HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

ins
INSTITUTE FOR
NETWORKED SOLUTIONS

JavaScript to WebAssembly Cross Compiler
Studienarbeit

Abteilung Informatik

Hochschule für Technik Rapperswil

Herbstsemester 2018

Autoren:	Matteo Kamm, Mike Marti
Betreuer:	Prof. Dr. Luc Bläser
Projektpartner:	Institute for Networked Solutions

Inhaltsverzeichnis

1 Abstract

Moderne Webbrowser unterstützen die Ausführung des binären WebAssembly Formats, das eine hohe Performance sowie geringere Fluktuationen der Programmlaufzeiten zwischen den Browsern ermöglicht.

Verschiedene Compiler erlauben bereits heute das Übersetzen der Programmiersprachen C, C++ sowie Rust zu WebAssembly. Da diese Sprachen nicht im Browser ausgeführt werden können, sind solche Compiler nur ahead-of-time nutzbar. Für eine just-in-time Kompilation im Browser müsste der Compiler in JavaScript implementiert werden. JavaScript Code könnte dann durch eine Übersetzung direkt beim Client im Browser beschleunigt werden.

Damit die Ausführung ohne weitere Tools im Browser lauffähig ist, wurde der Transpiler in TypeScript geschrieben. Der Transpiler kann über eine dokumentierte Schnittstelle angesprochen werden. Übersetzt wird ein stark typisiertes und für rechenintensive Aufgaben ausgelegtes Subset von JavaScript. Anschliessend wird das WebAssembly über ein Call-Wrapper aufgerufen. Für das Erstellen des abstrakten Syntaxbaums wurde BabelJS verwendet. Das Generieren des binären WebAssemblies wird von Binaryen übernommen.

Das Resultat ist ein schlanker und performanter Transpiler, der JavaScript zu WebAssembly übersetzt. Der Endnutzer benötigt für die Ausführung des Transpilers lediglich einen Browser und keine weiteren Programme. Die Transpilierung zu WebAssembly konnte trotz zusätzlicher Transpilationszeit in einigen Fällen eine schnellere Ausführung erzielen. Die Arbeit hat gezeigt, dass die WebAssembly Infrastruktur noch Verbesserungspotential aufweist. Es könnte beispielsweise noch mehr optimiert und dadurch die Laufzeit verbessert werden.

2 Einführung

2.1 Ausgangslage

Die meisten Webapplikationen werden heutzutage zumindest teilweise in der Programmiersprache JavaScript entwickelt. Immer mehr Programmlogik wandert vom Server zum Client. JavaScript ist eine schwach typisierte, dynamische und zur Laufzeit interpretierte Programmiersprache. TypeScript [8], eine typisierte Übermenge von JavaScript, wird häufig als Alternative zu JavaScript eingesetzt. Sie bietet zusätzliche Sicherheit dank der starken Typisierung des Programmcodes. Da TypeScript nicht direkt im Browser ausgeführt werden kann, muss der gesamte TypeScript Code vor der Ausführung in JavaScript Code übersetzt werden.

Moderne Webbrowser unterstützen die Ausführung des binären WebAssembly Formats [11, Binary Format, S. 85], das eine quasi maschinennahe Performance sowie geringere Fluktuationen der Programmlaufzeiten zwischen den Browsern ermöglicht. Diese Eigenschaften von WebAssembly wurden in einer füheren Masterarbeit [10] gezeigt. Das Binärformat von WebAssembly ist browserunabhängig. Derselbe WebAssembly Code kann somit unter allen modernen Browsern ausgeführt werden. Ein weiterer Nutzen von WebAssembly ist die Obfuskation[1]. Kritische Teile einer Software, die im Browser der Benutzer ausgeführt werden müssen, können durch WebAssembly versteckt werden.

WebAssembly wird, gleich wie JavaScript, von einem Interpreter zur Laufzeit interpretiert. Anders als herkömmliche Assemblersprachen verwendet WebAssembly keine Register, sondern eine sogenannte strukturierte Stackmachine [6]. Somit werden sämtliche Zwischenresultate auf dem Stack und nicht in Registern gespeichert.

