. Die letzte Limitierung zeigt sich in Form von schwankender Performance über unterschiedliche Browser[47]. Dies liegt vor allem daran, dass Javascript nicht im Voraus kompiliert werden kann, sondern erst zur Laufzeit[72]. Aber auch, da jeder Browserhersteller eine eigene Javascript-Engine verwendet, um geschriebene Skripte interpretieren zu können, so zum Beispiel Chromes V8 Engine[44] oder die Javascript Engine von Firefox[48]. Aus diesem Umstand können für Webanwendungen Flaschenhälse entstehen, die zu einer schlechten Benutzererfahrung und Bedienbarkeit führen. Derlei Einschränkungen haben zur Folge, dass Portierungen nativer Anwendungen zu plattformunabhängigen Webanwendungen nicht immer möglich sind.

Ein neuer Lösungsansatz für diese Problematik stellt WebAssembly dar. Mit WebAssembly

ist es nun möglich, neben Javascript auch andere Sprachen[78] in Webanwendungen einzubinden, was neue Möglichkeiten für Online Anwendungen eröffnet[75].

Durch die neuen Möglichkeiten, die sich nun durch die Verwendung anderer Programmiersprachen eröffnen, wird die Webentwicklung auch für Entwickler nativer Anwendungen interessant. Aber auch Webentwickler können von WebAssembly profitieren. So stehen ihnen nun durch neue Programmiersprachen wie C, C++ und Rust nun eine deutlich größere Auswahl an Ressourcen und Bibliotheken zur Verfügung. Durch einen vergrößerten Funktionsumfang der Browser vergrößert sich auch die Angriffsfläche für Hacking und Exploits, was die Thematik um WebAssembly auch für den Bereich der IT-Sicherheit sehr relevant macht.

Da sich der Einstieg in WebAssembly für unterschiedliche Entwicklergruppen (aufgrund des Hintergrunds und Vorwissens) verschieden schwer gestaltet, bietet diese Arbeit Wissen über die Grundlagen von WebAssembly sowie zu der Bedienung der dafür benötigten Tools (Kompiler). Ebenfalls werden die gängigsten Programmierkonzepte vorgestellt und anhand kleiner Beispielprojekte erklärt. Um die verwendeten Beispiele und Konzepte besser verstehen zu können, empfiehlt es sich die Grundlagen von Javascript zu beherrschen.

# 2 Grundlagen

## 2.1 Vorstellung von WebAssembly

WebAssembly ist eine assembler-ähnliche Sprache, die sowohl in einem für Menschen lesbaren Textformat, genannt *WAT*[50], als auch in einem binären Format zur Verfügung steht. WebAssembly selbst ist als ein Kompilier-Ziel zu verstehen[75]. Das bedeutet, dass Programmiersprachen (wie aktuell C, C++ und Rust [78]) durch einen entsprechenden Compiler in das *.wasm* Format kompiliert werden können.

Dies ermöglicht, dass nun neben Javascript auch weitere Programmiersprachen im Web verwendet werden können. Jedoch soll WebAssembly keine Alternative oder einen Ersatz für Javascript darstellen[76]. Vielmehr soll es laut Surma (einem Developer Advocat bei Google) dazu dienen, die Grenzen von Webanwendungen zu durchbrechen und Flaschenhälse zu reduzieren, um so besser Anwendungen mit umfangreicheren und neuen Funktionalitäten zu ermöglichen, die es bisher nicht gab[82].

WebAssembly wird dabei in einer stackbasierten virtuellen Maschine ausgeführt[75]. Das bedeutet, dass durch WebAssembly geladene Module innerhalb einer Sandbox oder virtuellen Maschine gefangen sind und keinen Zugriff auf bspw. das Dateisystem des Benutzers haben[69]. Stackbasiert bedeutet in diesem Kontext, dass die virtuelle Maschine einen eigenen virtuellen Speicher verwaltet, der nach dem LIFO-Prinzip arbeitet (Last-in-first-out). Auf einem solchen Stack sind hauptsächlich *push* (Hinzufügen eines neuen Elements auf den Stack) und *pop* (Entfernen des obersten Elements auf dem Stack) Operationen möglich.

Die Verwendung von WebAssembly beschränkt sich jedoch nicht nur auf den Browser, sondern kann gemäß dem definierten Designziel der Portabilität auf jeder virtuellen stackbasierten Maschine ausgeführt werden und sogar mehrere Programmiersprachen durch ein einheitliches binäres Zielformat kombinieren[69]. So kann WebAssembly beispielsweise neben dem Browser auch im Backend für Node.JS ausgeführt werden oder lokal auf einem Computer.

Aktuell wird WebAssembly von allen großen Browserherstellern in der Version 1.0 (oft auch MVP genannt, was für Minimum Viable Product steht) unterstützt (Firefox, Chrome, Safari, Edge)[75] und bietet die Möglichkeit, die Sprachen C, C++ und Rust in dieses Format zu kompilieren[64]. Grund für den Einsatz der genannten Sprachen ist, dass sie alle über eine manuelle Speicherverwaltung verfügen. Das bedeutet, es gibt keinen Garbage-Collector, der nicht mehr verwendete Referenzen auf Objekte oder Variablen entsorgt und wieder freigibt. Diesen Umstand vereinfacht das Kompilieren nach WebAssembly dieser Sprachen im Ver-

gleich zu anderen [45]. Dennoch ist geplant, Unterstützung für weitere Sprachen[78], teils auch mit automatischer Speicherverwaltung, zu einem späteren Zeitpunkt zu ermöglichen.

Als Zwischenlösung ist es bereits auch möglich, in C/C++ geschriebene Runtimes für andere Sprachen in WebAssembly zu kompilieren[10], um so auf indirektem Weg auch andere Sprachen über WebAssembly ausführen zu können. Dieses Vorgehen wurde der offiziellen Emscripten Dokumentation zufolge bereits für die Sprachen Pyhton und Lua getestet[10].

## 2.2 Designziele von WebAssembly

Wie eingangs bereits erwähnt, besteht das übergeordnete Ziel von WebAssembly darin, Bytecode im Browser ausführen zu können. Daneben gibt es jedoch noch mehrere Designziele, denen die Spezifikation von WebAssembly zugrunde liegen[69]. Die folgenden zusammengefassten Ziele entstammen den offiziellen Spezifikationen von WebAssembly.

**Geschwindigkeit von WebAssembly**
WebAssembly hat das Ziel, Code mit nahezu nativer Geschwindigkeit ausführen zu können[69].

**Sicherheit von WebAssembly**
Durch den Umstand, dass WebAssembly in einer Sandbox-Umgebung ausgeführt wird, soll verhindert werden, dass Daten korrumpiert werden und Sicherheitsverstöße stattfinden[69].

**Effizienz von WebAssembly**
Die Effizienz von WebAssembly soll den Spezifikationen zufolge durch mehrere Aspekte erhöht werden[69]. Ein Aspekt stellt den Einsatz von JIT und AOT Kompilierung dar. Unter JIT und AOT verbergen sich die Kompilieransätze Just in Time und Ahead of Time. Während bei AOT der vollständige Programmcode noch vor der eigentlichen Ausführung kompiliert wird[5], setzen JIT Compiler auf die Strategie, einzelne Programmteile je nach Bedarf während der Ausführung zu kompilieren[6]. Ein weiterer Aspekt zur Effizienz zeigt sich durch die Verwendung eines kompakten binären Formats (mit der Endung .wasm). Durch die geringe Größe lassen sich Dateien diesen Formates schneller verschicken als andere Code-Formate oder Texte[69].

Zusätzliche Eigenschaften, die sich positiv auf die Effizienz auswirken sollen, sind die Modularität, Streamfähigkeit und Parallelisierbarkeit des Formats[69]. Durch die Streamfähigkeit des binären Formats soll es möglich sein, dass der Browser die Prozesse des Decodierens, Validierens und Kompilierens bereits anstoßen kann, bevor alle Daten vorliegen. Daneben soll es möglich sein jene genannten Prozesse des Browser zu parallelisieren. Zuletzt bewirkt noch die Modularität des Formats, dass ein Programm in kleinere Teile zerlegt werden kann und somit auch separat voneinander verschickt und ausgeführt werden kann.

**Portabilität von WebAssembly**
Das letzte Designziel bezieht sich auf die Portabilität von WebAssembly[69]. Darunter ist

zu verstehen, wie unabhängig bzw. abhängig WebAssembly von äußeren Faktoren wie Hard- und Software ist.

Den Spezifikationen[69] zufolge hat WebAssembly dabei das Ziel, unabhängig von der Hardware auf allen modernen Architekturen wie Desktop PCs, aber auch auf mobilen Geräten und Embedded-Systemen ausführbar zu sein. Zudem soll es auch plattformunabhängig sein, das heißt, dass WebAssembly nicht nur an den Browser gebunden ist, sondern auch in andere Umgebungen integriert werden kann. Um dieses Ziel zu erreichen, macht WebAssembly keine Annahmen bezüglich der Systemarchitektur, die nicht in moderner Hardware vertreten ist. Zusätzlich soll auch keine Programmiersprache oder -Paradigma bevorzugt werden.

## 2.3 WebAssembly kompilieren

Wie eingangs bereits erläutert, handelt es sich bei WebAssembly um ein kompiliertes binäres Format, welches unter anderem im Browser genutzt werden kann[75]. Um Quellcode in das WASM Dateiformat zu kompilieren, gibt es mehrere Möglichkeiten, die von der verwendeten Sprache abhängen[78].

Neben Emscripten gibt es noch weitere Möglichkeiten[76], Code in das *.wasm* Format zu kompilieren, wie bspw. mit dem Compiler PNaCI[8]. Es ist möglich nach WebAssembly ohne vorgefertigte Compiler Tools wie der Emsdks (Emscripten Software Development Kit) zu kompilieren[65]. Dieses Vorgehen nimmt jedoch den Vorteil, bereits bestehende C/C++ Codebasen nutzen zu können und setzt tiefes Wissen rund um WebAssembly und Compiler-Programmierung voraus. Zudem werden durch Emcc die in C zur Verfügung stehenden Standard-Bibliotheken verwaltet[19] und ein File-System[22] emuliert, die File System API. Im Nachfolgenden sind die drei gängigsten Programmiersprachen für WebAssembly und ihre Kompilierungsmöglichkeiten aufgelistet[65]

- C/C++ nach Wasm kompileren
  Um C oder C++ Code in das WebAssembly Format zu kompilieren, kann die Emscripten Compiler Toolchain eingesetzt werden[9].

- Rust nach Wasm kompileren
  Anders als im Falle von C/C++ gibt es für Rust ein Paket bzw. Tool namens **wasm-pack**[54]. Dieses Tool kann komfortabel über cargo[46] (Rusts Build System und Paket Manager) installiert werden. Wasm-pack unterstützt Entwickler in vielerlei Hinsicht. So kann das Tool genutzt werden, um den Build-Prozess zu unterstützen und zu testen sowie auch, um WebAssembly Code zu verpacken und entweder für Webanwendungen mit Javascript integrierbar zu machen (sowohl für den Browser als auch für Node.JS) oder für npm(Node Package Manager aus NodeJS) als Paket bereitzustellen[39].

- AssemblyScript nach Wasm kompileren
  TypeScript, was ein typisiertes Superset von Javascript darstellt[74], kann mittels der Binaryen Compiler Toolchain[36] (die ebenfalls als Teil von Emscripten verbaut ist) in AssemblyScript[35] kompiliert werden.

## 2.4 Emscripten Compiler Toolchain

Emscripten ist eine Compiler Toolchain, die es ermöglicht, geschriebenen C oder C++ Code in das WebAssembly Dateiformat zu kompilieren[10]. Daneben bietet es zahlreiche zusätzliche Funktionen an, die das Entwickeln mit WebAssembly wesentlich vereinfachen. So wird beispielsweise beim Kompilieren eine Javascript Datei erzeugt, die die Schnittstelle zum WASM Code darstellt und so den Zugriff und die Verwendung der darin bereitgestellten Funktionalitäten managt[14].

### 2.4.1 Installation der Emscripten SDK

```
# Get the emsdk repo
git clone https://github.com/emscripten-core/emsdk.git

# Enter that directory
cd emsdk

# Fetch the latest version of the emsdk
# (not needed the first time you clone)
git pull

# Download and install the latest SDK tools
./emsdk install latest

# Make the "latest" SDK "active" for the current user
./emsdk activate latest

# Activate PATH and other environment variables in
# the current terminal
source ./emsdk_env.sh
```

Für Windows empfiehlt Emscripten, die SDK nicht direkt auf Windows, sondern auf einem Subsytem für Linux und somit auch einer Linuxumgebung auszuführen[16]. Die oben aufgeführten Befehlszeilen entstammen der Emscripten Dokumentation zur Installation unter Linux und können dort genauer eingesehen werden[16].

### 2.4.2 Kompilieren von Programmen mit Emscripten

Um ein in C geschriebenes Programm nun in das WebAssembly-Format zu kompilieren, ist es als erstes notwendig, den Pfad zum zuvor installierten emcc Programm und alle anderen Umgebungsvariablen zu aktivieren[16]. Dazu muss man der Installations-Anleitung[16]

zufolge lediglich in das Verzeichnis wechseln, in welchem die Emsdk installiert wurde und folgende Befehle ausführen:

```
cd emsdk
source ./emsdk_env.sh
```

Die Änderungen am Pfad, welche durch diese Datei vorgenommen werden, sind jedoch an die aktuelle Terminal Sitzung gebunden. Wird das Terminal geschlossen, muss der Befehl im nächsten Terminal erneut ausgeführt werden, um Emscripten unter dem Programmnamen **emcc** aufrufen zu können[16]. Dies hat auch zur Folge, dass sich dazu kein eigenes Shell-Skript schreiben lässt, welches automatisiert in das entsprechende Verzeichnis der **emsdk** wechselt und die Datei **emsdk_env.sh** ausführt. Eine solche Datei würde zwar erfolgreich die entsprechenden Umgebungsvariablen dem Pfad hinzufügen, jedoch nur innerhalb des eigenen Prozesses. Sobald das Skript beendet ist, bzw. erfolgreich terminiert wurde, sind alle Änderungen verloren und **emcc** kann nicht aufgerufen werden. Anstelle eines Skriptes kann jedoch ein Alias(ein Name, der unter Linux in der *bashrc* Datei verwendet werden kann, um als Stellvertreter für einen definierten Befehl zu dienen) verwendet werden, der es erlaubt, schnell in das Verzeichnis **emsdk** zu wechseln und die entsprechende Datei auszuführen.

Nachdem dieser Befehl erfolgreich ausgeführt wurde, ist es möglich, das Programm emcc von der Kommandozeile aus zu nutzen. Die grundlegende Syntax lässt sich anhand dieses einfachen Beispiels erklären[32]:

```
emcc input.c -o output.js -s WASM=1
```

**input.c** ist das C Programm, das kompiliert werden soll. Der Parameter **-o** gibt an, welchen Namen und Dateityp das Ergebnis des Kompiliervorgangs haben soll[32]. In diesem Beispiel wurde als Ergebnis eine Javascript Datei namens **output.js** definiert. Der Dateityp Javascript bedeutet, dass neben dem puren binären **.wasm** Format auch ein sogenannter Glue-Code generiert wird, der es ermöglicht, in Javascript Zugriff auf WebAssembly Module zu erhalten[68]. Mögliche Dateiformate sind:

- .js
  Eine Javascript Datei und eine WASM Datei werden erzeugt
  (bspw. output.js, output.wasm)[17]

- .html
  Erzeugt ebenfalls eine Javascript und WASM Datei als auch eine HTML Seite[17] mit ca. 1298 Zeilen Code inklusive zusätzlichem Glue-Code

- .mjs
  Eine Javascript Datei nach dem Ecmascript 6 Standard sowie eine WASM Datei[17]

- .bc
  Erzeugt LLVM Bitcode(Low Level Virtual Machine)[17]

- .o
  Erzeugt eine WebAssembly Objekt Datei [17]

- .wasm
  Purer WebAssembly Code, ohne unterstützende Javascript Dateien [17]

Der letzte Parameter **-s** kann dazu benutzt werden bestimmte Optionen zu setzen. Die Option **WASM=1** gibt das Kompilierziel an. Es kann nach Javascript kompiliert werden (WASM=0), nach Wasm(WASM=1) oder beides gleichzeitig(WASM=2) [33].

Weitere Parameter, die für emcc verwendet werden können, werden im Folgenden weiter ausgeführt. Aus Gründen der Komplexität und des Umfang können nicht alle Build Optionen gezeigt werden. Alle hier nicht aufgeführten Optionen und Parameter können in emcc direkt nachgeschlagen werden mittels des Befehls `emcc --help` oder falls dort nicht vorhanden, im GitHub Repository unter `src/settings.js` [33].

**Zusätzliche Emscripten Funktionalitäten**
Um in Javascript Zugriff auf einige Helfer-Funktionen zu haben, die Emscripten anbietet, kann die Option `-s EXTRA_EXPORTED_RUNTIME_METHODS='["function"]'` [33] eingesetzt werden. Mögliche Parameter statt function sind bspw. `ccall`, `cwrap`, `getValue` und `setValue` [27]. Die Schreibweise in eckigen Klammern deutet an, dass die Parameter als Array angegeben werden und dementsprechend mehrere Werte an emcc weitergereicht werden können.

**C-Funktionen in Javascript verfügbar machen**
Damit der Compiler weiß, welche Funktionen möglicherweise aus Javascript heraus referenziert und aufgerufen werden, müssen diese Funktionen dem Compiler manuell bekannt gemacht werden[12]. Hierzu kann das emcc Kommando `-s EXPORTED_FUNCTIONS='["_main"]'` [33]verwendet werden. In diesem Beispiel wird die Methode **main()** bekannt gemacht und kann zu einem späteren Zeitpunk aus Javascript aufgerufen werden. Wichtig zu beachten ist, dass jede Funktion, die referenziert wird, mit einem Unterstrich beginnen muss[12].

**Optionen zur Speicherverwaltung**
Speicher ist begrenzt. Das gilt auch für den virtuellen stackbasierten Speicher, der in WebAssembly verwendet wird. Wenn für eine Anwendung nicht eindeutig vorausgesagt werden kann, wie viel Speicher sie benötigen wird, kann im Compiler die Option
`-s ALLOW_MEMORY_GROWTH=1` [33]gesetzt werden. Dies hat zur Folge, dass sich die Speichermenge den Anforderungen der Anwendung entsprechend anpassen kann [33]. Trotz der Tatsache, dass sie einige Optimierungsschritte im Compiler blockiert, verursacht diese Option der Dokumentation zufolge keinen bis minimalen Mehraufwand[26].

**Einbindung des Dateisystems**
Programme, welche in C geschrieben sind, können über die stdio.h Standardbibliothek Zugriff auf das Dateisystem ausüben. Dazu zählt das Lesen, Anlegen und Modifizieren von Dateien sowie das Navigieren im System. Da WebAssembly in einer Sandbox ausgeführt wird, hat es keinen Zugriff zu Systemen, die außerhalb liegen[77]. Um dennoch mit Dateien arbeiten zu können, bietet Emscripten die Möglichkeit, ein Dateisystem zu emulieren und stellt dazu

eine eigene API zur Verfügung[22], die über Javascript verfügbar ist. In den meisten Fällen erkennt Emscripten selbst, ob eine Anwendung das Filesystem nutzt und fügt das emulierte Dateisystem nur in den Kompilierprozess mit ein, wenn es notwendig ist[22]. Es gibt jedoch über die Option `-s FORCE_FILESYSTEM=1` [33] auch die Möglichkeit den Kompiler zu zwingen, das Dateisystem zusätzlich zu kompilieren[22].

### 2.4.3 Emscripten Speichermodell

#### ArrayBuffer, DataView und typisierte Arrays

Dieses Unterkapitel dient als Vorbereitung, um das Konzept der Speicherverwaltung und des -modells durch Emscripten besser nachvollziehen zu können. Die wichtigsten Bestandteile dabei sind die 3 Javascript Klassen ArrayBuffer[53], DataView[55] und TypedArray[60], die in den folgenden Abschnitten genauer erläutert werden. Die dabei verwendeten Informationen stammen aus der jeweiligen Dokumentation auf dem Mozilla Developer Network.

**ArrayBuffer in Javascript**[53]
ArrayBuffer sind generische Arrays mit fester Länge. Anders als normale Arrays enthalten diese Buffer rohe binäre Daten (einzelne Bytes). Anders ausgedrückt handelt es sich bei einem ArrayBuffer um nichts anderes, als um ein Array aus Bytes. Ein weiterer Unterschied zu herkömmlichen Arrays ist, dass man als Programmierer keinen direkten Zugriff auf dieses Array hat. Stattdessen müssen typisierte Arrays oder DataViews verwendet werden. Sie ermöglichen den Buffer in einem spezifischen Datenformat anzuzeigen und einen Schreib- und Lesezugriff zur Verfügung zu stellen.