Verschiedene Compiler erlauben bereits heute das Übersetzen der Programmiersprachen C, C++ [16] sowie Rust [13] zu WebAssembly. Die oben erwähnte Masterarbeit [10] übersetzt Typescript zu WebAssembly und basiert auf der LLVM [4]. Da die LLVM Komponenten in C++ geschrieben sind und Browser lediglich JavaScript interpretieren können, ist eine Verwendung der vorhergehenden Arbeit im Browser nicht möglich. Ausserdem wäre zur Laufzeit der TypeScript Code nicht mehr vorhanden, da dieser vor der Ausführung zu JavaScript übersetzt wird. Damit die Übersetzung zur Laufzeit und im Browser stattfinden kann, ist es nötig, dass der Transpiler in JavaScript bzw. TypeScript geschrieben wird.

Eine Übersetzung zur Laufzeit im Browser bringt einige Vorteile mit sich. Es ist möglich dynamisch zu entscheiden, welcher Code zu WebAssembly transpiliert werden soll und welcher Code besser als JavaScript ausgeführt wird. Zur Laufzeit sind mehr Informationen vorhanden, die eine solche Entscheidung ermöglichen. Ausserdem kann der WebAssembly Code zur Laufzeit besser optimiert werden, da mehr Systeminformationen vorhanden sind. Dank des Transpilers[2], der komplett im Browser des Benutzers läuft, muss der Entwickler keine ahead-of-time Kompilation[3] durchführen. Diese Art der Kompilation war bis anhin nur mit externen und mühsam installierbaren Tools möglich. Die Übersetzung im Browser würde für den Benutzer keine externen Tools und für den Entwickler keine spezielle Toolchain mehr voraussetzen.

[1] Technik, die den Programmcode verändert, sodass er für Menschen nur schwer verständlich ist.

[2] Eine spezielle Art von Compiler, der Code einer Programmiersprache in äquivalenten Code einer anderen Programmiersprache übersetzt.

[3] Kompilieren des Programmcodes bevor er ausgeführt wird.

Das Ziel der Studienarbeit ist es einen schlanken, portablen und performanten Cross Compiler[4] Prototypen zu schreiben. Mit Schlankheit ist hier auch die Möglichkeit der Ausführung im Browser gemeint. Der Prototyp soll ein geeignetes Subset von JavaScript zur Laufzeit in WebAssembly umwandeln können. Rechenintensiver JavaScript Code soll durch die Übersetzung zur Laufzeit verschnellert werden. Der Prototyp ist somit ein JIT-Compiler[5].

2.2 Übersicht

Die Abbildung 1 zeigt die unterschiedlichen Komponenten, die zur Laufzeit an der Übersetzung beteiligt sind.

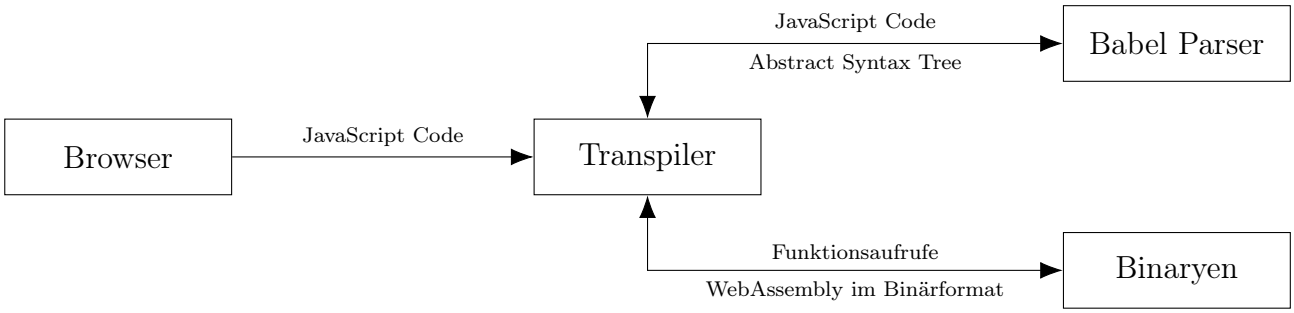


Abbildung 1: Übersicht Kompilationszeit

Der Transpiler ist die von uns während der Arbeit entwickelte Komponente. Während der Übersetzung ruft der Transpiler zuerst den Babel Parser [2] auf. Dieser erkennt die syntaktischen Elemente der Programmiersprache JavaScript und retourniert einen AST[6]. Die Compiler Infrastruktur Library Binaryen [5] generiert den eigentlichen WebAssembly Code im Binärformat. Dazu stellt sie eine API zur Verfügung, die dem Builder Pattern folgt.

Die Abbildung 2 zeigt die unterschiedlichen Komponenten, die zur Laufzeit am Aufruf des WebAssemblies beteiligt sind.

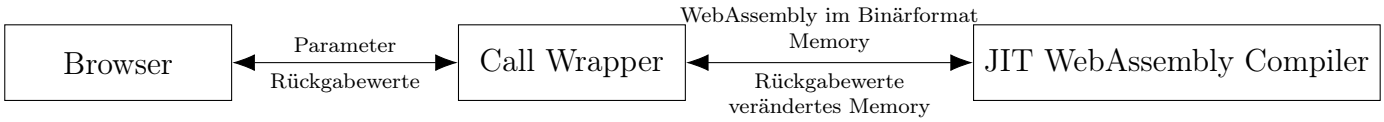


Abbildung 2: Übersicht Laufzeit

Ausgeführt wird das WebAssembly indem es von einem JIT-Compiler zu Maschinencode übersetzt wird. Dieser Compiler ist Teil des Browsers. Da WebAssembly in einer eigenen Sandbox ausgeführt wird, kann nicht direkt auf den Speicher des JavaScript Laufzeitsystems zugegriffen werden [11, Design Goals, S. 5]. Vom Call Wrapper erhält der Browser einen Snapshot des momentanen Speichers und retourniert nach der Ausführung den bearbeiteten Speicher sowie den Rückgabewert des WebAssembly Aufrufs.

[4]Synonym von Transpiler.
[5]Eine Art Compiler, der den Programmcode zur Laufzeit übersetzt.
[6]Abstract syntax tree, eine aggregierte Baumstruktur, die die Syntaxelemente als Knoten beinhaltet.

2.2.1 Beispiel

In diesem Abschnitt wird gezeigt, wie der JavaScript Code der Funktion *isPrime* in äquivalentem WebAssembly Code aussieht. Die Funktion entscheidet, ob eine Ganzzahl eine Primzahl ist.

Der JavaScript Code sieht wie folgt aus:

```
function isPrime(number) {
    for (var factor = 2; factor * factor <= number; factor++) {
        if (number % factor == 0) { return false; }
    }

    return true;
}
```

Beispiel 1: JavaScript isPrime Funktion

Zur Darstellung des WebAssemblies wurde hier das menschenlesbare Textformat [11, Text Format, S. 101] gewählt. Dieses nutzt sogenannte S-Expressions, eine Notation zur Darstellung von verschachtelten Listen.

Das WebAssembly der *isPrime* Funktion sieht im Textformat wie folgt aus:

```
(type $isPrime (func (param i32) (result i32)))
(func $isPrime (type $isPrime) (param $0 i32) (result i32)
 (local $1 i32)
 (set_local $1 (i32.const 2))
 (loop $label_1
  (if
   (i32.le_s
    (i32.mul (get_local $1) (get_local $1))
    (get_local $0)
   )
   (block
    (if
     (i32.eqz
      (i32.rem_s (get_local $0) (get_local $1))
     )
     (return (i32.const 0))
    )
    (set_local $1
     (i32.add (get_local $1) (i32.const 1))
    )
    (br $label_1)
   )
  )
 )
 (i32.const 1)
)
```

Beispiel 2: isPrime im WebAssembly Textformat

Dieses Format wird im Browser binär gespeichert und danach vom Laufzeitsystem interpretiert. Das Textformat wurde hier lediglich zur Veranschaulichung verwendet.

In diesem WebAssembly Beispiel wird zuerst ein neuer Typ für die Funktion *isPrime* definiert. Darauf folgt die Definition der Funktion. Sie setzt die lokale Variable mit dem Index $1 auf den Wert 2. Danach wird eine Loop begonnen. Sie beginnt mit einem If-Block. Die Bedingung vergleicht, ob die Variable $1 multipliziert mit sich selber kleiner als der Wert der Variablen

$2 ist. Trifft dies zu, so wird der enthaltene Block ausgeführt. Falls nicht, wird die Funktion verlassen. Bevor die Funktion verlassen wird, wird der Wert 1 geladen. In diesem Fall ist keine *return* Instruktion nötig, da sie implizit vorhanden ist. Im Block geschieht die Entscheidung, ob die Variable $0 modulo $1 gerechnet gleich 0 ist. Ausserdem wird die Variable $1 um 1 erhöht. Danach wird wieder an den Anfang der Loop gesprungen.