ArrayBuffer können entweder manuell bzw. über den Klassenkonstruktor erzeugt werden oder aus bereits existierenden Daten extrahiert werden. Im ersten Beispiel wird ein neuer ArrayBuffer mit der Länge 8 erzeugt.

```
1 const length = 8;
2 const buffer = new ArrayBuffer(length);
3 }
```

Im nachstehenden Beispiel wird in Zeile 1 eine Referenz auf ein `<input>` HTML-Tag über das `document` Objekt geholt und in der Variable namens `input` gespeichert. Das `<input>` Element im HTML-Code wurde so definiert, dass es nur Dateien als Eingabe akzeptiert. Anschließend wird der Variable `input` ein EventListener zugewiesen, der ausgelöst wird, sobald der Benutzer eine Datei ausgewählt hat. In Zeile 6 wird über den FileReader[58] der Prozess angestoßen, durch den ein ArrayBuffer aus der eingelesenen Datei extrahiert wird. Sobald dieser Prozess fertig ist, wird das `onload` Event in Zeile 8 ausgelöst.

```
1 const input = document.querySelector('input');
2
3 input.onchange = ev => {
4     const file = input.files[0];
5     const reader = new FileReader();
6     reader.readAsArrayBuffer(file);
```

```
7
8       reader.onload = () => {
9           let buffer = reader.result;
10          console.log(buffer);
11  }
```

**DataView in Javascript**[55]
Das DataView Objekt bietet die Möglichkeit, auf einem ArrayBuffer[53] Daten in jedem numerischen Datentyp zu lesen und zu schreiben. Dabei kann explizit angegeben werden, welche Endianness (Byte-Reihenfolge) [57] verwendet werden soll.

```
1 const buffer = new ArrayBuffer(16);
2
3 const dataView = new DataView(buffer);
4 const offset = 12, value = 4194765292;
5
6 dataView.setUint32(offset, value, false);
7 console.log(dataView.getUint32(12);
```

In diesem Code-Beispiel wurde zuerst ein neuer ArrayBuffer mit der Länge 16 angelegt (Zeile 1) und anschließend eine DataView auf diesen ArrayBuffer erzeugt (Zeile 3). In Zeile 4 wurden zwei Variablen (offset, value) definiert, die die Position im Buffer (offset) und den entsprechenden Wert (value) enthalten. Diese Werte werden in Zeile 6 durch die Funktion `dataView.setUint32()` [56] in den ArrayBuffer geschrieben. Der letzte Parameter, in diesem Fall `false`, gibt an, ob der Wert in LittleEndian oder BigEndian kodiert werden soll. Im Falle von LittleEndian[57] muss darauf geachtet werden, auch bei `get` Funktionen den optionalen Kodierungsparameter zu übergeben, da im Standardfall stets BigEndian[57] angenommen wird. Neben `uint32` werden auch folgende Datentypen unterstützt:

- int8 und uint8
- int16 und uint16
- int32 und uint32
- float32
- float64
- bigint64 und biguint64

**Typisierte Arrays in Javascript**[60]
Ähnlich der DataView bietet auch TypedArray eine Visualisierung der binären Daten im ArrayBuffer sowie Manipulationsmöglichkeiten der Daten. Der Hauptunterschied zum vorherigen View Objekt besteht darin, dass keine TypedArray Instanz erzeugt werden kann, da es sich hierbei um eine Elternklasse handelt. Stattdessen wird zu jedem numerischen Datentypen eine typisierte Array Klasse angeboten[62]:

- Int8Array
  Ein Array aus 8-Bit signierten Integern mit einem Wertebereich von -128 bis 127

- Uint8Array
  Ein Array aus 8-Bit unsignierten Integern mit einem Wertebereich von 0 bis 255
- Int8ClampedArray
  Ein Array mit dem selben Eigenschaften wie das Uint8Array
- Int16Array
  Ein Array aus 16-Bit signierten Integern mit einem Wertebereich von -32768 bis 32767
- Uint16Array
  Ein Array aus 16-Bit unsignierten Integern mit einem Wertebereich von 0 bis 65535
- Int32Array
  Ein Array aus 32-Bit signierten Integern mit einem Wertebereich von -2147483648 bis 2147483647
- Uint32Array
  Ein Array aus 32-Bit unsignierten Integern mit einem Wertebereich von 0 bis 4294967295
- Float32Array
  Ein Array aus 32-Bit Floating Points mit einem Wertebereich von $1.2*10^{-38} bis 3.4*10^{38}$
- Float64Array
  Ein Array aus 64-Bit Floating Points mit einem Wertebereich von $5.0 * 10^{-324}$ bis $1.8 * 10^{308}$
- BigInt64Array
  Ein Array aus 64-Bit signierten Integern mit einem Wertebereich von $-2^{63}$ bis $2^{63} - 1$
- BigUint64Array
  Ein Array aus 64-Bit unsignierten Integern mit einem Wertebereich von 0 bis $2^{64} - 1$

Ein typisiertes Array kann auf mehrere Arten erzeugt werden [61].

### Speichermodell und -typen

Nachdem nun die grundlegenden Eigenschaften und Funktionalitäten typisierter Arrays vorgestellt wurden, werfen wir jetzt einen Blick auf das in Emscripten verwendete Speichermodell. Der offiziellen Dokumentation von Emscripten zufolge, nutzt die Compiler-Toolchain im Hintergrund ein einziges typisiertes Array, auf welches verschiedene Sichten zur Verfügung gestellt werden[18]. Diese Sichten werden durch typisierte Arrays realisiert, die in der von Emscripten erzeugten Javascript Datei (auch Glue-Code genannt) erzeugt werden und dem Nutzer so Zugriff auf den ArrayBuffer mittels verschiedener typisierter Arrays ermöglichen[18]. Welche Typen dem Nutzer zur Verfügung stehen, zeigt die Tabelle im nächsten Unterkapitel.

Emscripten realisiert dieses Modell durch den Einsatz des in Javascript vorhandenen WebAssembly Objekts[51], welches unter anderem die Funktion

`WebAssembly.Memory({initial: ... [, maximum: ... [, shared: ... ]]})` [52] zur Verfügung stellt. Das WebAssembly Objekt kann entweder manuell durch einen Programmierer oder von Kompilern wie Emscripten dazu benutzt werden, WebAssembly Code (enthalten in Dateien mit der Endung *.wasm*) zu laden, einen neuen Speicher oder eine Tabelle zu erzeugen und um mögliche Fehler abzufangen[51]. Der folgende Code-Ausschnitt stammt aus einer von Emscripten erzeugten Javascript Datei und zeigt, wie in Emscripten ein neuer virtueller Speicher angelegt wird.

```
1  if (Module['wasmMemory']) {
2      wasmMemory = Module['wasmMemory'];
3   } else
4   {
5      wasmMemory = new WebAssembly.Memory({
6        'initial': INITIAL_INITIAL_MEMORY / WASM_PAGE_SIZE
7        ,
8        'maximum': 2147483648 / WASM_PAGE_SIZE
9      });
10  }
```

Die Instanziierung des ArrayBuffers findet in Zeile 5 statt. Die dabei übergebenen Parameter definieren die Größe des ArrayBuffers. Die Konstanten `INITIAL_INITIAL_MEMORY` und `WASM_PAGE_SIZE` werden im Code mit den Werten 16777216 und 65536 initialisiert. Beide Parameter, initial und maximum, geben die Größe des virtuellen Speichers gemessen in WebAssembly Pages an. Eine WebAssembly Page hat dabei immer eine konstante Größe von 65,536 Bytes, bzw. 64 KiB (ebenfalls im Glue-Code dokumentiert). Somit wird im obigen Beispiel ein Speicherbereich von initial 256 WebAssembly Pages (16384 KiB, entspricht 16,78 MB) und maximal 32768 WebAssembly Pages (2097152 KiB, entspricht 2,15 GB) reserviert.

Interessant ist, dass der Wert des maximum Attributs von einem Kompilierparameter abhängig ist. Im dargestellten Beispiel wurde der Code mit der Kommandozeilen Option `-s ALLOW_MEMORY_GROWTH=1` [33] kompiliert. Wird die gleiche Codebasis zugrunde gelegt und ohne diese emcc Option kompiliert, ändert sich die Berechnung für den Maximumwert des ArrayBuffers:
`'maximum': INITIAL_INITIAL_MEMORY / WASM_PAGE_SIZE`
Alle anderen Werte wie `INITIAL_INITIAL_MEMORY` und `WASM_PAGE_SIZE` werden davon nicht beeinflusst.

### Memory Typen

Die nachfolgened Tabelle veranschaulicht, welche Speicheransichten Emscripten anbietet, welche Speicherkapazität sie haben und welche Mechanismen dahinter verwendet werden. Alle Speicheransichten sind über folgende Schreibweise im Javascript-Code aufrufbar[68]: `Module.HEAP8`. Die genaue Verwendung wird in den folgenden Kapiteln erläutert.

| Speicheransicht | Datentyp | Verwendeter TypedArray |
|---|---|---|
| HEAP8 | 8-bit signierter Integer Speicher | Int8Array |
| HEAP16 | 16-bit signierter Integer Speicher | Int16Array |
| HEAP32 | 32-bit signierter Integer Speicher | Int32Array |
| HEAPU8 | 8-bit unsignierter Integer Speicher | Uint8Array |
| HEAPU16 | 16-bit unsignierter Integer Speicher | Uint16Array |
| HEAPU32 | 32-bit unsignierter Integer Speicher | Uint32Array |
| HEAPF32 | 32-bit float Speicher | Float32Array |
| HEAPF64 | 64-bit float Speicher | Float64Array |

Die Kenntnis um die verwendeten Speicheransichten und den dabei eingesetzten typisierten Arrays stammt aus der von Emscripten erzeugten Javascript-Datei. Innerhalb dieser Datei, bzw. des Glue-Codes, finden sich folgende Code-Zeilen, die Aufschluss über die eingesetzten Datentypen zur Speicheransicht geben:

```
function updateGlobalBufferAndViews(buf) {
  buffer = buf;
  Module['HEAP8'] = HEAP8 = new Int8Array(buf);
  Module['HEAP16'] = HEAP16 = new Int16Array(buf);
  Module['HEAP32'] = HEAP32 = new Int32Array(buf);
  Module['HEAPU8'] = HEAPU8 = new Uint8Array(buf);
  Module['HEAPU16'] = HEAPU16 = new Uint16Array(buf);
  Module['HEAPU32'] = HEAPU32 = new Uint32Array(buf);
  Module['HEAPF32'] = HEAPF32 = new Float32Array(buf);
  Module['HEAPF64'] = HEAPF64 = new Float64Array(buf);
}
```

## 2.5 Potenziale und Beispiele für WebAssembly-Anwendungen

Einer der Hauptvorteile, durch den sich WebAssembly von Javascript abhebt und dadurch neue Potenziale für Webanwendungen darstellt, ist die Performance (effizient und schnell) von Anwendungen[75]. Eine häufig getroffene Annahme ist hierbei laut Surma auf der Google I/O '19, dass WebAssembly grundsätzlich schneller und performanter ausgeführt werden kann als Javascript[82]. Dies sei, wie im Weiteren ausgeführt wird, jedoch nur zum Teil richtig, da Javascript theoretisch gesehen gleich schnell wie WebAssembly Code ausgeführt werden kann[82]. Jedoch wird die Performance und Geschwindigkeit im Falle von Javascript stark vom jeweils verwendeten Browser beeinflusst[82]. Aufgrund der Tatsache, dass es sich bei Javascript um eine Sprache handelt, die erst zur Laufzeit interpretiert und kompiliert wird, ist das Ergebnis dieses Vorgangs stark von der verwendeten Javascript Engine abhängig. Das zeigt bspw. auch ein Benchmark-Test zwischen Googles V8 Engine[44] und Microsofts Chakra Engine [38]. Das Ergebnis des Tests zeigte einen höheren Score für die V8 Engine[7]. Daraus resultiert, dass einige Operationen und Vorgänge in einem Browser in moderater Geschwindigkeit beendet werden können, während die gleichen Operationen in einem anderen Browser mehr Zeit in Anspruch nehmen. Durch diese Schwankungen ge-

staltet es sich schwierig zuverlässig abschätzen zu können, wie lange eine Operation einer Webanwendungen dauert[82].

WebAssembly auf der anderen Seite ist, anders als Javascript, keine Skriptsprache, sondern kann lokal auf einem Computersystem kompiliert[10] und optimiert[26] werden. Dadurch wird lediglich Bytecode zum Browser ausgeliefert. Dieser kann aufgrund meist geringer Größe schnell übertragen werden und ist sofort zur Ausführung bereit. Ein ausführlicher Benchmark Test des Mozilla Entwicklers Nick Fitzgerald zeigte hierzu, dass WebAssembly in der Ausführung zwar nicht immer schneller ist, jedoch anders als Javascript in allen Browsern ähnlich schnell ausgeführt wurde[47].

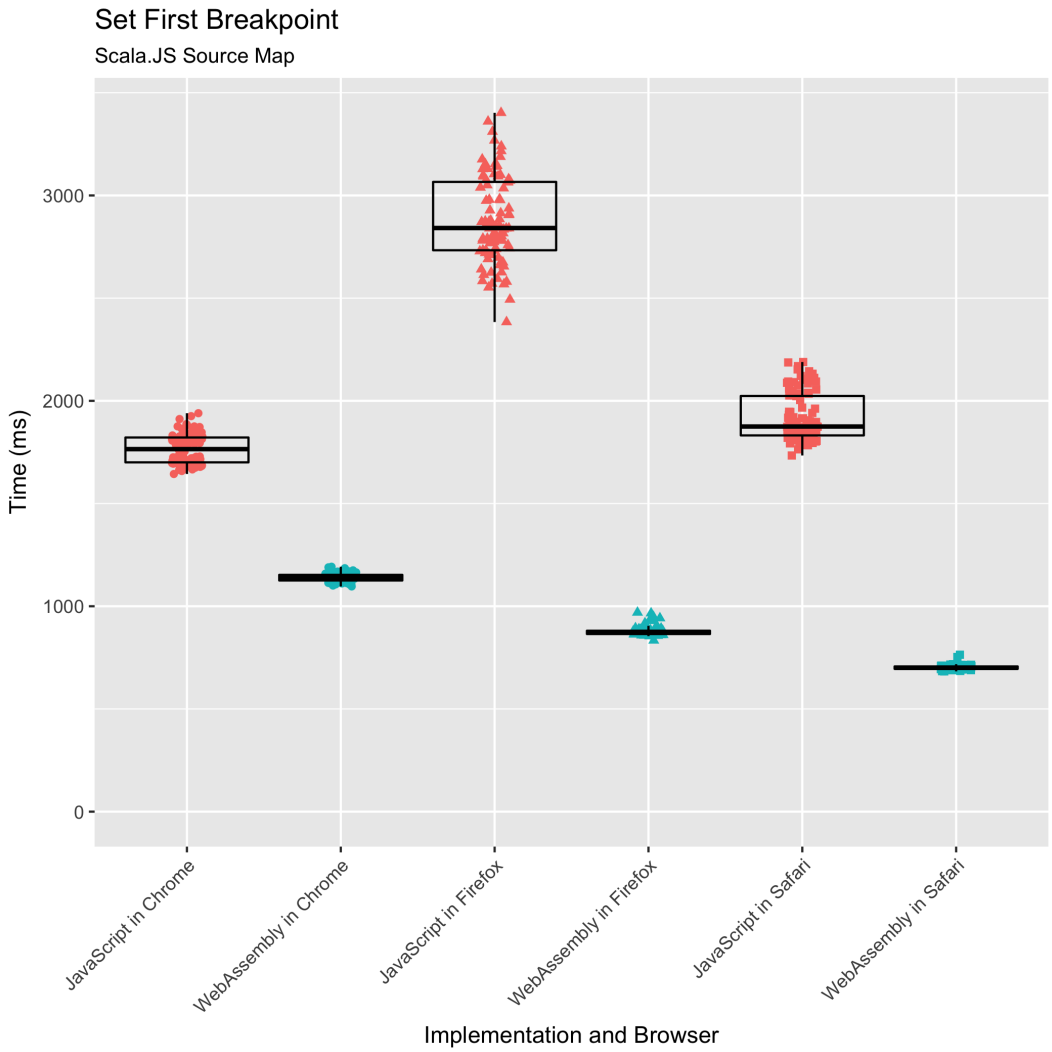


Abbildung 2.1: Benchmarkt-Test von Nick Fitzgerald bzgl. Scala.JS unter Javascript-Engines und WebAssembly [47]

Demzufolge bietet die neue Eigenschaft, Code in unterschiedlichen Browsern mit gleicher

Geschwindigkeit ausführen zu können, neue Potenziale für die Webentwicklung im Bezug auf Realtime-Anwendung.

Neben der Geschwindigkeit selbst besteht ein weiterer Vorteil von WebAssembly darin, dass nun durch drei zusätzliche Sprachen auch eine Vielzahl neuer Bibliotheken hinzukommen, die im Web ausgeführt werden können. Da die Sprachen C und C++ schon länger existieren als Javascript(C wurde 1972 veröffentlicht[81] und Javascript in 1995[73] ) und noch immer sehr aktuell sind[71], gibt es demzufolge auch mehr Bibliotheken mit anderen Möglichkeiten. So zum Beispiel OpenCV[67], eine Bibliothek für mehrere Sprachen (unter anderem C++), die Funktionalitäten wie Bildbearbeitung, Videoanalyse sowie Video-Codecs zur Verfügung stellt. OpenCV steht für Open Source Computer Vision und ist eine Bibliothek, die erstmals 1999 durch Intel veröffentlicht wurde[66]. In Kombination mit WebAssembly ermöglicht diese Bibliothek bspw. Gesichtserkennung über den Webbrowser.

**Aktuelle Beispiele:**

Zu den ersten Anwendungen, die mit WebAssembly geschrieben wurden, zählt die Portierung der Unreal Engine 4 ins Web sowie die Unity Engine[10]. Beide genannten Anwendungen werden zur Spieleentwicklung eingesetzt und zählen zu den größten Wettbewerbern auf dem Markt. Zur Portierung wurde in beiden Fällen auf Emscripten als Kompiler zurückgegriffen. Neben der Spieleentwicklung gibt es jedoch auch andere Bereiche und Firmen, die bereits WebAssembly nutzen. So auch die Firma Autodesk mit ihrer Anwendung namens AutoCAD, die dank WebAssembly nun auch im Web verfügbar ist[45]. AutoCAD ist ein Programm, welches zur Bearbeitung und Entwicklung technischer Zeichnungen und Vektorgrafiken eingesetzt wird[2].

Als letztes Beispiel ist Squoosh[70] zu nennen. Dabei handelt es sich um eine Webanwendung zur Bildkomprimierung, die durch mehrere Bildkompressoren viele Bearbeitungsmöglichkeiten zur Verfügung stellt, die andernfalls nur auf nativen Anwendungen zu finden wären[40]. Squoosh schöpft die Möglichkeiten von WebAssembly insofern aus, als dass sie gleich Bibliotheken unterschiedlicher Sprachen durch das einheitliche Kompilierformat miteinander kombiniert. Verwendet wurden Bibliotheken und Funktionalitäten der Sprachen C++ und Rust[40].

# 3 Zielsetzung

Ziel dieser Arbeit ist es, Wissen zur Verwendung um WebAssembly und seinen Anwendungsmöglichkeiten zu bündeln und für Entwickler bereit zu stellen sowie die aktuellen Möglichkeiten zur Benutzung dieser Technologie aus der Sicht von Webprogrammierern zu untersuchen, kennen zu lernen und zu bewerten. Des Weiteren soll analysiert werden, ab wann sich der Einsatz von WebAssembly lohnt und welche Kenntnisse für die erfolgreiche Nutzung notwendig sind.

Im Rahmen der Analyse wurden 5 kleinere Web-Anwendungen mit WebAssembly geschrieben, mit jeweils unterschiedlichem Fokus auf bestimmte Funktionalitäten. WebAssembly bietet die Möglichkeit aktuell 3 native Programmiersprachen in den Browser zu bringen, namentlich C, C++ und Rust. Im Rahmen dieser Arbeit wurde an dieser Stelle insofern eine Einschränkung vorgenommen, als dass die vorgestellten Programme in C programmiert werden.