Der Transpiler erkennt hierbei, dass die Variable *factor* eine Ganzzahl ist und nutzt daher den Typen *i32*. Die Typen werden durch eine simple Typinferenz bestimmt. Hierbei wird darauf geachtet, dass ein möglichst effizienter Typ gewählt wird. Bei den Parametertypen und beim Rückgabetyp muss der Benutzer dem Transpiler die Typen explizit mitteilen. Wie die Inferenz genau funtkioniert, ist im Abschnitt 4.1 ausführlich beschrieben.

Wird der Transpiler zur Übersetzung genutzt, so sieht der Aufruf der isPrime Funktion aus Listing 1 wie folgt aus:

```
const wrapper = new Transpiler()
    .setSignature('isPrime', WebAssemblyType.BOOLEAN, WebAssemblyType.INT_32
       )
    .transpile(isPrime.toString());

wrapper.setFunctionName('isPrime').call(3301);
```

Beispiel 3: Übersetzen der isPrime Funktion

Das Beispiel überprüft, ob es sich bei der Zahl 3301 um eine Primzahl handelt. Der Entwickler gibt dazu die Signaturen der zu übersetzenden Funktionen an. Der Rückgabewert ist in diesem Fall Boolean und der erste Parameter ist ein Integer. Danach übergibt er den JavaScript Code. *toString* kann hier genutzt werden um von einem Funktionsobjekt den Sourcecode zu erhalten. Mit *setFunctionName* wird der Einstiegspunkt gesetzt. Das bedeutet, diese Funktion wird als erstes aufgerufen.

2.3 Struktur des Berichts

Der Bericht ist in drei Hauptteile unterteilt. Im Kapitel 3 wird das unterstützte Subset von JavaScript genauer beschrieben. Dieses Kapitel entspricht unseren funktionalen Anforderungen. Wie die Übersetzung zu WebAssembly stattfindet, ist in den Kapiteln 4 und 5 beschrieben. Zuerst wird auf die Cross Compilation sowie das Laufzeitverhalten eingeganen. Das Kapitel 6 wertet die Resultate der Performancemessungen aus. Die durch die Arbeit gewonnenen Erkentnisse sind im Kapitel 7 aufgeführt.

3 Language Set

Der Cross Compiler soll ein Subset von JavaScript in WebAssembly umwandeln können. Dieser Abschnitt beschreibt das Subset genauer. Der Compiler nimmt an, dass der Benutzer gültigen JavaScript Code übergibt. Auf die Verifikation des Codes wird verzichtet.

3.1 Grundsatz des Subsets

Das Subset fokussiert sich auf rechenintensiven, prozeduralen Programmcode. Sortieralgorithmen oder der Primzahlentest fallen beispielsweise in diese Kategorie. Sprachkonstrukte, die eine Speicherallozііerung zur Laufzeit voraussetzen, werden aus diesem Grund nicht unterstützt. Die meisten Programme können auf Speicherzugriffe nicht verzichten, deshalb werden Arrays bei der Parameterübergabe angeboten. Deren Grösse ist beim Aufruf bestimmbar und es ist keine dynamische Vergrösserung des Speichers notwendig.

Zur Laufzeit werden keine Null-Checks und keine Bounds-Checks[7] bei Arrayzugriffen durchgeführt. Da von einem validen JavaScript Code ausgegangen wird, sind diese nicht notwendig. Dadurch kann an Performance gewonnen werden. Sollte trozdem fehlerhafter Code übergeben werden, so sind die Konsequenzen nicht dramatisch. WebAssembly wird in einer Sandbox ausgeführt [11, Design Goals, S. 1], deshalb ist es nicht möglich, den Speicher des Browsers zu beeinflussen.

Programmcode, der DOM[8] Elemente nutzt, kann nicht transpiliert werden, da WebAssembly aktuell keine direkten Zugriffe auf Elemente des DOMs unterstützt. Durch häufige Kontextwechsel zwischen WebAssembly und JavaScript würde die Performance stark beeinträchtigt werden. Desweiteren unterstüzt WebAssembly das Importieren von JavaScript Objekten nicht. Ein Proposal der Community Group möchte dies mittels Host Bindings ändern [15].

Die Programmiersprache JavaScript ist dynamisch typisiert. Das bedeutet, sie prüft die Typen erst zur Laufzeit des Programms und erlaubt das beliebige Ändern von Variablentypen. WebAssembly Instruktionen sind hingegen strikt typisiert und es ist nicht ohne Weiteres möglich den Typen einer lokalen Variablen zu verändern. Unsere Sprache führt eine strikte Typisierung ein, um diese Diskrepanz aufzulösen. JavaScript kennt nur den numerischen Typen *Number*, der einem Double entspricht. Neben disem Typen verfügt das Subset über einen 32-bit Ganzzahl Typen.