Grund dafür ist, dass die verschiedenen Sprachen auch unterschiedliche Möglichkeiten im Bezug auf WebAssembly bieten. So ist es bspw. möglich mit Rust direkt für WebAssembly zu programmieren, da speziell für diese Programmiersprache die Entwicklung von Mozilla stark gefördert wurde. Durch diese Diversität ändert sich auch der Anwendungskontext für WebAssembly. Statt eine Web-Anwendung mit bereits vorhandenen Bibliotheken zu bereichern und zu erweitern, kann stattdessen auch direkt für Web-Anwendungen mit WebAssembly programmiert werden. Um einen besseren Fokus auf die grundlegenden Funktionalitäten und Möglichkeiten von WebAssembly zu bewahren, wurde für diese Arbeit die Sprache C gewählt.

Eine weitere daraus folgende Einschränkung ist die Wahl des Compilers. Da diese Arbeit das Arbeiten mit WebAssembly unter C vorstellt, fällt die Wahl des Compilers auf Emscripten Compiler Frontend (emcc genannt) und widmet einen Teil des Grundlagen Kapitels dem Umgang und der Theorie der Emscripten Compiler Toolchain.

# 4 Programmierung mit WebAssembly

Dieses Kapitel hat das Ziel, einen Einblick in die wichtigsten Programmierkonzepte und Vorgehen von WebAssembly mit Javascript zu geben.

## 4.1 Module Objekt

Das Module Objekt[24] ist Hauptbestandteil des Glue-Codes, der von emcc generiert wird, und stellt die eigentliche Schnittstelle zwischen Javascript und dem WebAssembly Code dar. Über dieses Objekt können auf alle zusätzlich exportierten Funktionen von emcc und mehr zugegriffen werden. So stellt das Module Objekt beispielsweise ein Event bereit, das aufgerufen wird, sobald alle WebAssembly Module erfolgreich geladen wurden[25]. Dies ist ein sehr wichtiges Event, da erst ab diesem Punkt mit dem Wasm Code interagiert werden kann.

Ein mögliches Vorgehen, um zu überprüfen, ob WebAssembly vollständig geladen wurde, zeigt das folgende Beispiel.

```
1 let wasmReady = false;
2 Module.onRuntimeInitialized = () => {
3     wasmReady = true;
4 }
```

Die Variable `wasmReady` befindet sich außerhalb der Funktion und ist daher auch im Scope anderer Funktionen verfügbar. Initial wird die Variable auf `false` gesetzt. Erst nachdem das Event `onRuntimeInitialized` [25] aufgetreten ist, wird der Wert auf `true` gesetzt. Dadurch ist für nachfolgende Funktionen erkennbar, wann sie Zugriff auf die kompilierten WebAssembly Funktionen haben.

Ein Problem, das sich bei diesem Vorgehen jedoch abzeichnet, ist, dass Funktionen keine Rückmeldung haben, ab wann alle Module vollständig geladen wurden. Dies ist jedoch nur dann ein Problem, wenn der Nutzer der Webanwendung den Zeitpunkt bestimmen kann, ab wann bspw. eine Funktion aus WebAssembly aufgerufen wird. Im anderen Fall, wenn eine oder mehrere Funktionen einmalig und ohne Nutzerinteraktion ausgeführt werden sollen, reicht es den entsprechenden Aufruf innerhalb des oben genannten Events zu platzieren. Falls jedoch Nutzerinteraktion im Rahmen der Webanwendung vorgesehen ist, wären mögliche Lösungen zum einen alle Interaktionsmöglichkeiten zu blockieren, bis die Module geladen wurden oder entsprechende Aufrufe in einer Warteschlange für eine bestimmte Zeitdauer zu speichern und zu einem späteren Zeitpunkt auszuführen.

## 4.2 Funktionsaufrufe

Funktionsaufrufe sind ein wesentlicher Teil der Programmierung und ermöglichen in diesem Kontext die Interaktion zwischen Javascript Code und dem WebAssembly Code. Die dazu verfügbare Schnittstelle von Emscripten lässt sich der Dokumentation zufolge auf zwei Arten nutzen[23].

### 4.2.1 C-Funktionen in Javascript aufrufen

Dies stellt den vermutlich am häufigsten genutzten Anwendungsfall dar, da WebAssembly größtenteils als Anbieter für Funktionen bereits entwickelter Bibliotheken fungiert.

Wenn C-Funktionen innerhalb der Webanwendung referenziert und genutzt werden sollen, muss dies bereits beim Kompilieren berücksichtigt werden[23]. Dem Compiler muss zunächst mitgeteilt werden, dass und welche C-Funktionen nach dem Kompiliervorgang erhalten bleiben sollen (siehe Kapitel 2.4.2 *C-Funktionen in Javascript verfügbar machen*). Alle übrigen Codezeilen, die im Weiteren nicht genutzt oder initial ausgeführt werden, werden von der LLVM (welche als ein Teil des Emscripten Compilers verbaut ist[31]) als Dead-Code eliminiert, um die Dateigröße zu reduzieren[33].

Zusätzlich müssen noch beim Kompilieren zwei bzw. eine der beiden Funktionen exportiert werden, die emcc als Teil des Glue-Codes automatisch generiert (siehe Kapitel 2.5.2 *Zusätzliche Emscripten Funktionalitäten*). Die Funktionen `ccall` und `cwrap` werden später dazu genutzt, die gewünschte C-Funktion aufzurufen und managen dabei auch die Datentypkonvertierung zwischen Javascript Datentypen wie Number, String und Boolean und den C bekannten Datenformaten[12].

Ein beispielhafter Kompilierbefehl wäre:

```
emcc input.c \
-o output.js \
-s WASM=1\
-s EXPORTED_FUNCTIONS='["_helloworld"]' \
-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]'
```

Dieser Befehlt würde bewirken, dass in Javascript die Funktionen `cwrap` (automatisch generierter Glue-Code)und `helloworld` (beispielhafte C-Funktion) verfügbar sind.

Nachdem die zwei neuen Dateien *output.js* und *output.wasm* generiert wurden, kann nun eine neue Javascript Datei angelegt werden oder der Code gleich innerhalb eines Script-Tags in HTML hinterlegt werden. Wichtig ist, dass in jedem Fall die Datei *output.js* vor dem eigenen Code geladen wird, wodurch der Zugriff auf das Module Objekt ermöglicht wird. In den folgenden Beispielen soll die in C programmierte Funktion `helloworld` ausgeführt werden. Diese Funktion wird einmal mittels `cwrap` und anschließend mit `ccall` aufgerufen. Beide Funktionen bewirken das gleiche Ergebnis, haben jedoch unterschiedliche Parameter

und Einsatzmöglichkeiten[12]:

**Funktionsaufruf mit ccall[28]**
Im Folgenden wird die Funktion `ccall` anhand der API-Dokumentation auf emscripten.org genauer erklärt. Diese Funktion ist die Basisfunktion, die auch von `cwrap`[30] verwendet wird. Das Ergebnis der aufgerufenen Funktion, sofern es eines gibt, wird von `ccall` zurückgegeben. Als Parameter nimmt sie den Namen der gewünschten Funktion entgegen, ihren Rückgabewert, einen Array, der die Datentypen der Funktionsparameter enthält, und zuletzt ein weiteres Array, welches alle Parameter umfasst, die der Funktion übergeben werden. Wenn die Funktion keinen Rückgabewert hat, kann null übergeben werden. Mögliche Datentypen sind:

- boolean
- number
  Kann für jeden numerischen Wert oder auch Pointer genutzt werden (da Pointer numerische Speicheradressen enthalten)
- string
  Kann für `char` Werte und `char*` genutzt werden
- array
  Kann für Javascript Arrays oder typisierte Arrays, die 8-Bit Integer Werte enthalten, verwendet werden. Wenn ein typisiertes Array übergeben wird, muss es entweder eine Instanz des Uint8Array oder Int8Array sein. Dies liegt daran, dass die Daten intern in ein C-Array bestehend aus 8-Bit Integern geschrieben werden.

Das folgende Codebeispiel zeigt den Einsatz von ccall zum Aufruf einer C-Funktion `multiply`, die zwei `int` Parameter erwartet, diese multipliziert und das Ergebnis zurückgibt:

```
1 const result = Module.ccall("multiply", "number", ["number", "number"],
     [3, 5]);
2
3 console.log(result);
```

Nach dem Ausführen der beiden Befehle erscheint in der Browserkonsole die Zahl 15.

**Funktionsaufruf mit cwrap[30]**
Im Folgenden wird die Funktion `cwrap` anhand der API-Dokumentation auf emscripten.org genauer erklärt. Die nächste Möglichkeit, C-Funktionen über Javascript anzusprechen, ist `cwrap`. Diese Funktion nutzt intern ebenfalls `ccall`[28], was in der von emcc generierten Javascript Datei eingesehen werden kann. Anders als `ccall` gibt diese Funktion als Rückgabewert nicht das Ergebnis der Operation zurück, sondern die Referenz auf die Funktion. Das bedeutet, dass die Referenz der C-Funktion `multiply` in einer Variable gespeichert werden kann und so `multiply` einfach und komfortabel unter dem Namen dieser Variable im Skript verwendet werden kann. Auch bei `cwrap` muss wieder angegeben werden, welchen Rückgabetyp die Funktion und ihre Parameter haben. Die Parameter selbst werden nicht übergeben.

Dieses Codebeispiel soll verdeutlichen, wie eine Funktionsreferenz durch `cwrap` gespeichert und aufgerufen werden kann. In der ersten Zeile wird die Referenz auf die C-Funktion `multiply` in der Javascript Variable namens `funcMultiply` gespeichert. In Zeile 3 wird die in `funcMultiply` gespeicherte Funktion aufgerufen, die Parameter übergeben und das Ergebnis in`result` gespeichert.

```
1 const funcMultiply = Module.cwrap("multiply", "number", ["number", "number
    "]);
2
3 const result = funcMultiply(3,5);
4 console.log(result);
```

### 4.2.2 Javascript Funktionen in C aufrufen

Neben der Möglichkeit, Funktionen aus anderen Sprachen in Javascript zu nutzen, ist es mithilfe der Emscripten Bibliothek auch möglich, Javascript Funktionen in C aufzurufen und eigene Funktionen und Zeilen innerhalb einer C/C++ Datei zu schrieben[13].

Damit der Kompiler zwischen Javascript und dem eigentlichen C Code unterscheiden kann, stellt die emscripten.h Bibliothek drei Möglichkeiten zur Verfügung, um Javascript Code zu kennzeichnen[13]. Die erste Möglichkeit präsentiert sich in Form einer Funktion namens `emscripten_run_script`[21]. Diese Funktion nimmt als Parameter einen String entgegen und ist ein komfortabler Weg, einzelne Aufrufe von Javascriptcode in C auszuführen.

```
1 void js_hello_world()
2 {
3     emscripten_run_script("alert('Hello World!')");
4     emscripten_run_script("console.log('Hello World!')");
5 }
6
7 int main()
8 {
9   js_hello_world();
10 }
```

Das obige Beispiel zeigt, wie `emscripten_run_script` eingesetzt werden kann, um die Ausgabe *Hello World!* als alert und in der Konsole zu sehen. Zusätzlich können aber auch eigene Funktionen ausgeführt werden, die in einer Javascript Datei implementiert und in die index.html Datei eingebunden wurden. Ein Beispiel für einen solchen Aufruf zeigt der nachfolgende Code. Vorausgesetzt wird die Javascript Funktion `increment`:

```
1 let counter = 0;
2
3 function increment() {
4     counter++;
5 }
```

Im Javascript-Code wird die Variable `counter` definiert, welche durch den Aufruf der Funktion `increment` um eins erhöht werden soll. Der folgende C-Code nutzt diese Funktion, um dieses Ergebnis zu erreichen. Die Variable `counter` soll dabei um 10 erhöht werden.

```
void counter(int repeat)
{
    for (int i = 0; i < repeat; i++) {
        emscripten_run_script("increment()");
    }
    emscripten_run_script("console.log('Counter: ', counter)");
}

int main()
{
  counter(10);
}
```

Eine andere Möglichkeit, um Javascript in C einzusetzen, stellen `EM_JS` und `EM_ASM` dar[13]. Während `EM_JS`[21] es ermöglicht, innerhalb von C neue Javascript Funktionen zu definieren und implementieren, die anschließend im normalen C-Code ausgeführt werden können, ist es innerhalb eines Abschnitts, der mit `EM_ASM`[21] gekennzeichnet ist, möglich, mit Javascript zu programmieren. Die Unterschiede sollen durch die beiden nachfolgenden Beispiele verdeutlicht werden.

```
EM_JS(void, call_alert, (), {
    alert("Hello World!\nThis was a function call in C using JS.");
});

void addition(int num1, int num2)
{
    int res = EM_ASM_INT({
        let result = $0 + $1;
        console.log('Calculated result: ' + result);
        return result;
    }, num1, num2);
    printf("%d\n", res);
}

int main()
{
  call_alert();
  addition(40, 2);
}
```

In Zeile 1 des obigen Codes wird durch den Einsatz von `EM_JS` die Javascript Funktion namens `call_alert` implementiert. Die Funktion erzeugt einen Alert im Browser und gibt einen Text aus.

In Zeile 7 wird wiederum `EM_ASM` verwendet, um die Berechnung der Addition vorzunehmen, das Ergebnis zu speichern, es in der Browserkonsole auszugeben und anschließend als Rückgabewert auszugeben. Auffällig ist zum einen, dass `ES_ASM` mit dem Zusatz `_INT` versehen wurde. Dies liegt daran, da durch `ES_ASM` ein Integer-Wert zurückgegeben[21] und in der Variable `res` gespeichert wird. Alternativ ist für Gleitkommazahlen die Endung `_DOUBLE` verfügbar[13]. Ein anderes Merkmal ist, dass der eigentliche Programm-Code innerhalb der

geschweiften Klammern steht und anschließend (Zeile 11) erst die Parameter übergeben werden. Da die Namen, unter denen die Parameter ansprechbar sind, nicht selbst gewählt werden können, sind sie über die Notation `$<Stelle>` verfügbar[13]. So ist der erste Parameter bspw. wie in Zeile 8 zu sehen `$0`. Zu beachten ist der Emscripten-Dokumentation zufolge, dass innerhalb des `EM_ASM` Macros nur einfache Anführungszeichen und nicht doppelte verwendet werden, da dies andernfalls zu Fehlern führt[13].

## 4.3 Arbeiten mit komplexen Datentypen

In den bisherigen Beispielen wurde die Verwendung von WebAssembly lediglich mit nativen Datentypen wie `number` bzw. `int` gezeigt. In diesem Kapitel soll erklärt werden, wie komplexe Datentypen wie `string` bzw. `char[]` von Javascript nach C transportiert werden können.

Da der interne Speicher in WebAssembly in der Form eines ArrayBuffer[18] vorliegt, müssen zwangsläufig, wie in Kapitel 2.4.3 erläutert, entweder DataViews oder typisierte Arrays verwendet werden, um auf diesen Buffer zugreifen zu können. Emscripten greift intern auf die Verwendung von typisierten Arrays zurück und speichert diese in Variablen (bspw. HEAP8 für einen Int8Array, ebenfalls in Kapitel 2.4.3 unter Memory Typen zu finden)[18].

Um nun auf den Speicher zuzugreifen, gibt es in Projekten, die mit Emskripten kompiliert wurden, zwei Möglichkeiten. Die erste und auch empfohlene Variante nutzt Funktionen die von dem Kompiler automatisch im Glue-Code generiert werden[11]: `getValue(ptr, type)` und `setValue(ptr, value, type)`. Die zweite Möglichkeit, mit dem Speicher zu interagieren, ist direkt mit den typisierten Arrays die Emscripten generiert zu arbeiten[11]. Dieses Vorgehen wird von der offiziellen Dokumentation nicht empfohlen[11] und sollte nur verwendet werden, wenn man weiß, wie damit umzugehen ist. Ein Vorteil dieser Methode ist jedoch eine bessere Performance bzw. Geschwindigkeit, da weniger Funktionsaufrufe stattfinden[11]. Der nachfolgende Code soll beispielhaft verdeutlichen, wie mit dem Speicher gearbeitet werden kann.

```
1 const str = "Hello World";
2 const len = str.length;
3
4 const int8Str = new Int8Array(intArrayFromString(str));
5 const ptr = Module._malloc(len * int8Str.BYTES_PER_ELEMENT);
6
7 Module.HEAP8.set(int8Str, ptr);
8
9 Module._free(ptr);
```

Dieser Codeblock bewirkt, dass der String `"Hello World"` in den WebAssembly Speicher bzw. genauer über die HEAP8 Speichersicht in den zugrundeliegenden ArrayBuffer geschrieben wird. Durch dieses Vorgehen ist der String nun auch über den C-Code zugreifbar, indem in Javascript beim Funktionsaufruf entsprechend der erstellte Pointer übergeben wird. Dies

ist jedoch nur bis zu Zeile 9 möglich, da hier der Pointer und damit auch die Speicherstelle wieder freigegeben werden. In Zeile 4 wird der String zuerst über die von Emscripten bereitgestellte Funktion `intArrayFromString(str)` [29] zu einem 0-terminierten Array bestehend aus Integer Werten nach dem UTF-8 Standard umgewandelt. Anschließend wird dieses Array als Konstruktor-Parameter für einen neuen Int8Array verwendet, um so alle Werte zu übergeben. Zeile 5 wiederum nutzt ebenfalls eine Emscripten-Funktion namens `_malloc()` [27], die es ermöglicht, einen Pointer auf eine reservierte Speicherstelle mit angegebener Größe zu erzeugen.

Wie man diesen String nun wieder aus dem Speicher auslesen und konvertieren kann, zeigt der nächste Code Auszug.

```
1 const typedArr = new Int8Array(Module.HEAP8.buffer, ptr, len);
2 const castedStr = intArrayToString(typedArr);
3
4 console.log(castedStr); // Hello World
```

In der ersten Zeile wird erneut ein typisiertes Array erzeugt, allerdings durch einen anderen Konstruktor. Durch den Ausdruck `Module.HEAP8.buffer` wird der Speicher durch die entsprechende Speicheransicht (`HEAP8`) als ArrayBuffer ausgelesen. Die weiteren Parameter `ptr` und `len` repräsentieren den sogenannten byteOffset und die length. In Kombination bewirkt dieser Konstruktor, dass ein neues typisiertes Array erzeugt wird und lediglich der Bereich des ArrayBuffers, der ab dem Pointer beginnt und durch den Längen Parameter eingegrenzt wird, ausgewählt wird. Durch die zweite Zeile wird das zuvor erstellte IntegerArray wieder zu einem String umgewandelt und schließlich in Zeile 4 auf der Konsole ausgegeben.

## 4.4 Zugriff aus das Dateissytem mit WebAssembly

Das Arbeiten mit Dateien ist ein wesentlicher Bestandteil vieler Anwendungen mit WebAssembly, was vor allem daran liegt, dass WebAssembly u. a. besonders für das Bearbeiten von Audio-, Bild- oder Videodateien geeignet ist[80]. Da WebAssembly jedoch in einer Sandbox, also einer virtuellen Umgebung ausgeführt wird, hat es keinerlei Zugriff auf das Dateisystem des Nutzers[77]. Um dennoch mit Dateien arbeiten zu können, stellt der Emscripten Kompiler die sog. File System API zur Verfügung, die es ermöglicht, innerhalb der WebAssembly-Umgebung ein eigenes virtuelles Dateisystem zu verwalten[22].

Um diese Schnittstelle jedoch nutzen zu können, muss der Compiler diese erst zur Verfügung stellen. Emscripten hat die Fähigkeit zu bemerken, ob ein C Programm den Zugriff auf das Dateisystem benötigt oder nicht[22]. Es kann durch die entsprechende Compilieroption `-s FORCE_FILESYSTEM=1` [33] auch erzwungen werden, die Filesystem API bereitzustellen. Das Design der API wurde stark vom Linux/POSIX Dateisystem inspiriert und stellt ähnliche Funktionalitäten zur Verfügung[22]. Die Schnittstelle unterstützt dabei mehrere Systemtypen, welche unter der Filsystem API unter emscripten.org einsehbar sind. Die folgenden Systemtypen entstammen dieser Quelle:

- MEMFS
  Dieses System ist standardmäßig ausgewählt und wird auf das Verzeichnis `"/"` gemountet.
- NODEFS
  Dieses System wiederum ist nur von Nutzen, wenn es im Kontext von NodeJS ausgeführt wird. Grund dafür ist, dass es für das Schreiben und Lesen im System die **FS API** aus NodeJS benutzt.
- IDBFS
  Diese Abkürzung steht für IndexedDB File System. Das Dateisystem nutzt eine IndexedDB[59], um große Datenmengen im Browser speichern und schnell wiederfinden zu können. Dies wird ermöglicht durch die Implementierung der Funktion `FS.syncfs()`
- WORKERFS
  Das Dateisystem kann nur innerhalb eines WebWorkers verwendet werden und ermöglicht es Dateien und Blobs (bspw. Bilder) innerhalb eines Workers lesen zu können, ohne sie kopieren zu müssen. Dies kann besonders nützlich für große Dateien sein, für die andere Operationen zu teuer (bzgl. CPU Leistung) sind.

Um ein anderes Dateisystem außer MEMFS nutzen zu können, muss es von der Schnittstelle gemountet werden. Dies kann durch den Aufruf der Funktion `FS.mount(type, opts, mountpoint)` erreicht werden[22]. Für den Parameter type kann ein gewünschter Systemtyp angegeben werden, während opts für ein Konfigurationsobjekt steht. Der mountpoint wiederum kann dazu benutzt werden, um das System auf einem bereits existierenden Pfad zu mounten.

```
1 FS.mkdir('/example');
2 const opts = { root: '.'}
3 FS.mount(NODEFS, opts, 'example');
```

In Zeile 1 wird durch die File System API ein neues Verzeichnis namens *example/* angelegt. In der zweiten Zeile wird der Konfigurationsparameter als `opts` gespeichert und in Zeile 3 verwendet, um das Dateisystem vom Typ NODEFS auf dem Pfad *example/* zu mounten.

Ein beispielhafter Anwendungsfall wäre, dass ein Benutzer einer Webanwendung eine Datei auf die Anwendung hochladen kann und Javascript mittels der Filesystem API die Datei in das virtuelle System lädt. Wie dieser Prozess in Javascript abgebildet werden kann, zeigt das nachfolgende Codebeispiel.

```
1 const fileName = "Beispiel.txt"
2 const fileContent = "Beispielhafter Inhalt einer Textdatei.";
3
4 try {
5   FS.createDataFile("/", fileName, fileContent, true, true);
6 } catch (ex) {
7   console.error("Error occured, while writing file to virtual memory.", ex
     );
8 }
```

## 4.5 Debugging WebAssembly

Sofern nicht anders eingestellt, gestaltet sich das Debuggen in WebAssembly als äußerst anspruchsvoll, da zum Zeitpunkt der Ausführung lediglich der Javascriptcode im Debug-Modus der Entwicklerkonsole überprüft werden kann. Der C Code ist zu diesem Zeitpunkt nicht mehr vorhanden. Stattdessen wird eine in WebAssembly kompilierte Version des Codes in binärem Format verwendet. Um den in C geschriebenen Code zu debuggen, empfiehlt es sich daher, dies vor dem eigentlichen Kompilieren zu tun, bspw. mit Unittests oder kleineren Testskripten zur Überprüfung der Funktionalität. Eine weitere Art des Debuggins ist das Arbeiten mit Konsolenausgaben, da dies der einfachste Weg ist, Zwischenergebnisse einzusehen. Durch Emscripten werden Konsolenausgaben in C, wie `printf("Konsole: Text");` automatisch konvertiert, sodass die entsprechende Ausgabe in der Browserkonsole erscheint.

Neben diesem Ansatz bietet der Emscripten Kompiler jedoch noch eine weitere Möglichkeit an, um sogar im Browser Zugriff auf die ursprünglichen C-Dateien zu erhalten und diese im Debugger untersuchen zu können. Dieses Vorhaben wird über sog. Source-Maps realisiert[15]. Um dem Kompiler mitzuteilen, die notwendigen Informationen zum Debuggen zu erhalten, reicht es, das Flag `-g` zu übergeben[17]. Dieses Flag hat die Möglichkeit noch genauer zu spezifizieren, wie viel Debug-Informationen erhalten werden sollen, was letztlich auch die Größe der kompilierten Dateien beeinflusst, da manche Compiliereroptimierungen dadurch nicht mehr umgesetzt werden können[15]. Verfügbare Level sind `-g0` bis `-g4`. Dabei steht `-g` für den gleichen Informationsgehalt wie `-g3`[17]. Um nun auch die zuvor erwähnten Source-Maps zu realisieren, wird zusätzlich die Option `--source-map-base <base-url>` eingesetzt (in der emcc Dokumentation[17] oder durch den Aufruf von `emcc --help` zu finden). Wichtig dabei ist es, dass die `<base-url>` auf den Pfad zeigt, in dem später die erzeugte Source-Map-Datei liegt. Daher muss die Pfadangabe mit einem Schrägstrich enden. Der folgende Code-Ausschnitt zeigt einen beispielhaften Compilieraufruf mit Debugg-Informationen:

```
#!/usr/bin/env bash

emcc main.c \
-o file.js \
-s WASM=1 \
-s EXPORTED_FUNCTIONS='["_main", "_append_to_file", "_malloc", "_free"]' \
-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap", "getValue", "setValue"]' \
-g --source-map-base http://localhost:5500/exampleApp/build/
```

Das Ergebnis dieser Operation lässt sich im Browser nochmals veranschaulichen. Die folgende Abbildung zeigt einen Ausschnitt aus der Entwicklerkonsole des Browser, in dem das zuvor kompilierte Projekt ausgeführt wird. Interessant sind die Verzeichnisse *file://* und *wasm/*. Das Verzeichnis *wasm/* enthält eine Datei namens *00019f52*, welche automatisch generiert wurde und das sogenannte **.wat** Format repräsentiert. Dieses Format ist eine für Menschen lesbare Form des binären **.wasm** Formats. Es können zwar Breakpoints in diese Datei gesetzt werden, das Debugging selbst ist allerdings nur für Entwickler mit genauen Kenntnissen dieses Formats hilfreich. Das andere Verzeichnis, das in der Abbildung markiert wurde, heißt *file://* und enthält wiederum ein Unterverzeichnis, in dem eine C Datei vorliegt. Die Datei

main.c ist die im vorherigen Codebeispiel gezeigte kompilierte Datei, die nun im Browser verwendet werden kann.

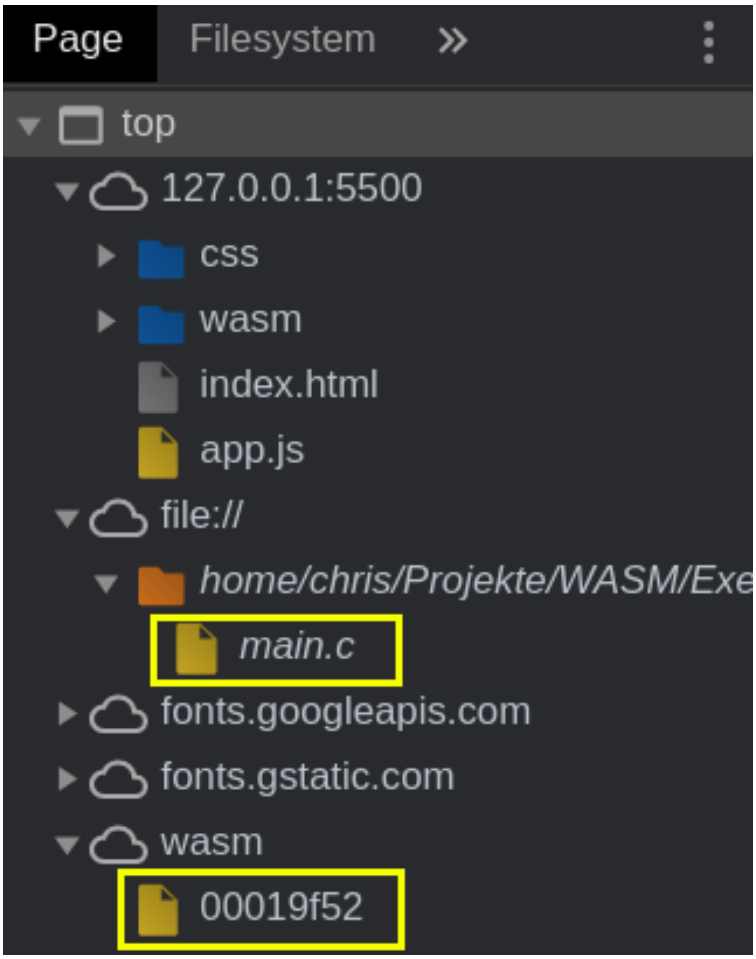


Abbildung 4.1: Debugging Verzeichnisse im Browser

# 5 Einarbeitung von WebAssembly anhand eigener Beispiele

Im Verlauf dieses Kapitels werden fünf beispielhafte Projekte vorgestellt, um das Arbeiten mit WebAssembly und dem Emscripten Compiler praktisch aufzuzeigen. Die vorgestellten Projekte können auf dem für dieses Projekt erstellte GitHub-Repository nochmals eingesehen, heruntergeladen und modifiziert werden. Das Repository ist unter dem Link *https://github.com/coffee-chris/WebAssembly_Examples* erreichbar.

## 5.1 Anwendung: Hello World

Das HelloWorld Projekt nutzt ein von Emscripten erstelltes HTML Template und zeigt die Konsolenausgabe *"Hello World!"* an.

### 5.1.1 Ziel und Aufbau der Anwendung

Ziel dieser Anwendung war es, die grundlegende Funktionalität von Emscripten als Compiler anhand eines einfachen Programms zu testen.

Der Aufbau des Projekts lässt sich in Abb. 5.1 genauer betrachten. Es gibt eine C-Datei, die entsprechend kompiliert werden soll. Im Verzeichnis darüber befindet sich lediglich ein Shell-Skript namens build.sh. Diese Datei stößt den Kompilierprozess an und erzeugt dadurch die Dateien index.html, index.js und index.wasm.


Abbildung 5.1: Aufbau HelloWorld

### 5.1.2 Umsetzung

Das folgende Codebeispiel zeigt den Quelltext aus main.c:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("\nHello World!");
    return 0;
}
```

Es gibt nur eine Funktion, die sogenannte Hauptfunktion `main()`, die lediglich die Zeichenkette *"Hello World!"* auf der Konsole ausgibt und dann 0 als Rückgabewert liefert. Dieser Code wird mittels des Emscripten Compiler in das WebAssembly Format übersetzt sowie der entsprechende Glue-Code für die Nutzung von Javascript erzeugt, inklusive einer HTML Datei, die ein Template bereitstellt.

```
#!/bin/bash
emcc src/main.c \
-o index.html \
-s EXIT_RUNTIME=1
```

Durch dieses Skripte werden die drei weiteren Dateien namens index.html, index.js und index.wasm generiert. Grund dafür ist, da das Ergebnis der Operation als HTML Datei angegeben wurde, definiert durch `-o index.html`.

### 5.1.3 Ergebnis

In der folgenden Abbildung ist die dynamisch erzeugte index.html zu sehen, die auf einem lokalen Web-Server gehostet wird. Ebenfalls zu sehen ist der Text *"Hello World!"*, der in einem Konsolen ähnlichen Bereich auf der Webseite angezeigt wird. Dabei handelt es sich um das zuvor geschriebene C-Programm. Ebenfalls sichtbar bei einem Blick in die Entwicklerkonsole des Browser ist die Nachricht auf der Konsole mit selbigem Text.

## 5.2 Anwendung: Fibonacci Folge

Diese Anwendung kann als Eingabe des Benutzers eine numerische Zahl entgegennehmen. Anschließend werden alle Stellen der Fibonacci Reihe bis zur angegebenen Zahl berechnet und angezeigt.

### 5.2.1 Ziel und Aufbau der Anwendung

Das Ziel dieser Anwendung ist es aufzuzeigen, wie man aus Javascript heraus Funktionen aus C mittels `cwrap` [30] aufruft und wie man Arrays als Funktionsparameter übergeben kann. Die nachfolgende Abbildung zeigt den Aufbau im Detail. Das Projekt unterteilt sich in die Verzeichnisse *src/* und *web/*. Während *src/* den benötigten C-Code für die Anwendung enthält, befinden sich im Verzeichnis *web/* alle anderen Dateien, die für die Programmierung der Webanwendung notwendig sind. Die Datei app.js enthält den nötigen Programmcode um das Verhalten der Website zu programmieren als auch die von Emscripten bereitgestellten

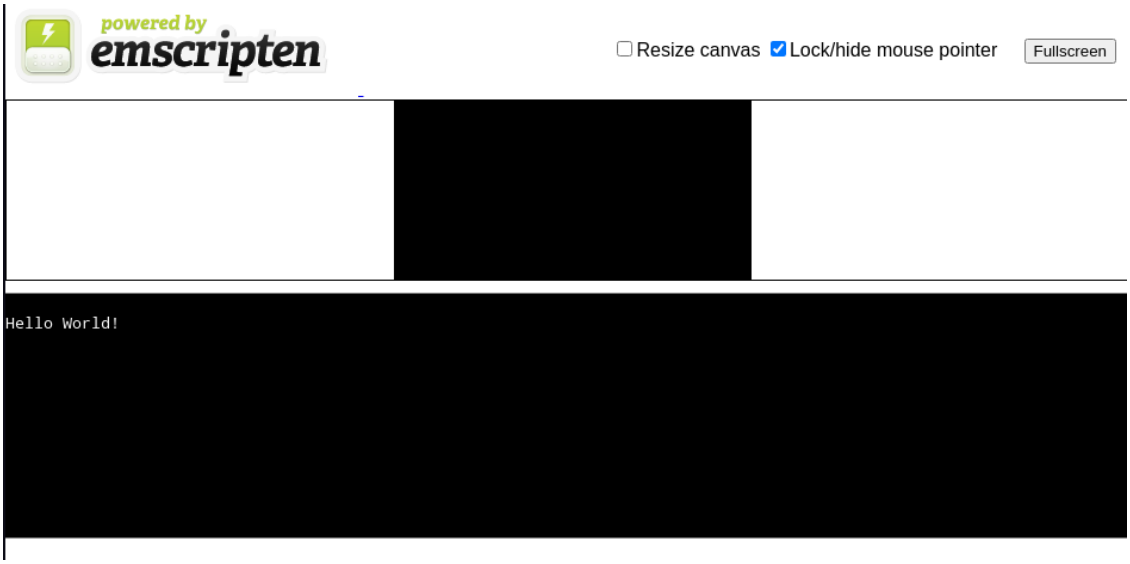


Abbildung 5.2: Ergebnis HelloWorld

Funktionen aus der Datei main.js zu nutzen. Der *build/* Ordner enthält primär das Shell-Skript build.sh, das dazu benutzt wird, den restlichen Inhalt zu erzeugen.

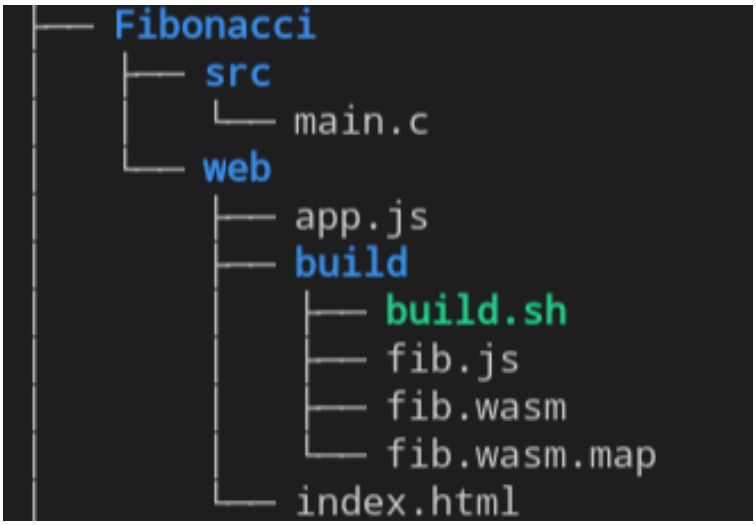


Abbildung 5.3: Aufbau Fibonacci

### 5.2.2 Umsetzung

**Native Programmierung**

Als Erstes war es für die Umsetzung nötig, eine Funktion in C zu programmieren, welche es ermöglicht, die Fibonacci Folge bis zu einem bestimmten Punkt berechnen zu können.

```
void calc_fib_sequence(int limit, uint8_t *res_ptr)
{
    int pre = 0;  // predecessor
    int suc = 1;  // successor
    int value = 0;

    for (int i = 1; i <= limit; i++) {
        res_ptr[i-1] = pre;

        printf("%d", pre);
        if (i != limit)
            printf(", ");

        value = pre + suc;
        pre = suc;
        suc = value;
    }
}
```

Die Funktion `calc_fib_sequence` nimmt zwei Parameter entgegen: eine Begrenzung für die Berechnung der Folge und einen Pointer vom Typ uint8_t, der dazu dient, alle Stellen der Folge als Array zu speichern und so dem Nutzer das Ergebnis zur Verfügung stellt. Der Rest des Codes dient zur eigentlichen Berechnung der Reihe.

**Kompilierung**

Nachdem der geeignete Code nun zur Verfügung steht, besteht der nächste Schritt darin, ihn mit Emscripten zu kompilieren. In diesem Fall soll kein HTML erzeugt werden, sondern lediglich der WebAssembly Code und Javascript. Hierfür wird emcc mit den folgenden Optionen innerhalb der build.sh Datei ausgeführt:

```
#!/bin/bash
emcc ../../src/main.c \
-o fib.js \
-s WASM=1 \
-s EXPORTED_FUNCTIONS='["_calc_fib_sequence", "_malloc", "_free"]' \
-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]'
```

Neben der eigentlichen Kompilierung sollen außerdem noch einige Funktionen exportiert und zur Verfügung gestellt werden. Dazu zählt die in C programmierte Funktion `calc_fib_sequence` , als auch `_malloc` und `_free`, die für das Arbeiten mit dem Speicher nötig sind, sowie die Funktion `cwrap`, die dadurch vom Javascript Glue-Code bereitgestellt wird.

**Web-Programmierung**

Der wichtigste Schritt besteht nun darin, die erzeugte Datei fib.js und die noch zu imple-

mentierende app.js in die HTML Datei index.html einzubinden. Dies lässt sich durch die Verwendung des `<script>` Tags mit entsprechender Pfad-Angabe lösen. Zu beachten ist hierbei, dass die beiden Dateien am besten als letzter Eintrag im `<body>` Tag eingetragen werden, da hierdurch sichergestellt werden kann, dass alle anderen HTML Elemente geladen wurden und somit für Javascript zugänglich sind.

Als Nächstes muss die kompilierte Funktion in Javascript aufgerufen werden. Dazu wird wiederum die Funktion `cwrap` eingesetzt. Der folgende Ausschnitt aus der Datei app.js zeigt, wie dies umgesetzt wurde.

```
1 let calcFib = null;
2
3 Module.onRuntimeInitialized = () => {
4     wasmLoaded = true;
5     calcFib = Module.cwrap( "calc_fib_sequence",
6                             "number",
7                            ["number", "number"]);
8     console.log("WASM loaded ...");
9 }
```

Nachdem WebAssembly vollständig geladen wurde, was ab Zeile 3 garantiert wird, wird der Variable `calcFib` die Referenz auf die Funktion`calc_fib_sequence` mittels `cwrap` übergeben. Dadurch kann die Variable `calcFib` aus Zeile 1 im späteren Code wie eine Funktion verwendet werden.

Nachdem nun die Funktion selbst zur Verwendung bereitsteht, geht es jetzt darum, die dafür notwendigen Parameter vorzubereiten. Der erste Parameter ist eine numerische Zahl und kann daher der Funktion direkt übergeben werden. Emscripten übernimmt dabei die Aufgabe der Typenkonvertierung: Aus dem Datentyp `number` aus Javascript muss ein entsprechender, für C kompatibler Integer Typ werden. Der nächste Parameter, der übergeben werden soll, ist sinngemäß ein Array, in welches die Ergebniswerte gespeichert und später wieder ausgelesen werden sollen. Anders als Zahlen können Arrays nicht direkt als Parameter übergeben werden. Stattdessen muss im Speicher von WebAssembly ein entsprechender Platz für das Array allokiert und der Pointer auf die erste Stelle als Parameter gesetzt werden. Grund dafür ist, dass in C Arrays als Pointer übergeben werden, da ein Arrays in C nichts anderes als eine Sequenz auf- oder absteigender Speicheradressen ist(C von A bis Z[81], Kapitel 14.5). Die Funktion `writeToMem` löst dieses Problem, indem sie je nach angegebener Größe einen Uint8Array in den WebAssembly Speicher schreibt und den Pointer zurückgibt.