Obwohl JavaScript eine objektorientierte Programmiersprache ist, wurde die Objektorientierung nicht in das Subset aufgenommen. Die Umsetzung mittels WebAssembly würde dem Grundsatz des Subsets widersprechen. Pre- und Post Increment bzw. Decrements sowie Shorthand Assignments wurden des Komforts wegen in das Subset aufgenommen. Sie haben keinen negativen Einfluss auf die Performance und verbessern zugleich die Lesbarkeit des Programmcodes.

[7] Überprüfung, ob der Index innerhalb der zugelassenen Grenzen liegt.
[8] Document Object Modell, API zur Ansteuerung von HTML Dokumenten

3.2 Typen

Unsere Subsprache kennt neben Double auch den Basistypen Integer. Falls möglich, wird dieser Basistyp genutzt, da Berechnungen mit Ganzzahlen um einiges schneller ausgeführt werden können. Ob ein Wert als Ganzzahl dargestellt werden kann, wird zur Kompilationszeit durch die Typinferenz entschieden. Siehe dazu Abschnitt 4.1.

Es ist möglich Arrays von numerischen Typen zu nutzen. Arrays von boolean Werten sind nicht zugelassen. Sie finden selten Verwendung in rechenintensiven Algorithmen und gehören somit laut dem Grundsatz des Subsets nicht zur Sprache.

Somit werden folgende Typen unterstützt:

- 32-bit Integer
- Array von 32-bit Integer
- 64-bit Float
- Array von 64-bit Floats
- Boolean

3.3 Unterstützte Sprachkonstrukte

Unterstützt werden die üblichen intra-prozeduralen Sprachelemente wie Verzweigungen, Loops, Zuweisungen etc. Zusätzlich dazu werden statische Funktionsaufrufe und rekursive Aufrufe angeboten. Jede Funktion muss einen Wert zurückgeben, welcher nicht zwingend verwendet werden muss.

Arrays können ausschliesslich als Parameter per Referenz übergeben werden. Die Werte der Arrays werden vor der Ausführung in einen isolierten Speicher kopiert. Somit sind Änderungen der Arrays nicht direkt im Speicher des JavaScript Laufzeitsystems sichtbar. Das Zurückkopieren muss explizit gewünscht und dem Wrapper mitgeteilt werden (opt-in). Eine asynchrone Ausführung des WebAssembly Codes via CallWrapper wird momentanen nicht unterstützt, wodurch Race Conditions ausgeschlossen sind. In einer späteren Version könnten ungünstige Verzahnungen zu unerwarteten Resultaten führen. Dann müsste sich der Entwickler bewusst sein, dass übergebene Arrays während der Ausführung des WebAssemblies nicht verändert werden dürfen.

Folgende Sprachkonstrukte werden unterstützt:

- Arrays als Parameter
- Assignment
- Binäre Operatoren: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `>`, `>=`, `<`, `<=`
- For-Loop
- Funktionsaufrufe
- Identifier
- If-else
- Literale: Boolean, Integer und Double
- Pre- und Post-Increment/Decrement: `++`, `--`
- Return
- Shorthand Assignment: `+=`, `-=`, `*=`, `/=`, `%=`
- Unäre Operatoren: `+`, `-`, `!`
- Variablen Deklaration
- While-Loop

Genauere Einschränkungen sowie Beispiele der einzelnen Sprachkonstrukte sind im Anhang A beschrieben. Eine API Dokumentation des Transpilers befindet sich im Anhang C.

4 Cross Compilation

Der Cross Compiler nimmt den validen JavaScript Code entgegen und übersetzt ihn zu WebAssembly im Binärformat. Zur Erkennung der syntaktischen Elemente des JavaScript Codes nutzt er den BabelJS Parser [2]. Mithilfe des von BabelJS erhaltenen ASTs kann dann der nötige WebAssembly Code generiert werden. In diesem Abschnitt werden die einzelnen Schritte, die der Cross Compiler durchläuft, genauer erläutert.

4.1 Typinferenz

Die simple, intra-prozedurale Typinferenz kümmert sich um die Herleitung der statischen Typen sämtlicher Expressions einer Funktion. Diese Inferenz ist aufgrund der starken Typisierung von WebAssembly erforderlich. Zusätzlich zum AST Node wird jeweils die Typinformation abgespeichert. Später können diese Informationen zur Generierung des WebAssemblies genutzt werden.