```
1  function writeToMem(size = 0) {
2      if (size > 0 && typeof size == "number") {
3          const length = size, offset = 1;
4          const uint8Arr = new Uint8Array(length);
5
6          const ptr = Module._malloc(length * offset);
7          Module.HEAPU8.set(uint8Arr, ptr);
8          return ptr;
9      }
10 }
```

Bevor der eigentliche Funktionsaufruf stattfinden kann, muss noch eine Funktion geschrieben werden, die es ermöglicht, nach der Operation die Ergebnisse aus dem Speicher auslesen zu können. Der nachfolgende Javascript Code zeigt die Funktion `readFromMem`, die genau diese Anforderung umsetzt.

```
1  function readFromMem(ptr=null, length=0) {
2      const resultArr = [];
3      if (ptr && length > 0) {
4          const uint8Arr = new Uint8Array(  Module.HEAPU8.buffer,
5                                            ptr,
6                                            length);
7
8          uint8Arr.map(value => resultArr.push(value));
9      }
10     return resultArr;
11 }
```

Damit das Array aus dem Speicher gelesen werden kann, ist der Pointer auf die erste Speicherstelle zwingend notwendig sowie die zu erwartende Größe des Array. In Zeile 4 wird ein Uint8Array aus dem Speicher mittels des Pointers und der Länge des Arrays ausgelesen und seine Werte in Zeile 5 in ein reguläres Array übertragen, welches als Ergebnis der Operation ausgegeben wird.

Der vollständige Funktionsaufruf ist im nächsten Codeblock zu sehen: Als Erstes wird die Begrenzung der Sequenz aus einem Input-Feld gelesen und in ein numerisches Format konvertiert (Zeile 1 und 2). Als Nächstes wird in Zeile 6 das Array in den Speicher geschrieben und ein Pointer erzeugt. Der Pointer und die Begrenzung werden in Zeile 9 schließlich der Variable `calcFib` übergeben, welche die Funktion `calc_fib_sequence` enthält. Zum Schluss wird das Ergebnis der Operation aus dem Speicher ausgelesen (Zeile 12) und auf der Webseite angezeigt. Wichtig ist es, die erzeugten Pointer bzw. den Speicherplatz am Ende wieder freizugeben, was in Zeile 17 passiert.

```
1  const limitStr = input.value;
2  const limit = toNumb(limitStr);
3
4  // Write empty array into memory, for the C-Code to access
5  // and create pointer to reference it
6  const ptr = writeToMem(limit);
7
8  // Call function from C-Code
9  calcFib(limit, ptr);
10
11 // Retrieve values from memory
12 const resArr = readFromMem(ptr, limit);
13
14 displayResult(resArr);
15
16 // Important to free the reserved address space form memory
17 Module._free(ptr);
18 resetInput();
```

### 5.2.3 Ergebnis

Das Ergebnis der Programmierung lässt sich in Abbildung 5.4 betrachten. Zu sehen ist ein Ausschnitt aus der erstellten Webseite, die dem Nutzer die Möglichkeit bietet eine Zahl einzugeben, bis zu welcher die Fibonacci Folge berechnet werden soll. In diesem Fall sollte bis 5 berechnet werden. Das Ergebnis der Operation sieht man weiter unten im Bild, wo alle Werte der Folge bis zur fünften Stelle aufgelistet sind.

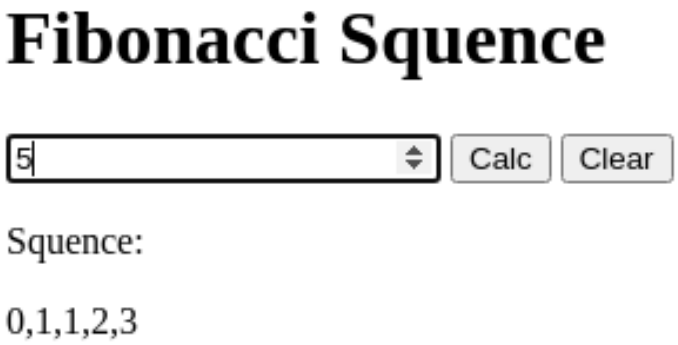


Abbildung 5.4: Ergebnis Fibonacci

## 5.3 Anwendung: Einfacher Taschenrechner

Der Taschenrechner bietet die Möglichkeit 2 Zahlen anzugeben, auf die entweder Addition, Subtraktion, Multiplikation oder Division angewandt werden kann.

### 5.3.1 Ziel und Aufbau der Anwendung

Ziel dieses Programms ist es zu demonstrieren, wie Funktionen aus dem C-Code mittels der Helferfunktion `ccall` ausgeführt werden können und wie man Funktionen aus der `emscripten.h` Bibliothek in C nutzen kann, um Funktionen durch die Kompilierung behalten zu können. Der Projektaufbau der Anwendung ist in der Abbildung 5.5 einsehbar. Das Verzeichnis *src/* enthält die Datei calculator.c, welche die Berechnungsfunktionen implementiert. Das Verzeichnis *web/* wiederum enthält eine HTML Datei zum Anzeigen der Webseite, eine Javascript Datei namens app.js, welche das Verhalten der Website implementiert als auch die entsprechenden WebAssembly Funktionen aufruft. Zuletzt befindet sich noch das Unterverzeichnis *build/* in web, das ein Shell-Skript mit entsprechendem Compiler-Aufruf beinhaltet sowie die dadurch generierten Dateien.

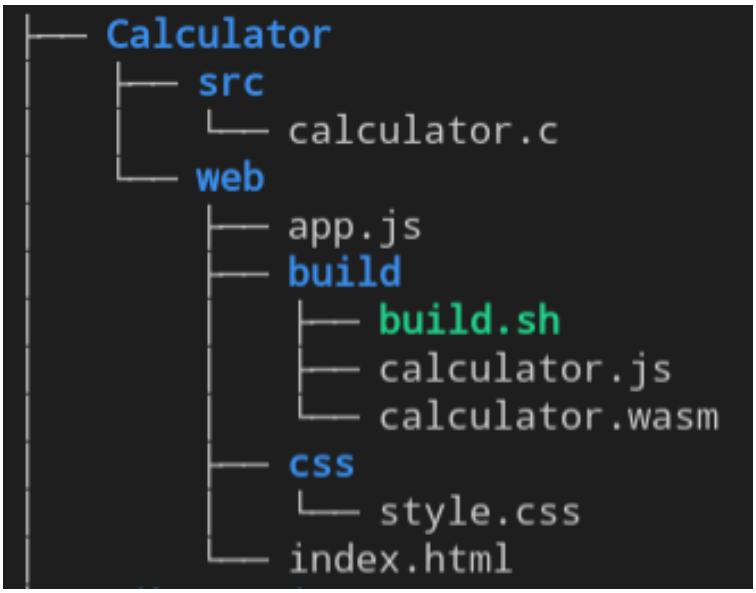


Abbildung 5.5: Aufbau HelloWorld

### 5.3.2 Umsetzung

**Native Programmierung**

Da der Taschenrechner die grundlegenden Rechenarten Addition, Subtraktion, Multiplikation und Division beherrschen soll, müssen diese in C implementiert werden. Die Implementierung dieser Funktionen ist in den meisten Programmiersprachen sehr ähnlich, weshalb aus Gründen der Übersichtlichkeit im folgenden Codebeispiel nur die Implementierung der Addition zu sehen ist. Zu beachten ist jedoch, was in Zeile 9 steht. Dieser Ausdruck, den die `emscripten.h` Datei bereitstellt, gibt dem Compiler an, dass diese Funktion für die spätere Verwendung noch benötigt wird und nicht als sogenannter *Dead Code* eliminiert werden soll. Dies stellt eine Alternative zum vorherigen Anwendungsbeispiel dar, in dem ebenfalls Funktionen erhalten und exportiert werden mussten. Statt der direkten Angabe im Code griff man stattdessen auf die Kommandozeilenoption `-s EXPORTED_FUNCTIONS`[33] zurück.

```
1  # include <stdio.h>
2  # include <emscripten.h>
3
4  EMSCRIPTEN_KEEPALIVE
5  int main(void) {
6      return 0;
7  }
8
9  EMSCRIPTEN_KEEPALIVE
10 float addition(float numb1, float numb2) {
11     return numb1 + numb2;
12 }
```

**Kompilierung**

Durch die zusätzliche Angabe in Zeile 9 verkürzt sich auch der Aufruf des Compilers, wie im folgenden Code aus der Datei build.sh zu sehen ist. Auch in diesem Fall wird nur eine WebAssembly Datei und eine Javascript Datei namens calculator.js und calculator.wasm erzeugt.

```
1 #!/bin/bash
2 emcc ../../src/calculator.c \
3 -o calculator.js \
4 -s NO_EXIT_RUNTIME=1 \
5 -s EXTRA_EXPORTED_RUNTIME_METHODS='["ccall"]'
```

**Web-Programmierung**

Nach der erfolgreichen Kompilierung der C-Datei muss als Nächstes die erzeugte calculator.js Datei und app.js in die Webseite, bzw. in die index.html, eingebunden werden. Anschließend muss die Datei app.js implementiert werden. Sie enthält die Zuweisung von EventListenern für die in der index.html definierten Buttons. Wenn ein solcher Button gedrückt wird, soll mittels der Funktion `ccall` die bereits programmierte Funktion aus dem C-Code zur Berechnung ausgeführt werden. Da die vier möglichen Aufrufe sehr ähnlich sind, wurde für diesen Fall eine Javascript Funktion geschrieben, die als Parameter die jeweilige Rechenmethode und ein Array aus den zu verrechnenden Zahlen erhält. Der Name der Funktion lautet `calc` und lässt sich im folgenden Beispiel genauer untersuchen.

```
1 function calc(method, numArr) {
2     const result = Module.ccall(
3         method,
4         'number',
5         ['number', 'number'],
6         [numArr[0], numArr[1]]
7     );
8     return result;
9 }
```

Wie im obigen Code zu sehen ist, findet in Zeile 2 der eigentliche Funktionsaufruf statt. Mittels der Methode `ccall` wird die entsprechende in C programmierte Funktion aufgerufen und das Ergebnis der Operation in der Variable `result` gespeichert.
Die Verwendung von `calc` in einem EventListener zeigt das folgende Beispiel. Beim Klick auf den Additionsbutton sollen die Summanden aus der Weboberfläche ausgelesen werden und der entsprechende Funktionsaufruf von `addition` (siehe erstes Codebeispiel) stattfinden.

```
1  function eventHandler(calcMethod) {
2      if (wasmLoaded) {
3          const numArr = getInput();
4          const result = calc(calcMethod, numArr);
5          displayResult(result);
6      } else {
7          console.warn(‘Can not execute function ${calcMethod},
8                                       because WebAssembly is not loaded‘)
     ;
9      }
10 }
11
12 buttonAdd.addEventListener('click', () => {
13     eventHandler('addition');
14 });
```

Zeile 11 des obigen Codeausschnitts beschreibt die Zuweisung eines Klick-Listeners für den Additionsbutton, der in diesem Fall die Funktion `eventHandler` mit dem Parameter `addition` ausführen soll. Die Funktion `eventHandler` ist ab Zeile 1 zu sehen. Sie nimmt als Parameter die Berechnungsmethode entgegen und managt die Beschaffung der Summanden aus der Oberfläche, ob WebAssembly richtig geladen wurde und dass die eigentliche Berechnung ausgeführt wird, indem sie die Funktion `calc` ausführt.

### 5.3.3 Ergebnis

Das Zusammenspiel von Javascript, WebAssembly und HTML lässt sich in Abbildung 5.6 erkennen. Zu sehen ist die Darstellung des Taschenrechners im Browser, welcher das Ergebnis der Multiplikation von 6 und 8 anzeigt.

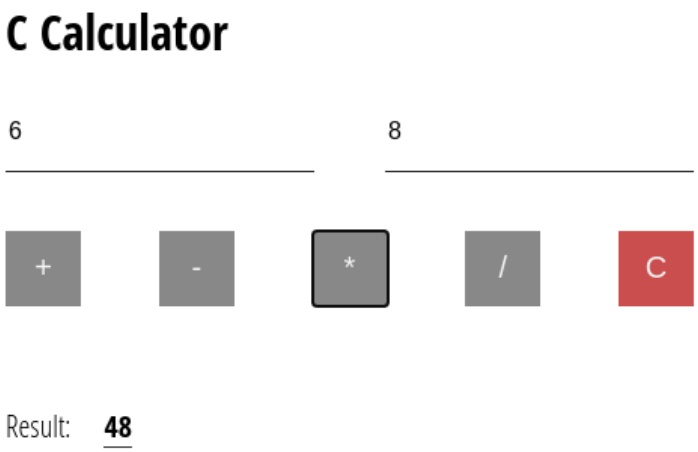


Abbildung 5.6: Ergebnis Taschenrechner

## 5.4 Anwendung: Palindrom Test

Palindrom Test ist eine Anwendung, um prüfen zu können, ob es sich bei einem Wort um ein Palindrom handelt (das Wort kann rückwärts genau gleich wie vorwärts gelesen werden). Dabei kann der Benutzer auf einer Weboberfläche das gewünschte Wort eingeben und auswerten lassen. Das Ergebnis wird ebenfalls auf der Oberfläche angezeigt.

### 5.4.1 Ziel und Aufbau der Anwendung

Ziel der Anwendung ist es, den Umgang mit komplexeren Datentypen wie Strings unter WebAssembly beispielhaft zu demonstrieren. Dabei kommen erneut die Helferfunktionen `cwrap`, `mallo` und `free` zum Einsatz.

Der Aufbau dieses Projektes gleicht den bisherigen: Es gibt zwei Verzeichnisse, die einmal den notwendigen C-Code enthalten (*src/*) und zum anderen die für das Web notwendigen HTML und Javascript Dateien (*web/*). Anders als bisher wurde die Logik zur Bestimmung eines Palindroms in der palindrom.c Datei implementiert, die mittels einer Header-Datei in der main.c importiert werden kann.

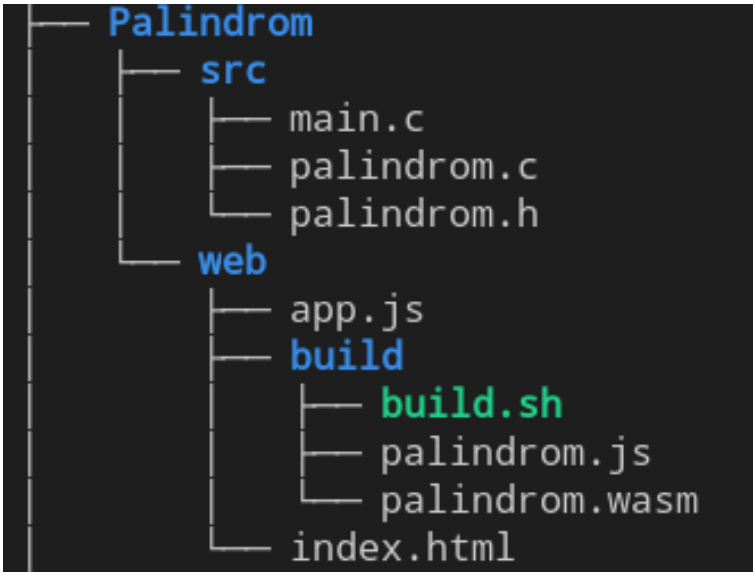


Abbildung 5.7: Aufbau HelloWorld

### 5.4.2 Umsetzung

**Native Programmierung**

Für die Umsetzung dieser Anwendung ist es als Erstes nötig eine Funktion in C zu schreiben, die prüfen kann, ob es sich bei einem String um ein Palindrom handelt. Der folgende Codeausschnitt aus palindrom.c zeigt, wie diese Funktion implementiert wurde.

```
1  // Datei: palindrom.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int isPalindrom(char *str_ptr, int str_length)
6  {
7      int index1 = 0;
8      int index2 = str_length - 1;
9      while (index1 < str_length / 2) {
10         if (str_ptr[index1] != str_ptr[index2])
11             return 0;
12         else {
13             index1++;
14             index2--;
15         }
16     }
17     return 1;
18 }
```

In Zeile 4 wird die Funktion `isPalindrom` definiert, die als Parameter einen Character Pointer und die Länge des Strings erhält. In den nächsten Zeilen wird jeweils eine Referenz auf den Anfang und das Ende des Strings bzw. Character Arrays gesetzt und anschließend in einer Schleife geprüft, ob die gegenüberliegenden Buchstaben übereinstimmen. Sollte dies

der Fall sein, gibt die Funktion den Werte 0 zurück, falls nicht, den Wert 1.

**Kompilierung**

Nachdem der C-Code geschrieben wurde, gilt es nun, ihn entsprechend in WebAssembly zu kompilieren. Da für die weitere Programmierung in Javascript mit dem Speicher interagiert werden soll, müssen die Methoden `malloc` und `free` exportiert werden. Zudem soll die Palindrom-Funktion namens `isPalindrom` genutzt werden können, was wiederum `cwrap` oder `ccall` erfordert. Die folgenden Zeilen aus der Datei build.sh verdeutlichen, wie gemäß dieser Anforderungen kompiliert werden kann.

```
# Datei: build.sh

emcc \
../../src/palindrom.c \
-o palindrom.js \
-s WASM=1 \
-s EXPORTED_FUNCTIONS='["_isPalindrom", "_malloc", "_free"]' \
-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap", "getValue", "setValue"]'
```

**Web-Programmierung**

Durch das Ausführen dieses Befehls werden die Dateien palindrom.js und palindrom.wasm erzeugt. Um diese nun verwenden zu können, muss als Erstes das Skript palindrom.js in die HTML Datei index.html eingebunden werden, was durch folgende Zeile erfüllt wird:
`<script src="./build/palindrom.js"></script>`
Der nächste Schritt besteht darin, die durch Emscripten erzeugte Javascript Schnittstelle zum WebAssembly Code zu nutzen. Dafür wird die Datei app.js eingesetzt, die ebenfalls in index.html eingebunden werden muss. Die Datei app.js kann nun auf die in palindrom.js erzeugten Funktionen und Objekte zugreifen. So kann geprüft werden, ob WebAssembly vollständig und komplett geladen wurde:

```
// Datei: app.js

let isPalindrom = null;
let wasmReady = false;
Module.onRuntimeInitialized = () => {
    wasmReady = true;
    isPalindrom = Module.cwrap( "isPalindrom",
                                "number",
                                ["number", "number"]);
}
```

Falls dem so ist, wird die Variable `wasmReady` auf `true` gesetzt, welche über das ganze Skript verfügbar (im Scope bleibt). Im selben Zug wird auch `cwrap` genutzt, um eine Referenz auf die C-Funktion `isPlindrom` in der gleichnamigen Javascript Variablen zu speichern. Um die Funktion `isPalindrom` ausführen zu können, müssen zuerst die Parameter entsprechend vorbereitet werden. Dies gilt vornehmlich für den String, da dieser nicht in seiner jetzigen Form der Funktion übergeben werden kann. Stattdessen muss er zuerst umgewandelt und manuell in den Speicher vom WebAssembly geschrieben werden. Diesen Prozess bzw. diese Funktion zeigt der folgende Codeausschnitt zu `writeToMem`:

```
1  // Datei: app.js
2
3  function writeToMem(str = "") {
4      let ptr = 0;
5      if (str.length > 0) {
6          const len = str.length;
7
8        const convertedStr = new Uint8Array(intArrayFromString(str));
9          ptr = Module._malloc(len * convertedStr.BYTES_PER_ELEMENT);
10
11         Module.HEAP8.set(convertedStr, ptr);
12     }
13     return ptr;
14 }
```

Die Funktion zeigt, wie ein Javascript String in den WebAssembly Speicher geschrieben werden kann. Dazu muss als Erstes die Länge des Strings gespeichert werden (Zeile 4), da dies in C nur herausgefunden werden kann, wenn der String tatsächlich NULL-terminiert ist(C von A bis Z[81], Kapitel 14.12). In Zeile 6 werden mehrere Befehle ausgeführt. Zunächst wird der ursprüngliche String in einzelne Zeichen zerlegt und diese zu ihrer numerischen Darstellung konvertiert. Dies geschieht durch den Aufruf der Funktion `intArrayFromString`, welche von Emscripten generiert wird. Als Nächstes wird dieses Array als Eingabe für die Generierung eines typisierten Uint8Arrays genutzt. Es wird das Format Uint8 genutzt, da ein Character genau 8 Bits, bzw. 1 Byte groß ist und der Wertebereich dieses Datentyps zwischen 0 und 255 liegt. Damit sind alle möglichen ASCII Characters abgedeckt und können in das Array aufgenommen werden. Zudem schließt man somit negative Werte aus, da ASCII Werte üblicherweise im Bereich von 0 bis 128 liegen. Nachdem der ursprüngliche String nun als Uint8Array konvertiert wurde, ist es möglich, ihn mittels `malloc` in den Speicher zu schreiben und einen Pointer auf die erste Stelle zu erhalten (Zeile 7 und 9).

Im letzten Schritt müssen die bereits vorgestellten Funktionen aus der Datei app.js einem EventListener zugewiesen werden, um sie im Falle des entsprechenden Events ausführen zu können. Der nachfolgende Code zeigt diesen Prozess:

```
1  // Datei: app.js
2
3  btSubmit.addEventListener("click", ev => {
4      if (wasmReady && isPalindrom != null) {
5
6          // retrieve input value
7          const str = inputBox.value.toLowerCase();
8          const ptr = writeToMem(str);
9
10         const result = isPalindrom(ptr, str.length);
11
12         Module._free(ptr);
13
14         displayResult(result);
15     }
16
```

```
17      inputBox.value = "";
18  });
```

Dem Button btSubmit wird ein EventListener zugewiesen, welcher auf ein Klick Event reagiert. Innerhalb des Listeners wird der Code definiert, der in diesem Fall ausgeführt werden soll. Nachdem der vom Nutzer eingegebene String in Zeile 7 aus einem Input-Feld geholt wird, wird die Funktion `writeToMem` ausgeführt und ein Pointer erzeugt. Dieser Pointer wird zusammen mit der Länge des Strings als Parameter der Funktion `isPalindrom` in Zeile 10 übergeben. Um die manuell reservierten Speicherstellen wieder freizugeben, wird in Zeile 12 die Funktion `free` auf den Pointer ausgeführt.

### 5.4.3 Ergebnis

Ergebnis dieses Projektes ist die in Abbildung 5.8 und Abbildung 5.9 gezeigte Anwendung. In den genannten Abbildungen werden beispielhaft zwei Strings auf Palindrom-Eigenschaft geprüft. In Abbildung 5.8 wurde der String *Tacocat* eingegeben, der von der Anwendung erfolgreich als Palindrom erkannt wird. Die nächste Abbildung zeigt dazu ein Negativbeispiel, indem der String *palindrom* geprüft wird.

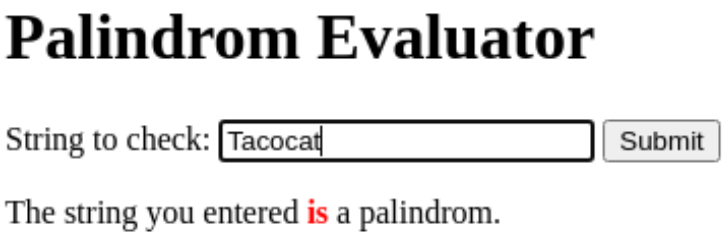


Abbildung 5.8: Ergebnis echtes Palindrom

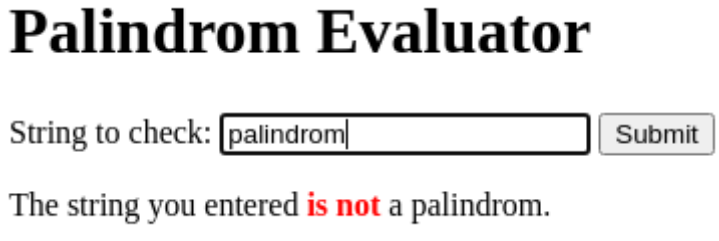


Abbildung 5.9: Ergebnis falsches Palindrom

## 5.5 Anwendung: Bearbeiten von Textdateien im Browser

Mit der Anwendung *TextfileEditor* ist es möglich, Textdateien auf dem Computer des Nutzers im Browser bearbeiten und speichern zu können. Das Hochladen der Datei auf einen entfernten Server ist dabei nicht nötig.

### 5.5.1 Ziel und Aufbau der Anwendung

Der Aufbau dieser Anwendung ähnelt den vorangegangenen Projekten und lässt sich in der Abb. 5.10 genauer betrachten. Erneut ist die Struktur in die Verzeichnisse *src/*, für die native Entwicklung unter C und *web/*, für die Webentwicklung mit WebAssembly und Javascript aufgeteilt. Neben dem *build/* Ordner ist in diesem Projekt auch ein *css/* Verzeichnis vorhanden, welches die Datei style.css enthält und von der index.html genutzt wird.

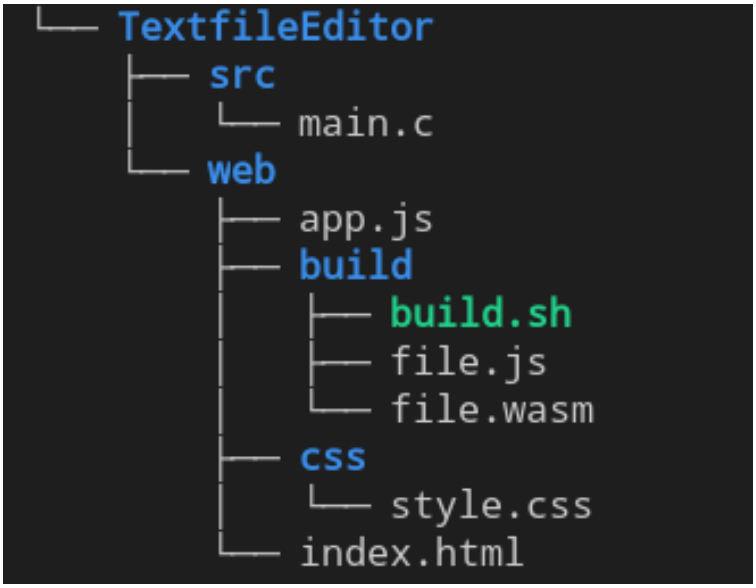


Abbildung 5.10: Aufbau HelloWorld

### 5.5.2 Umsetzung

**Native Programmierung**

Anforderung an das C-Programm ist es, eine Funktion zur Verfügung zu stellen, welche eine gegebene Datei im Schreibmodus öffnen kann und den bereits vorhandenen Inhalt entweder ersetzen oder erweitern kann. Dazu wurde die Funktion `append_to_file` geschrieben, welche sich in der Datei main.c befindet. Der folgende Code verdeutlicht, wie die genannten Anforderungen in C umgesetzt wurden.

```
1 int append_to_file(char *file_name, char *content)
2 {
3     FILE *file = fopen(file_name, "w");
```

```
4
5       if (file == NULL) {
6           fprintf(stderr, "File could not be opened ...");
7           return 1;
8       }
9
10      //fprintf(file, "%s", content);
11      fputs(content, file);
12      fputs("\n", file); // make a new line
13
14      fclose(file);
15      return 0;
16  }
```

Der Funktion `append_to_file` werden die Parameter `file_name` und `content` übergeben, bei denen es sich um Pointer des Typs Character handelt. Mittels der Funktion `fopen` aus der Standard Input/Output Bibliothek aus C kann die Datei über ihren Namen im Schreibmodus (symbolisch als *w* angegeben) geöffnet werden (Zeile 3). Anschließend kann in Zeile 11 der Inhalt, welcher der Funktion übergeben wurde, über `fputs` in die Datei geschrieben werden. Dabei wird jedoch kein neuer Inhalt der Datei angehängt. Stattdessen wird der bisherige Inhalt durch die in `content` enthaltenen Daten überschrieben. Dies liegt am verwendeten Zugriffsmodus aus Zeile 3. Wenn der Inhalt erfolgreich in die Datei geschrieben wurde, wird 0 als Rückgabewert ausgegeben.

**Kompilierung**

Nachdem der C-Code implementiert wurde, muss er an dieser Stelle mittels Emscripten in das richtige Zielformat kompiliert werden. Dabei soll die Funktion `append_to_file` erhalten/exportiert werden sowie die Helferfunktionen `malloc`, `free`, `cwrap`, `getValue` und `setValue`. Zudem soll sichergestellt werden, dass das WebAssembly Filesystem unterstützt wird. Um die Datei main.c dementsprechend zu kompilieren, wird die Datei build.sh, welche nachfolgend zu sehen ist, ausgeführt.

```
1  #!/usr/bin/env bash
2
3  emcc \
4  ../../src/main.c \
5  -o file.js \
6  -s WASM=1 \
7  -s EXPORTED_FUNCTIONS='["_main", "_append_to_file", "_malloc", "_free"]' \
8  -s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap", "getValue", "setValue"]' \
9  -s FORCE_FILESYSTEM=1 \
10 -s ALLOW_MEMORY_GROWTH=1
```

Die Option `-s ALLOW_MEMORY_GROWTH=1`[33] ist in diesem Fall nötig und stellt sicher, dass der virtuelle Speicher von WebAssembly groß genug ist, bzw. skalieren kann, um mögliche Dateien in sein Filesystem aufnehmen zu können. Mithilfe dieser Option können im Speicher bis zu 2 GB reserviert werden.

**Web-Programmierung**

Um die vom Compiler neu erzeugten Dateien file.js und file.wasm nutzen zu können, muss file.js im ersten Schritt der Webentwicklung in die Datei index.html, zusammen mit der Javascript Datei app.js, eingebunden werden. Dadurch können der binäre WebAssemblycode und die Javascript Dateien geladen werden.

Bevor in der Datei app.js nun mit dem Dateisystem gearbeitet werden kann, muss geprüft werden, ob WebAssembly erfolgreich geladen wurde. Dazu dient folgender Codeausschnitt aus app.js:

```
1  let appendToFile = null;
2
3  let fileContent = "";
4  let wasmLoaded = false;
5  Module.onRuntimeInitialized = () => {
6      wasmLoaded = true;
7      appendToFile = Module.cwrap(  "append_to_file",
8                                    "number",
9                                    ["number", "number"]);
10     console.log("WASM ready ...");
11 }
```

Der oben gezeigte Code aus dem Skript definiert als Erstes drei Variablen mit globalen Scopes, was bedeutet, dass sie an jeder Stelle innerhalb des Skripts verfügbar sind. In Zeile 3 wird die Variable namens `fileContent` mit einem leeren String initialisiert und dient im späteren Code dazu, den Inhalt einer Datei zu halten. Anschließend werden ab Zeile 6 die entsprechenden Werte für die Variablen gesetzt für den Fall, dass WebAssembly erfolgreich geladen wurde. Die Variable `appendToFile` erhält hierbei eine Funktionsreferenz auf die Funktion `append_to_file` mittels `cwrap`.

Nachdem nun mittels `wasmReady` die Verfügbarkeit von WebAssembly überprüft werden kann und die Funktion `appendFile` nun zur Verfügung steht, geht es als Nächstes darum, die Parameter entsprechend den Anforderungen der C-Funktion `append_to_file` vorzubereiten. Die Funktion erhält zwei Parameter als Character Arrays, sprich Strings. Diese Parameter sind der Name der Datei und der Inhalt der Datei. Durch die nachfolgend abgebildete Funktion aus app.js soll ein String in numerische Werte umgewandelt und als Array in den WebAssembly Speicher geschrieben werden.

```
1  const setStrToMem = (content) => {
2      const length = content.length;
3      const uint8Str = new Uint8Array(intArrayFromString(content));
4      const ptr = Module._malloc(length * uint8Str.BYTES_PER_ELEMENT);
5      Module.HEAPU8.set(uint8Str, ptr);
6      return ptr;
7  }
```

Die Funktion wurde als ES6 Arrow-Function geschrieben und besitzt daher nicht das Schlüsselwort `function`. Stattdessen wird die Arrow-Function in eine konstante Variable, welche den Funktionsnamen repräsentiert, gespeichert und ist dadurch über diesen Namen ausführbar. Der Parameter `content` stellt einen beliebigen Javascript String dar. In Zeile 3 findet

die Umwandlung des Strings in ein typisiertes Uint8Array statt. Anschließend wird durch die Länge des Arrays und der Größe seiner Werte (da es sich um einzelne Characters handelt, ist jedes Element genau 1 Byte groß) ein Pointer auf die erste Speicherstelle berechnet. An die Stelle des Pointers wird in Zeile 5 das typisierte Array in den Speicher geschrieben.

Ab dieser Stelle könnte zwar der Funktionsaufruf von `appendFile` stattfinden, da die Parameter nun übergeben und im C-Code ausgelesen werden können. Allerdings könnte die Datei nicht bearbeitet werden, da WebAssembly unter diesem Namen keine Datei finden und öffnen kann. Die nächste benötigte Funktion muss folglich die vom Nutzer ausgewählte Datei in das virtuelle Dateisystem von WebAssembly laden. Der folgende Code zeigt eine Funktion, welche neue Dateien in das Dateisystem laden kann.

```
1  const FS_createFile = (fileName, content) => {
2      // create file with read and write permissions ...
3      try {
4          FS.createDataFile("/", fileName, content, true, true);
5      } catch (ex) {
6          console.warn( "Error occured, " +
7                        " possible, that file already exists ...");
8      }
9
10 }
```

Die Funktion `FS_createFile` nimmt als Parameter den Namen der Datei und ihren Inhalt als Strings entgegen. Anschließend wird in Zeile 4 mittels der Funktion `createDataFile`, die als Teil der Filesystem API über das FS Objekt verfügbar ist, eine neue Datei mit dem angegebenen Namen und Inhalt in das virtuelle Dateisystem geladen. Die Funktion `createDataFile` nimmt dabei als Parameter den Pfad entgegen, auf welchem die Datei gespeichert werden soll (in diesem Fall '/'), als weiteren Parameter außerdem den Namen und Inhalt der Datei. Die letzten beiden Parameter, die in diesem Fall auf true gesetzt sind, bestimmen, ob die Datei gelesen und beschrieben werden darf.

Auf dem bisherigen Stand ist es nun möglich, die Funktion `appendFile` mit passenden Parametern aufzurufen und die neu erzeugte Datei im virtuellen System zu bearbeiten. Was noch fehlt, um die Anwendung nutzbar zu machen, ist eine Möglichkeit, die vorgenommenen Änderungen bzw. die veränderte Datei zu erhalten. Auch für diese Aufgabe kann die Filesystem API eingesetzt werden. Durch die Funktion `readFile` kann eine Datei aus dem WebAssembly Dateisystem ausgelesen werden, was im folgenden Code innerhalb der Funktion `FS_getContent` demonstriert wird. Zusätzlich zeigt der Code aus der app.js Datei, wie man die daraus erhaltene Datei dem Nutzer wieder zur Verfügung stellt.

```
1  const FS_getContent = (fileName) => {
2      const content = FS.readFile(fileName, { encoding: "utf8" });
3
4      const url = URL.createObjectURL(
5          new Blob(
6              [content],
7              { type: " text/plain;charset=utf-8" })
```

```
8       );
9       return url;
10  }
11
12  const download = (url) => {
13      var element = document.createElement('a');
14      element.setAttribute('href', url);
15      element.setAttribute('download', "WASM_EDITED");
16      element.style.display = 'none';
17      document.body.appendChild(element);
18      element.click();
19      document.body.removeChild(element);
20      window.URL.revokeObjectURL(url);
21  }
```

Wie bereits beschrieben, ermöglicht die Funktion `FS_getContent` durch Einsatz von `readFile` den Inhalt einer Datei auszulesen. Das Ergebnis dieser Operation ist entweder ein String oder ein Uint8Array. Welcher Typ zurückgegeben wird, hängt von der Codierung der Datei ab. Im Falle von UTF-8 ist das Ergebnis ein String, im Falle von binärer Codierung ein Uint8Array. In diesem Beispiel enthält die Variable `content` in Zeile 2 einen String. Aus diesem String wird in den nächsten Zeilen eine URL erzeugt[63], die eine Textdatei mit einer UTF-8 Kodierung enthält. Dazu wird zuerst eine neue Blob-Instanz mit dem Inhalt und dem Dateityp text/plain erzeugt und diese dann als Parameter für die URL Funktion `createObjectURL` eingesetzt. Die dadurch erzeugte URL wird schließlich in Zeile 9 zurückgegeben.

Die zweite Funktion, die im obigen Code zu sehen ist, ermöglicht dem Benutzer, die Datei über die zuvor erzeugte URL zu downloaden. Dabei wird ein HTML Anchor-Tag, auch Link genannt, erzeugt und diesem als Hyperlink die URL zugewiesen. Dieser Link ist auf der Website nicht sichtbar, wird aber kurzzeitig dem DOM(Document Object Model) hinzugefügt (Zeile 16 und 17). Anschließend wird in Zeile 18 manuell das Click-Event ausgelöst, wodurch der Download-Prozess gestartet wird.

Die implementierten Funktionen müssen im letzten Schritt noch ihren zugehörigen Event-Handlern zugewiesen werden. Das erste Element, dem ein EventListener zugewiesen wird, ist ein Input-Element aus der index.html. Dieses Input-Element ermöglicht dem Nutzer, eigene Dateien aus seinem System auszuwählen und auf die Website hochzuladen. Die Implementierung des EventHandlers ist im folgenden Codeausschnitt zu sehen:

```
1   input.onchange = ev => {
2           const file = input.files[0];
3           fileName = file.name;
4
5           const fileReader = new FileReader();
6           fileReader.readAsText(file);
7
8           fileReader.onload = () => {
9               if (!wasmLoaded) {
10                  return;
11              }
```

```
12              const fileContent = fileReader.result;
13              contentHtml.innerText = fileContent;
14
15              FS_createFile(fileName, fileContent);
16
17              // Enable to make own changes to the document ...
18              contentHtml.setAttribute("contenteditable", "true");
19          }
20      }
```

Wird dem Input-Element eine Datei zugewiesen, wird das Change-Event ausgeführt, bzw. die dafür hinterlegte Funktion. Innerhalb der Funktion wird mit Hilfe einer FileReader[58] Instanz die entsprechende Datei ausgelesen und dadurch der Inhalt der Datei in der Variable `fileContent` in Zeile 12 gespeichert. In Zeile 15 wird die Funktion `FS_createFile` ausgeführt, um die Datei auch im virtuellen Speicher anzulegen.

Das zweite Element, dem ebenfalls ein EventListener zugewiesen wird, ist ein Button, um die vorgenommenen Änderungen speichern zu können. Der unten zu sehende Code beschreibt die Implementierung in der Datei app.js:

```
1  btSave.addEventListener("click", ev => {
2          if (fileName.length > 1) {
3              const fileNamePtr = setStrToMem(fileName);
4              const editedContent = contentHtml.innerText;
5              const contentPtr = setStrToMem(editedContent);
6
7              appendToFile(fileNamePtr, contentPtr);
8              const url = FS_getContent(fileName);
9              download(url);
10
11             Module._free(fileNamePtr);
12             Module._free(contentPtr);
13             // FS_deleteFile(fileName); => not supported
14
15             fileName = "";
16             contentHtml.innerHTML = "...";
17             resetInput();
18         } else {
19             alert("You have to select a file first ...");
20         }
21     });
```

Im Fall eines Click-Events wird der Name der Datei und ihr Inhalt in den Speicher geschrieben und jeweils der Pointer auf die entsprechende Speicherstelle gespeichert (siehe Zeile 3 und 5). Anschließend wird die Funktion `appendToFile` in Zeile 7 mit den beiden Pointern als Parameter aufgerufen. Nach dem Bearbeiten der Datei durch WebAssembly wird der Inhalt der Datei ausgelesen und in eine ObjectURL[63] umgewandelt, was durch die Funktion `FS_getContent` in Zeile 8 geschieht. Die generierte URL wird nun in der nächsten Zeile der download Funktion übergeben und so dem Nutzer zum Herunterladen angeboten. Im letzten Schritt wird der Speicher durch den Aufruf von `free` freigegeben und die Anwendung

entsprechend aufgeräumt.

### 5.5.3 Ergebnis

Das Ergebnis des Projekts lässt sich in der Bilderstrecke der Abbildungen 5.11 bis 5.14 betrachten. Die Bilder skizzieren beispielhaft das Auswählen, Bearbeiten und Speichern der Datei namens **text.txt**.

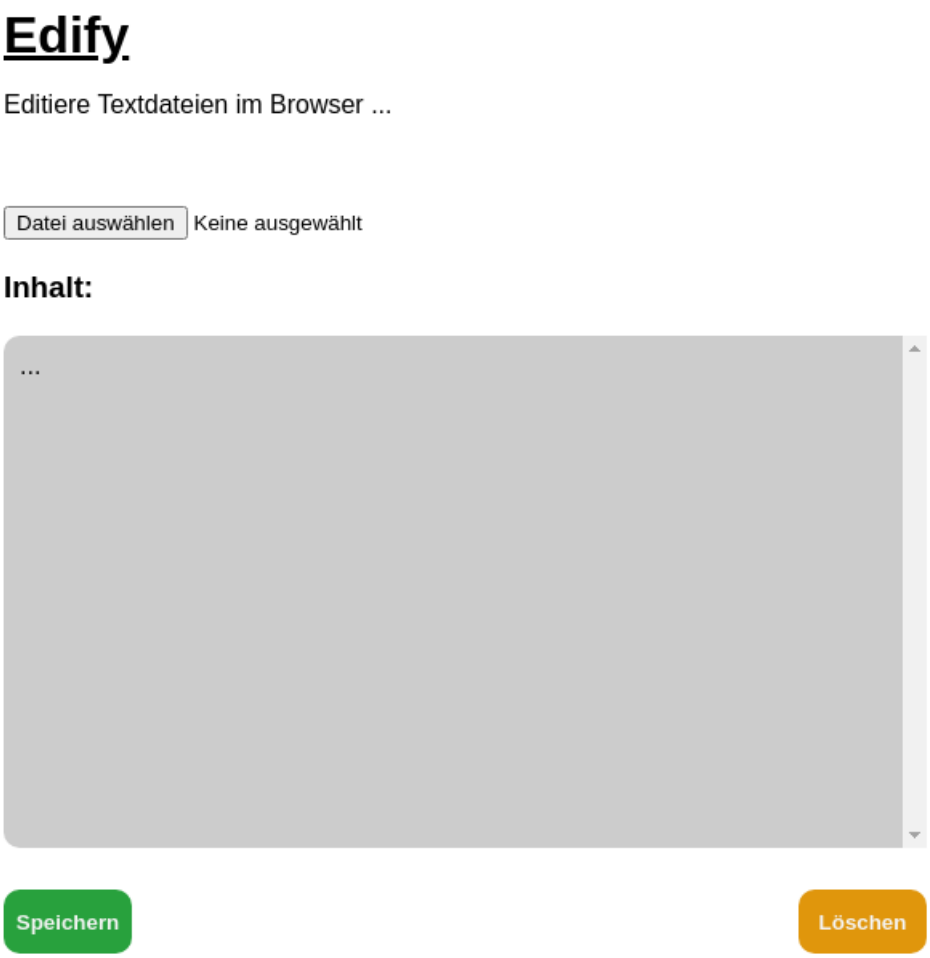


Abbildung 5.11: Ergebnis Textbearbeitung

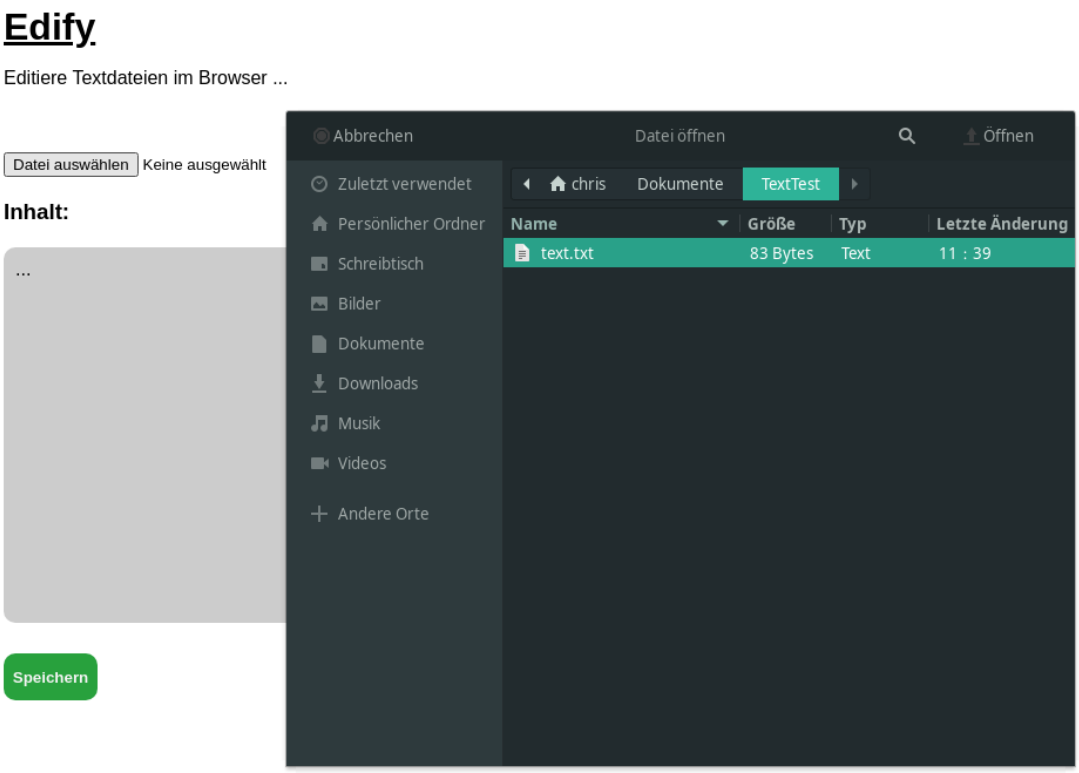


Abbildung 5.12: Ergebnis Textbearbeitung, auswählen einer Datei

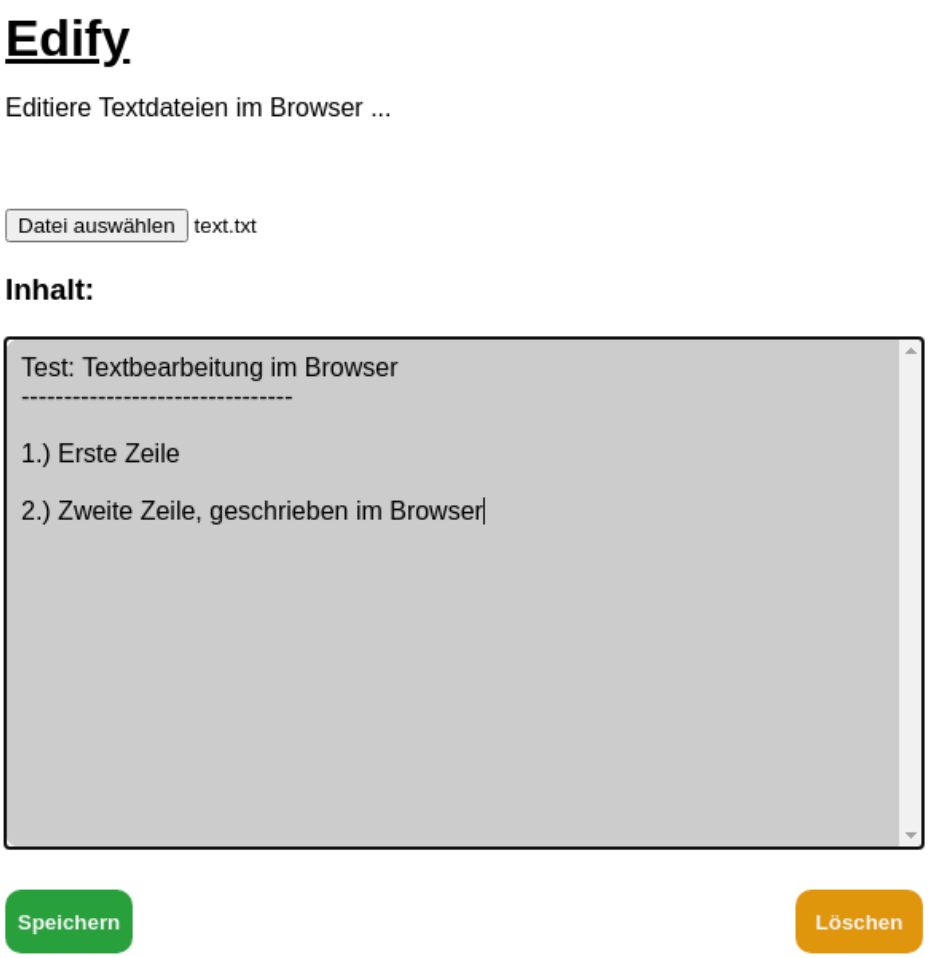


Abbildung 5.13: Ergebnis Textbearbeitung, editieren einer Datei

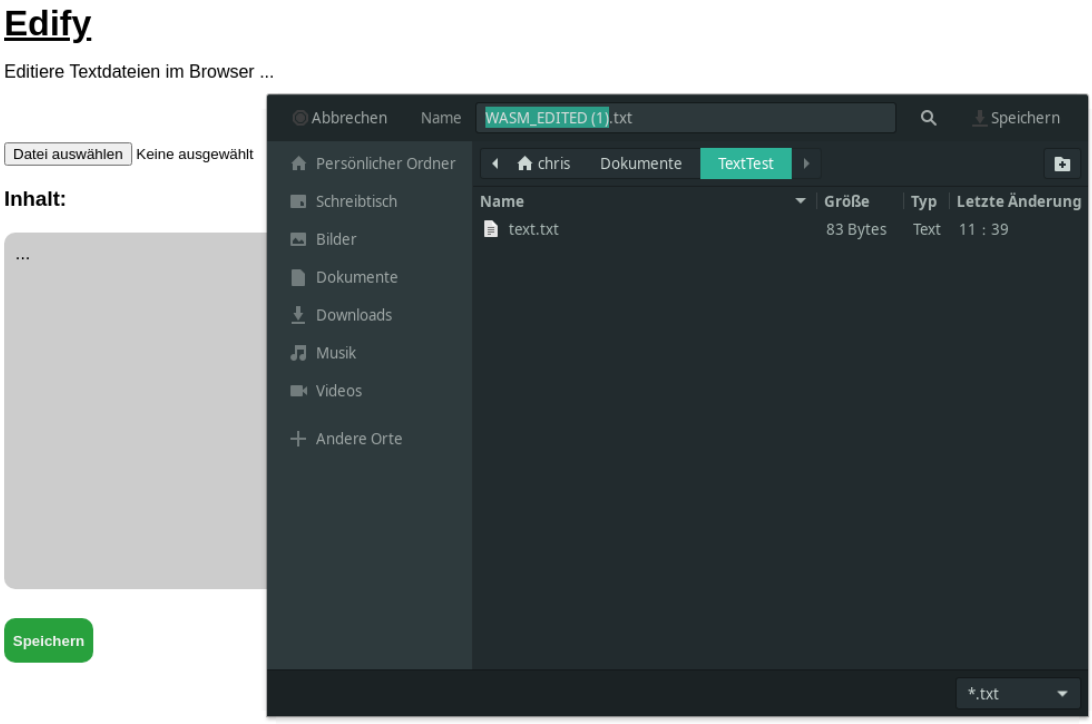


Abbildung 5.14: Ergebnis Textbearbeitung, speichern einer Datei

# 6 Bewertung

In diesem Kapitel werden die Informationen, die durch die bisherigen Kapitel vermittelt wurden, auf ihre Vor- und Nachteile überprüft, um so eine Bewertung für das Thema WebAssembly zu ermöglichen. Die Bewertung teilt sich in drei Unterkapitel auf. Dabei werden das generelle Konzept zur Verwendung von WebAssembly, der in dieser Arbeit genutzte WebAssembly Compiler sowie die generellen Programmiermöglichkeiten genauer betrachtet.

## 6.1 Bewertung des Konzepts von WebAssembly

Das Konzept von WebAssembly besteht darin, durch das Hinzufügen einer Abstraktionsebene, Bytecode auf allen Browsern ausführbar zu machen, um somit die Funktionalität und Möglichkeiten von Javascript zu erweitern[75]. Durch diesen Ansatz ist es möglich, höhere Sprachen wie C/C++ und Rust in das Zielformat WebAssembly zu kompilieren[80] und so auf Webanwendungen ausführen zu können. Ziel ist es dabei, bisherige Performanceprobleme seitens Javascript[47] zu beheben, durch eine virtuelle Umgebung Sicherheit zu bieten und durch Portabilität Plattformunabhängigkeit zu bieten[69].

Bereits realisierte Webanwendungen wie bspw. AutoCAD[2] oder Construct3[1] haben gezeigt, dass das Konzept von WebAssembly funktioniert und auch kommerzielle Anwendungen erfolgreich erstellt und genutzt werden können.

Auch die Möglichkeiten und der Funktionsumfang dieser Anwendungen übersteigen die der regulären Webanwendungen mit Javascript[80]. So ist bspw. die kostenlose Anwendung Sqoosh[70] in der Lage komplexe Bildbearbeitung durchzuführen, die andernfalls nur in nativen Anwendungen zu finden wären[40]. Beispiele dieser Funktionalitäten sind zahlreiche Komprimierungsverfahren in verschiedene Dateiformate wie JPG, WebP, PNG, AVIF und MozJPEG. Daneben gibt es noch weitere Optionen wie die Einstellung der Alpha-Filter und die Konvertierung in ein anderes Farbmodell (RGB zu YUV). Die Wirkung der jeweiligen Einstellungen kann dabei sogar live im Browser beobachtet werden.

Bei Betrachtung der Sicherheitsziele von WebAssembly lässt sich sagen, dass zwar einige Schwachstellen bereits gefunden wurden, diese sich aber nur auf die konkrete Implementierung des Konzepts beschränken und nicht WebAssembly an sich kompromittieren[34]. Das bedeutet, dass diese Lücken prinzipiell geschlossen und behoben werden können. Allerdings ist zu bedenken, dass es auch Sicherheitsaspekte gibt, die sich nur schwer beheben lassen, berichtet Carsten Eilers von entwickler.de[34]. So ist es bspw. möglich, mittels WebAssembly Crypto Miner besser ausnutzen zu können oder auch Schadsoftware aus C/C++ wie bspw.

Spectre 1 und 2 zu kompilieren und im Browser auszuführen[34]. Besonders die Schadsoftware Spectre ist in Hinsicht auf WebAssembly problematisch, da wie das Magazin entwickler.de berichtet, die bisherigen Schutzmaßnahmen gegen Spectre Angriffe keine Wirkung haben. Daraus zu schließen birgt WebAssembly als Technologie vor allem für Nutzer mehr Gefahren als für Entwickler bzw. Anbieter von Webanwendungen mit WebAssembly Einsatz.

Bezüglich der Portabilität auf andere Systeme gibt es bereits Software wie wasmtime[41], die es ermöglicht, WebAssembly auch außerhalb des Browser zu verwenden. Wasmtime ist bspw. eine Runtime, welche der GitHub Dokumentation zufolge fast überall eingesetzt werden kann, so bspw. auch auf Microchips oder Servern. Mit wasmtime erweitert sich auch der Umfang der bisher unterstützten Sprachen auf Python, .NET und Go[41].

Gemessen an der Realisierung der Designziele[69] WebAssemblys und den bisherigen Erfolgen lässt sich sagen, dass es sich bei WebAssembly um eine vielversprechende Technologie handelt, die zwar gewisse Risiken birgt, aber auch große Vorteile (bzgl. Effizienz, Funktionalität, Portabilität) verspricht. Ebenfalls lässt sich erkennen, dass WebAssembly besonders für bestimmte Aufgabenstellungen und Anforderungen geeignet ist (bspw. Arbeiten mit Media-Dateien oder Implementierung einer Gesichtserkennung)[80]. Hieraus lässt sich jedoch nicht ableiten, dass WebAssembly grundsätzlich für jede Webanwendung geeignet ist. Durch den Einsatz mehrerer Sprachen und die Notwendigkeit zusätzlicher Software in Form von Compilern wird die Komplexität eines Projektes zusätzlich erhöht. Es sollte daher vor dem Einsatz von WebAssembly analysiert werden, ob die Notwendigkeit für den Einsatz besteht und wenn nicht, ob der zusätzliche Aufwand gegenüber den Vorteilen von WebAssembly vertretbar ist.

## 6.2 Bewertung des Compilers

Das Angebot an Compilern für WebAssembly umfasst nicht nur die Emscripten Compiler-Toolchain, sondern noch einige Weitere. Da in diesem Projekt jedoch mit Emscripten gearbeitet wurde, beschränkt sich die Bewertung auf diesen Compiler ohne Vergleiche oder Querverweisen zu anderen. Die Bewertungskriterien beziehen sich auf die Dokumentation, den Funktionsumfang und die Nutzung.

Emscripten bietet eine große Vielfalt an Informationen bzgl. generellen Informationen[10], Anleitungen zur Installation[16] und kleine Beispiele zum Erstellen erster Projekte[20]. Darüber hinaus bietet Emscripten eine ausführliche Dokumentation der API, welche die wichtigsten Funktionen und Bibliotheken (bspw. der Filesystem API[22]) erklärt und erläutert. Zu bemängeln ist lediglich, dass die Dokumentation in mancher Hinsicht unschlüssig wirkt und neue Nutzer mit den schnellen Kontextwechseln (zwischen C/C++ und Javascript) verwirren kann. Ein Beispiel für die Unschlüssigkeit der Dokumentation ist die Verwendung der Funktionen `_malloc()` und `_free()`. Beide Funktionen werden durch Emscripten als Teil des Javascript Glue-Codes erzeugt. Beide Funktionen werden oft in Code-Beispielen innerhalb der Dokumentation eingesetzt. Dabei wird jedoch nicht erwähnt, dass es zwingend notwendig ist, beide Funktionen manuell mittels der Option

`-s EXPORTED_FUNCTIONS='["_malloc", "_free"]'` bei dem Kompilierprozess zu exportieren. Sollten die Funktionen nicht exportiert werden, werden sie zwar im generierten Glue-Code implementiert, stehen aber während der Ausführung des Codes nicht zur Verfügung. Stattdessen erhält der Nutzer folgende Fehlermeldung:

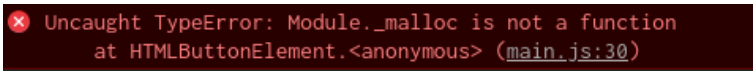


Abbildung 6.1: Fehlermeldung im Browser

Der Funktionsumfang von Emscripten bzw. der damit gelieferten emsdk umfasst mehr als nur die Kompilierung in das .wasm Format. Der Compiler bietet zahlreiche Optionen an, die über emcc –help oder auf emscripten.org[17] einsehbar sind. Dazu zählen u. a. mehrere mögliche Zielformate des Kompiliervorgangs, Komprimierungslevel, Debugging-Optionen, dynamisches Speicherwachstum sowie das Einbinden des Filesystems. All diese Optionen machen Emscripten zu einem voll Funktionsfähigen Compiler, der sogar noch weitere Tools mit sich bringt, wie in der Dokumentation auf emscripten.org erläutert[10].

Betreffend der Nutzung des Compilers lässt sich sagen, dass sich Emscripten problemlos mit der Kommandozeile bedienen lässt. Jedoch sind Aufrufe mit mehreren Optionen umständlicher, da einige Kommondozeilen-Argumente ungewöhnlich lang sind, wie z. B. `-s EXTRA_EXPORTED_RUNTIME_METHODS` oder auch `-s ALLOW_MEMORY_GROWTH=1`[33]. Wenn diese Compilieranweisungen allerdings in eine Shell-Datei ausgelagert werden, verbessert sich dabei nicht nur die Bedienbarkeit, sondern auch die Wartbarkeit und Nachvollziehbarkeit des Kompilierprozesses. Zudem ermöglicht Emscripten eine vergleichsweise einfache Entwicklung und Programmierung mit WebAssembly, da durch den Compiler viel Boilerplate-Code(notwendiger Code, der jedoch keine wesentlichen Funktionalitäten bereitstellt) und Glue-Code erzeugt wird. Gerade der Glue-Code in Form von Javascript ist besonders hilfreich, da hier viele Funktionen implementiert werden, die das Arbeiten mit WebAssembly vereinfachen. Beispiele hierfür sind `_malloc` und `_free` für die Interaktion mit dem Speicher, das Event `onRuntimeInitialized`, die Konvertierung von numerischen und alphabetischen Werten, das Module Objekt selbst und die Filesystem API, die das Arbeiten mit Dateien und Dateisystemen erheblich vereinfacht.

Zusammenfassend lässt sich daher sagen, dass der Emscripten Compiler durch seinen Funktionsumfang und seine Dokumentation sowie sein gutes Nutzungsverhalten ein hilfreiches Tool bei der Programmierung mit WebAssembly unter C/C++ ist, welches Komplexität nehmen und Produktivität erhöhen kann.

## 6.3 Bewertung der Programmierung mit WebAssembly

Die Bewertung der Programmierung unter WebAssembly misst sich an den Kriterien, wie viel Vorwissen für einen erfolgreichen Einstieg nötig ist und wie komplex bzw. komfortabel die Entwicklung ist, was durch den Umfang an Helfer-Funktionen und Ausführlichkeit der Dokumentation beeinflusst wird.

Anders als in vielen Programmiersprachen o. ä. setzt WebAssembly mehr Wissen zu verschiedenen Bereichen voraus. Dies ist als Erstes dem Umstand geschuldet, dass man in WebAssembly üblicherweise mit mindestens zwei verschiedenen Sprachen arbeitet. Dabei handelt es sich bei einer der Sprachen immer um Javascript, da WebAssembly Javascript nicht ersetzt, sondern erweitert[76]. Im Bezug auf Javascript sind gute bis fortgeschrittene Kenntnisse zu empfehlen, um bspw. das Speicherprinzip, welches durch Emscripten als typisierte Arrays angeboten wird[18], nachvollziehen zu können. Abhängig davon, wie tief man

sich mit der Thematik auseinandersetzen will, werden wiederum mehr Vorkenntnisse nötig. Theoretisch ist es jedoch auch möglich, sich weitestgehend auf die Webentwicklung, bzw. Javascript, zu beschränken, indem bereits implementierte Funktionalitäten aus Bibliotheken anderer Sprachen genutzt werden. Doch auch in diesem Fall muss sich der Programmierer in Javascript mit manuellem Speichermanagement auseinandersetzen, um bspw. Parameter für Funktionen vorzubereiten.

Die Komplexität der Programmierung mit WebAssembly wird zum einen dadurch beeinflusst, dass während der Entwicklung zwischen grundsätzlich verschiedenen Sprachen gewechselt werden kann. Ein Beispiel hierfür wäre C und Javascript. Während C eine typisierte Sprache ist, die über ein manuelles Speichermanagement und mehrere Datentypen verfügt, ist Javascript nicht typisiert, stellt nur wenige Datentypen bereit und verfügt über einen Garbage Collector.

Ein anderer Punkt ist, dass die Programmierung mit Javascript anspruchsvoller wird, da Javascript mit bisher nicht genutzten Funktionalitäten arbeitet und bisher nicht vorhandene Probleme gelöst werden müssen. Während sich in der nativen Programmierung meist nicht viel ändert (ausgenommen den Fall Javascript Funktionen aus C aufzurufen), müssen in Javascript bspw. Pointer erzeugt werden, um Parameter in den virtuellen Speicher zu schreiben und sie so für den nativen Code verfügbar zu machen. Hilfreich bei derlei Aufgaben sind Funktionen, die durch den Glue-Code von Emscripten zur Verfügung gestellt werden.

Bei genauerer Betrachtung lässt sich feststellen, dass WebAssembly zu Beginn viel Wissen fordert und auch während der Entwicklung eine anspruchsvolle Technologie ist, welche verschiedene Programmierkenntnisse voraussetzt. Dieser Zustand könnte sich jedoch in Zukunft zum Bessern verändern, da WebAssembly zu diesem Zeitpunkt noch in der Version 1.0 vorliegt und somit das Minimum Viable Product darstellt[79]. Entwickler können also künftig mehr Funktionalitäten von WebAssembly erwarten. Das Problem bleibt jedoch, dass sich durch die erforderlichen Programmierkenntnisse nur schwer eine klare Zielgruppe an Programmierern identifizieren lassen. WebAssembly ist folglich besonders für diejenigen interessant, die sich sowohl mit nativer Programmierung als auch mit der Programmierung im Web beschäftigen.

# 7 Zusammenfassung

WebAssembly stellt eine neue Technologie dar[75], um die Vorteile nativer Anwendungen (hinsichtlich Leistungsumfang und Performance[47]) mit den Vorteilen des Webs (keine Installationen und Systemunabhängigkeit) zu vereinen. WebAssembly selbst ist dabei ein neues binäres Format, das in Browsern ausgeführt werden kann[75]. Dadurch ist es möglich, Sprachen wie C/C++ und Rust in binäre WebAssembly Module zu kompilieren und im Browser zu nutzen. Geeignet ist WebAssembly, um bestehende Webanwendungen hinsichtlich der Performance zu verbessern und um Flaschenhälse zu optimieren[47]. Daneben ermöglicht der Einsatz von WebAssembly auch neue Arten von Anwendungen, die aufgrund fehlender Bibliotheken oder Leistungsfähigkeit bisher nicht in Javascript allein umsetzbar waren. Beispielhafte Anwendungsfälle sind die Bearbeitung bzw. das Arbeiten mit Video-, Audio- oder Bilddateien[80].

Die Nutzung von WebAssembly beschränkt sich jedoch nicht nur auf den Browser[75]. Da WebAssembly aus Sicherheitsgründen auf einer virtuellen stackbasierten Maschine ausgeführt wird, kann es u. a. auch in NodeJS oder lokal auf einem Computer ausgeführt werden.