Der Benutzer muss dem Transpiler bei den Typen der Parameter sowie des Rückgabewertes von Funktionen helfen, da diese nicht automatisch hergeleitet werden. Dies ist sehr ähnlich zu TypeScript, da dort die Typen auch angegeben werden sollten. Im Gegensatz zu TypeScript ist bei uns die Angabe zwingend erforderlich. Anhand der AST Nodes und einem vorgegebenen Schema können die Typen inferiert werden. Dazu wird der AST mittels einer Postorder Traversierung [3] visitiert. Mögliche numerische Erweiterungen müssen dabei berücksichtigt werden. Siehe dazu Anhang A.2.

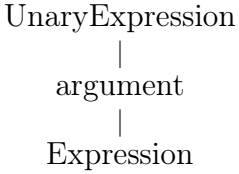
In Zukunft wäre eine Herleitung der meisten Typen basierend auf den Parameterwerten zur Laufzeit denkbar. Der Benutzer müsste dann keine Typen zum Transpilationszeitpunkt angeben. Der WebAssembly Code könnte erst beim effektiven Aufruf generiert werden, wodurch sich die Laufzeit verschlechtern würde.

4.2 Template-Based Code Generation

Diese Art der Codegenerierung nutzt eine Postorder Traversierung [3]. Wird ein Teilbaum während der Traversierung erkannt, so kann ein vordefiniertes Muster zur Übersetzung angewandt werden. In diesem Abschnitt werden vier solche Muster genauer erläutert. Die hier aufgeführten Muster wurden aufgrund ihrer hohen Relevanz gewählt. Sie stellen wichtige Bestandteile der Subsprache dar und können gleichzeitig relativ simpel erklärt werden.

4.2.1 Unäre Operatoren

Unäre Operatoren werden von BabelJS wie folgt in einen AST abgebildet:

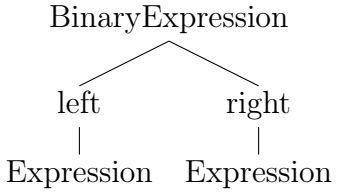


Bei der Traversierung wird zuerst das *Argument* der *UnaryExpression* visitiert. Anschliessend wird die erzeugte Binaryen Expression vom Expressionstack gepopt und mit dem unären Operator aus der *UnaryExpression* verknüpft. Die dadurch erhaltene Binaryen Expression wird wiederum auf den Expressionstack gelegt.

Logische Negation Die unäre Negation wird mit der WebAssembly Instruktion *i32.eqz* realisiert. Sie pusht die Zahl 1 auf den Stack, falls der Operand gleich 0 ist, ansonsten wird 0 gepusht.

4.2.2 Binäre Operationen

Binäre Operatoren werden von BabelJS wie folgt in einen AST abgebildet:



Bei der Traversierung werden zuerst nacheinander die beiden *Expressions* visitiert. Anschliessend werden die beiden zuvor erzeugten Binaryen Expressions vom Expressionstack gepopt und mit dem binären Operator aus der *BinaryExpression* verknüpft. Die dadurch erhaltene Binaryen Expression wird wiederum auf den Expressionstack gepusht.

Binäre Operationen der Form `left operator right` werden wie folgt in WebAssembly abgebildet:

```
(type.operator
 (left)
 (right)
)
```

Beispiel 4: WebAssembly Template für binäre Operationen

Boolsche Operationen Der Transpiler nutzt eine short-circuit Auswertungsstrategie[9] bei den Operatoren `&&` und `||`. Dies wird mittels der in WebAssembly vorhandenen *select* Instruktion erreicht. Die select Funktion von Binaryen nimmt eine Expression als Condition und zwei weitere Expressions entgegen. Je nachdem ob die Condition zutrifft, wird die erste oder die zweite Expression als nächstes ausgeführt.

[9] Auswertungsstrategie, bei der das zweite Argument nur dann ausgewertet wird, wenn das erste Argument das Endresultat noch nicht eindeutig bestimmt.

Im Textformat von WebAssembly hat die *select* Instruktion folgende Bedeutung: Selektiere den zweiten Operanden, falls der Dritte gleich 0 ist. Ansonsten selektiere den ersten Operanden.

Konkret wird folgender WebAssembly Code für `&&` Verknüpfungen generiert:

```
(select
 (right)
 (i32.const 0)
 (left)
)
```

Beispiel 5: WebAssembly Template für das logische Und

Und folgender WebAssembly Code für `||` Verknüpfungen:

```
(select
 (i32.const 1)
 (right)
 (left)
)
```

Beispiel 6: WebAssembly Template für das logische Oder

4.2.3 Expression Statements

Statements, die lediglich aus einer einzelnen Expression bestehen, werden *ExpressionStatements* genannt. Funktionsaufrufe, Increments, Decrements oder Assignments zählen zu dieser Art von Statements. Der durch solche *ExpressionStatements* generierte Wert muss in gewissen Fällen explizit vom Laufzeitstack verworfen werden. Dazu generiert der Transpiler eine *drop* Instruktion.

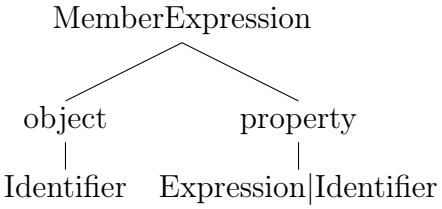
Das dazugehörige WebAssembly Template ist trivial:

```
(drop
 (expression)
)
```

Beispiel 7: WebAssembly Template der Expression Statements

4.2.4 Arrayzugriffe

Arrayzugriffe werden von BabelJS wie folgt in einen AST abgebildet:



Je nachdem, ob es sich um einen Zugriff auf einen Wert oder ein Property handelt, entspricht property einer *Expression* oder einem *Identifier*.

Lesezugriff Für das Laden eines Wertes wird ein Pointer, ein Offset und ein Alignment[10] benötigt. Die Adresse ist jeweils in der lokalen Variablen des Arrays abgespeichert. Der *offset* Parameter von Binaryen kann nicht verwendet werden, da der Wert erst zur Laufzeit bekannt ist. Stattdessen wird der Pointer mittels einer Addition und Multiplikation richtig berechnet:

$$\text{Address} = \text{Base} + \text{Index} \times \mathit{sizeof}(\text{Type})$$

Das dazugehörige WebAssembly Template bildet die obige Gleichung ab:

```
(type.load
 (i32.add
  (get_local base)
  (i32.mul
   (i32.const index)
   (i32.const sizeof(type))
  )
 )
)
```

Beispiel 8: WebAssembly Template der Lesezugriffe von Arrays

Schreibzugriff Der Schreibzugriff erfolgt analog zum Lesezugriff mit dem einzigen Unterschied, dass bei der *store* Instruktion der zu speichernde Wert mitgegeben werden muss. Der Offset kann aus dem gleichen Grund wie beim Lesezugriff 4.2.4 nicht genutzt werden. Die von Binaryen zur Verfügung gestellte store Funktion gibt eine *Expression* zurück, welche jedoch auf den Stack der Statements gelegt wird. Es muss mit einer TypeScript-Annotation gearbeitet werden, damit die Typinkompatibilität ignoriert wird.

Das WebAssembly Template ist dem obigen sehr ähnlich:

```
(type.store
 (i32.add
  (base)
  (i32.mul
   (i32.const index)
   (i32.const sizeof(type))
  )
 )
 (value)
)
```

Beispiel 9: WebAssembly Template der Schreibzugriffe von Arrays

Zugriff auf die Länge Die Länge eines Arrays befindet sich immer an der Stelle -1. Der offset Parameter wird nicht verwendet, da die Binaryen API negative Offsets verbietet. Der Pointer muss mittels einer Subtraktion richtig berechnet werden:

$$\text{Address} = \text{Base} - 4$$

Hier kann natürlich das Template aus dem Beispiel 8 verwendet werden, da es sich um einen lesenden Zugriff handelt.

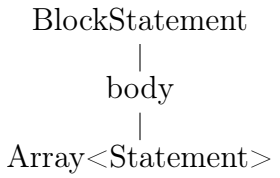
[10]Das Alignment gibt an, wie die Adresse ausgerichtet ist. In unserem Fall ist dieser Wert für *i32* Werte gleich 4 und für *f32* Werte gleich 8.

4.3 Control Flow

Unter Control Flow Elementen versteht man alle Sprachkonstrukte, welche einen Einfluss auf den Ablauf der Ausführung haben. Normalerweise werden einzelne Statements einer Funktion nacheinander abgearbeitet. Mit Control Flow Elementen kann zwischen Instruktionen gesprungen werden.