Für die Kompilierung sind spezielle Compiler wie bspw. Emscripten[9] notwendig. Emscripten ist eine Compiler-Toolchain, die es ermöglicht, C und C++ Code in das WebAssembly Format zu kompilieren[10]. Zusätzlich bietet dieser Compiler die Möglichkeit Javascript-Code zu erzeugen, der die Aufgabe übernimmt, die binären WebAssembly Module zu laden, zur Verfügung zu stellen und weitere Funktionen zum effektiven Arbeiten im WebAssembly anzubieten[33].

Die Programmierung von Webanwendungen mit WebAssembly folgt meist dem Prinzip, dass bestimmte Funktionalitäten in nativen Sprachen implementiert und als WebAssembly Module kompiliert werden(Beispiele hierfür sind AutoCAD[2], deren Codebasis um WebAssembly erweitert wurde und die Webanwendung Squoosh[70], welche ebenfalls bereits existierende Bibliotheken anderer Sprachen nutzt). Diese Funktionen können anschließend durch den Einsatz von Javascript geladen und genutzt werden. Um Funktionen jedoch nutzen zu können, muss oftmals mit dem Problem unterschiedlicher bzw. nicht kompatibler Datentypen umgegangen werden. In dieser ersten Version von WebAssembly ist es bspw. nicht ohne Weiteres möglich, einen String aus Javascript einer Funktion in bspw. C zu übergeben, welche als Parameter einen Array aus Charactern erwartet. Die Lösung für dieses und ähnliche Probleme erfordert in Javascript die direkte Manipulation des virtuellen WebAssembly Speichers (wie in Kap. 4.3 und 5 gezeigt wurde).

# 8 Ausblick

WebAssembly ist seit 2017 auf dem Markt und momentan noch in der Version 1.0, auch MVP-Version genannt, verfügbar[79]. Demnach existiert diese Technologie seit 4 Jahren. Wenn man sich nun vor Augen führt, dass bspw. Javascript 1995 erstmals erschienen ist[73] und damit 2020 sein 25-jähriges Jubiläum feiern durfte, wird klar, dass WebAssembly noch am Anfang der Entwicklung steht.

Ein Vorteil besteht jedoch darin, dass WebAssembly nicht von Null anfängt, da es sich dabei nicht um eine eigene Programmiersprache handelt, sondern sich die bisherigen Entwicklungen mehrerer gut entwickelter Technologien zu Nutze machen kann (C, C++, Rust und Javascript)[78].

Aufgrund des jungen Alters von WebAssembly gibt es noch viele Funktionalitäten, die sich Entwickler für effektiveres Arbeiten wünschen. Aus diesen Grund gibt es auf dem GitHub Repository der WebAssembly Entwicklung und der damit verbundenen Infrastruktur[42] die Möglichkeit, Vorschläge für zu unterstützende Funktionalitäten einzureichen, um so die Entwicklung dieser Technologie stetig voranzutreiben. Anträge dieser Art (Proposals genannt) werden in 5 Phasen behandelt[43]. Dabei muss ein Antrag initial erst auf Nützlichkeit und Umsetzbarkeit geprüft werden. Anschließend wird der Antrag immer weiter bearbeitet: Von der konkreten Spezifikation zur Implementierung und schließlich bis hin zur endgültigen Standardisierung[43].

Aktuell befindet sich bspw. der Antrag *Bulk memory operations*[37] in Phase 4 des Entwicklungsprozesses[42] und ist damit kurz vor der Fertigstellung. Dieser Antrag[37] befasst sich mit der Performance-Optimierung der Speicherfunktionen `memcpy` und `memmove`. Andere Anträge[42] aus Phase 1 befassen sich wiederum auch mit Typ-Importen, Garbage Collection und Interface Typen.

Rückblickend lässt sich sagen, dass die technologische Entwicklung der Technologie und Infrastruktur rund um WebAssembly stetig weiter entwickelt wird. Schwieriger ist es bisher jedoch zu sagen, wie die Akzeptanz und Nutzung von WebAssembly bei Firmen aussieht, da hierzu keinerlei Statistiken vorliegen.

# Literaturverzeichnis


[1] CONSTRUCT 3, 2021.
https://www.construct.net/en[Aufgerufen im Januar].

[2] Autodesk. Was ist AutoCAD?, 2021.
https://www.autodesk.de/products/autocad/overview?term=1-YEAR&support=null[Aufgerufen im Januar].

[3] Statistisches Bundesamt. Häufigkeit der Internetnutzung von Nutzern in %, 2021.
https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Einkommen-Konsum-Lebensbedingungen/_Grafik/_Interaktiv/it-nutzung-haeufigkeit.html[Aufgerufen im Januar].

[4] Statistisches Bundesamt. Internetnutzung von Personen nach Altersgruppen in %, 2021.
https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Einkommen-Konsum-Lebensbedingungen/_Grafik/_Interaktiv/it-nutzung-alter.html[Aufgerufen im Januar].

[5] Dev-Insider. Was ist ein AOT Compiler?, 2021.
https://www.dev-insider.de/was-ist-ein-aot-compiler-a-993285/[Aufgerufen im Januar].

[6] Dev-Insider. Was ist ein JIT Compiler?, 2021.
https://www.dev-insider.de/was-ist-ein-jit-compiler-a-984944/[Aufgerufen im Januar].

[7] Red Hat Developer. Javascript Engine & Performance Comparison (V8, Chakra, Chakra Core), 2021.
https://developers.redhat.com/blog/2016/05/31/javascript-engine-performance-comparison-v8-charkra-chakra-core-2/[Aufgerufen im Januar].

[8] Chrome Developers. PNaCI, 2021.
https://developer.chrome.com/docs/native-client/[Aufgerufen im Januar].

[9] Emscripten, 2021.
https://emscripten.org/index.html[Aufgerufen im Januar].

[10] Emscripten. About emscripten, 2021.
https://emscripten.org/docs/introducing_emscripten/about_emscripten.html?highlight=unreal%20engine[Aufgerufen im Januar].

[11] Emscripten. Accessing memory from js, 2021. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-access-memory[Aufgerufen im Januar].

[12] Emscripten. Call compiled C/C++ code "directly" from javascript, 2021. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#call-compiled-c-c-code-directly-from-javascript[Aufgerufen im Januar].

[13] Emscripten. Calling javascript from C/C++, 2021. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#calling-javascript-from-c-c[Aufgerufen im Januar].

[14] Emscripten. Compiling webassembly - compiler output, 2021. https://emscripten.org/docs/compiling/WebAssembly.html#compiler-output[Aufgerufen im Januar].

[15] Emscripten. Debugging, 2021. https://emscripten.org/docs/porting/Debugging.html[Aufgerufen im Januar].

[16] Emscripten. Download and install, 2021. https://emscripten.org/docs/getting_started/downloads.html[Aufgerufen im Januar].

[17] Emscripten. Emscripten compiler frontend (emcc), 2021. https://emscripten.org/docs/tools_reference/emcc.html[Aufgerufen im Januar].

[18] Emscripten. Emscripten memory representation, 2021. https://emscripten.org/docs/porting/emscripten-runtime-environment.html#emscripten-memory-representation[Aufgerufen im Januar].

[19] Emscripten. Emscripten runtime environment, 2021. https://emscripten.org/docs/porting/emscripten-runtime-environment.html#emscripten-runtime-environment[Aufgerufen im Januar].

[20] Emscripten. Emscripten tutorial, 2021. https://emscripten.org/docs/getting_started/Tutorial.html[Aufgerufen im Januar].

[21] Emscripten. emscripten.h, 2021. https://emscripten.org/docs/api_reference/emscripten.h.html#[Aufgerufen im Januar].

[22] Emscripten. File system api, 2021. https://emscripten.org/docs/api_reference/Filesystem-API.html[Aufgerufen im Januar].

[23] Emscripten. Interacting with code, 2021. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html[Aufgerufen im Januar].

[24] Emscripten. Module object, 2021. https://emscripten.org/docs/api_reference/module.html?highlight=module[Aufgerufen im Januar].

[25] Emscripten. Module object, affecting execution, 2021. https://emscripten.org/docs/api_reference/module.html?highlight =onRuntimeInitialized#affecting-execution [Aufgerufen im Januar].

[26] Emscripten. Optimizing code - other optimization issues, 2021. https://emscripten.org/docs/optimizing/Optimizing-Code.html?#other-optimization-issues[Aufgerufen im Januar].

[27] Emscripten. preamble.js, 2021. https://emscripten.org/docs/api_reference/preamble.js.html?[Aufgerufen im Januar].

[28] Emscripten. Preamble.js - ccall(), 2021. https://emscripten.org/docs/api_reference/preamble.js.html#ccall[Aufgerufen im Januar].

[29] Emscripten. preamble.js - conversion functions, 2021. https://emscripten.org/docs/api_reference/preamble.js.html#conversion-functions-strings-pointers-and-arrays[Aufgerufen im Januar].

[30] Emscripten. Preamble.js - cwrap(), 2021. https://emscripten.org/docs/api_reference/preamble.js.html#cwrap[Aufgerufen im Januar].

[31] GitHub Emscripten, 2021. https://github.com/emscripten-core/emscripten[Aufgerufen im Januar].

[32] GitHub Emscripten. Emscripten compiler frontend (emcc), 2021. https://github.com/emscripten-core/emscripten/blob/master/docs/emcc.txt[Aufgerufen im Januar].

[33] GitHub Emscripten. Emscripten compiler frontend (emcc), 2021. https://github.com/emscripten-core/emscripten/blob/master/src/settings.js[Aufgerufen im Januar].

[34] Carsten Eilers entwickler.de. WebAssembly – Ist das sicher oder kann das weg?, 2021. https://entwickler.de/online/security/wie-sicher-ist-webassembly-579892572.html[Aufgerufen im Januar].

[35] GitHub. AssemblyScript, 2021. https://github.com/AssemblyScript/assemblyscript[Aufgerufen im Januar].

[36] GitHub. Binaryan, 2021. https://github.com/WebAssembly/binaryen[Aufgerufen im Januar].

[37] GitHub. Bulk memory operations and conditional segment initialization, 2021. https://github.com/WebAssembly/bulk-memory-operations/blob/master/proposals/bulk-memory-operations/Overview.md[Aufgerufen im Januar].

[38] GitHub. Chakracore, 2021. https://github.com/chakra-core/ChakraCore[Aufgerufen im Januar].

[39] GitHub. Rustwasm/wasm-pack, 2021. https://github.com/rustwasm/wasm-pack[Aufgerufen im Januar].

[40] GitHub. Squoosh, 2021. https://github.com/GoogleChromeLabs/squoosh[Aufgerufen im Februar].

[41] GitHub. wasmtime, 2021. https://github.com/bytecodealliance/wasmtime[Aufgerufen im Januar].

[42] GitHub. WebAssembly Proposals, 2021. https://github.com/WebAssembly/proposals[Aufgerufen im Januar].

[43] GitHub. WebAssembly W3C Process, 2021. https://github.com/WebAssembly/meetings/blob/master/process/phases.md[Aufgerufen im Januar].

[44] What is V8?, 2021. https://v8.dev[Aufgerufen im Januar].

[45] Sven Kölpin. WebAssembly: Die nächsten Schritte, 2021. https://jaxenter.de/java/enterprisetales-webassembly-wasm-91563[Aufgerufen im Januar].

[46] The Rust Programming Language. Hello, Cargo!, 2021. https://doc.rust-lang.org/book/ch01-03-hello-cargo.html[Aufgerufen im Januar].

[47] moz://a HACKS. Oxidizing Source Maps with Rust and WebAssembly, 2021. https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/[Aufgerufen im Januar].

[48] Mozilla. All about Mozilla's JavaScript engine, 2021. https://blog.mozilla.org/javascript/[Aufgerufen im Januar].

[49] Mozilla. Mozilla and Epic Preview Unreal Engine 4 Running in Firefox, 2021. https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/[Aufgerufen im Januar].

[50] Mozilla Developer Network. Converting WebAssembly text format to wasm, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm[Aufgerufen im Januar].

[51] Mozilla Developer Network. Webassembly, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly[Aufgerufen im Januar].

[52] Mozilla Developer Network. Webassembly.memory() constructor, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory/Memory[Aufgerufen im Januar].

[53] Mozilla Developer Networks. ArrayBuffer, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer[Aufgerufen im Januar].

[54] Mozilla Developer Networks. Compiling from rust to webassembly, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm[Aufgerufen im Januar].

[55] Mozilla Developer Networks. DataView, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView[Aufgerufen im Januar].

[56] Mozilla Developer Networks. DataView.prototype.setUint32(), 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView/setUint32[Aufgerufen im Januar].

[57] Mozilla Developer Networks. Endianness, 2021. https://developer.mozilla.org/en-US/docs/Glossary/Endianness[Aufgerufen im Januar].

[58] Mozilla Developer Networks. FileReader(), 2021. https://developer.mozilla.org/en-US/docs/Web/API/FileReader/FileReader[Aufgerufen im Januar].

[59] Mozilla Developer Networks. IndexedDB, 2021. https://developer.mozilla.org/de/docs/Web/API/IndexedDB_API[Aufgerufen im Januar].

[60] Mozilla Developer Networks. TypedArray, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray[Aufgerufen im Januar].

[61] Mozilla Developer Networks. TypedArray Constructor, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray#constructor[Aufgerufen im Januar].

[62] Mozilla Developer Networks. TypedArray objects, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray#typedarray_objects[Aufgerufen im Januar].

[63] Mozilla Developer Networks. URL.createObjectURL(), 2021. https://developer.mozilla.org/de/docs/Web/API/URL/createObjectURL[Aufgerufen im Januar].

[64] Mozilla Developer Networks. WebAssembly, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly[Aufgerufen im Januar].

[65] Mozilla Developer Networks. Webassembly Concepts, 2021. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts[Aufgerufen im Januar].

[66] OpenCV. Face Recognition with OpenCV, 2021. https://docs.opencv.org/3.4/da/d60/tutorial_face_main.html[Aufgerufen im Januar].

[67] OpvenCV. Introduction, 2021. https://docs.opencv.org/master/d1/dfb/intro.html[Aufgerufen im Januar].

[68] Marco Selvatici. Webassembly Tutorial, 2021. https://marcoselvatici.github.io/WASM_tutorial/#your_first_WASM_WebApp[Aufgerufen im Januar].

[69] WebAssembly Core Specification, 2021. https://webassembly.github.io/spec/core/bikeshed/index.html[Aufgerufen im Januar].

[70] Squoosh, 2021. https://squoosh.app/[Aufgerufen im Februar].

[71] StackOverflow Survey. Most Popular Technologies - Programming, Skripting and Markup Languages, 2021. https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-professional-developers[Aufgerufen im Februar].

[72] t3n. JavaScript-Engines: Die Turbolader für den Browser, 2021. https://t3n.de/magazin/javascript-engines-turbolader-fuer-browser-247188/[Aufgerufen im Januar].

[73] t3n. JavaScript, Flash, CSS und mehr: Die Geschichte des Webdesigns in 9 GIFs, 2021. https://t3n.de/news/javascript-flat-geschichte-webdesign-585903/[Aufgerufen im Januar].

[74] Typescript, 2021. https://www.typescriptlang.org/[Aufgerufen im Januar].

[75] WebAssembly, 2021. https://https://webassembly.org/[Aufgerufen im Januar].

[76] WebAssembly. FAQ - Is WebAssembly trying to replace JavaScript?, 2021. https://webassembly.org/docs/faq/[Aufgerufen im Januar].

[77] WebAssembly. FAQ - Security, 2021.
https://webassembly.org/docs/security/[Aufgerufen im Januar].

[78] WebAssembly. I want to..., 2021.
https://webassembly.org/getting-started/developers-guide/[Aufgerufen im Januar].

[79] WebAssembly. Roadmap, 2021.
https://webassembly.org/roadmap/[Aufgerufen im Januar].

[80] WebAssembly. Use Cases, 2021.
https://webassembly.org/docs/use-cases/[Aufgerufen im Januar].

[81] Jürgen Wolf. *C von A bis Z - Das umfassende Handbuch.* Galileo Press GmbH, Bonn, 1. Auflage 2003.

[82] YouTube. WebAssembly for Web Developers (google I/O '19), 2021.
https://www.youtube.com/watch?v=njt-Qzw0mVY&t=1631s[Aufgerufen im Januar].