4.3.1 Block Statement

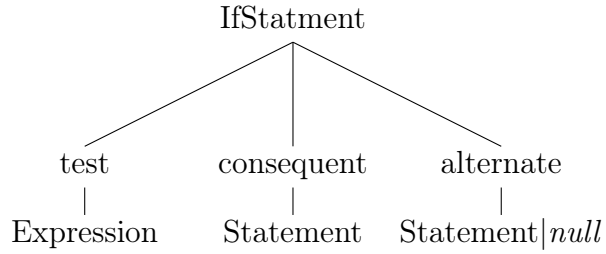
Block Statements werden von BabelJS wie folgt in einen AST abgebildet:



Bei der Visitierung von Control Flow Elementen wird mit WebAssembly Blöcken gearbeitet. Diese beinhalten mehrere Instruktionen und können ein Label am Ende angehängt bekommen [11, Control Instructions, S. 13]. Somit ist es möglich ans Ende eines Blocks zu springen. Blöcke können ineinander verschachtelt werden. Damit korrekter Code generiert werden kann, müssen bei der Visitierung von *BlockStatements* die *Statements* des umschliessenden Blocks zwischengespeichert werden. Ansonsten würde der Stand des Stacks durch die *Statements* innerhalb des Blocks verändert werden. Nach der Visitierung werden die zwischengespeicherten Statements und das neu erzeugte *BlockStatement* auf den Stack gelegt.

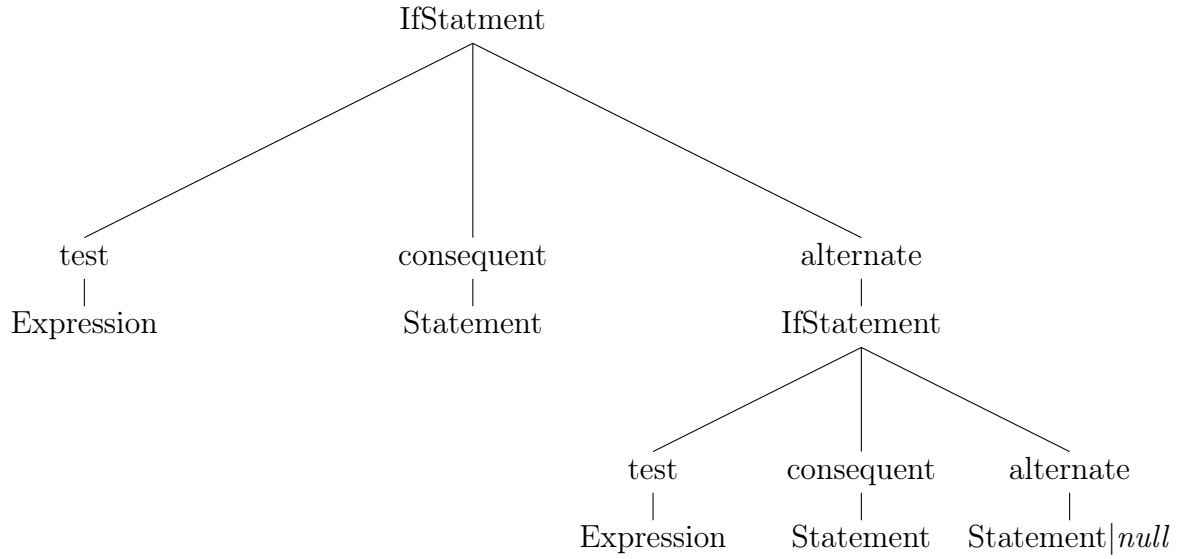
4.3.2 Branching

If-else Statements werden von BabelJS wie folgt in einen AST abgebildet:



Beim Traversieren von *IfStatements* wird zuerst die Bedingung und anschliessend die beiden *BlockStatements* abgearbeitet. Im Falle eines mehrzeiligen *IfStatements* entspricht das *Statement* einem *BlockStatement*. Sobald beide Blöcke vorhanden sind, kann mittels Binaryen ein if-Statement erzeugt werden, welches auf den Stack gepusht wird.

Bei else-if Statements sieht der AST leicht anders aus:



Wie im AST ersichtlich ist, wird von BabelJS bei einem else-if Statement kein umschliessendes *BlockStatement* erzeugt. Aus diesem Grund muss die Logik des Zwischenspeicherns des Statementstacks hier ebenfalls implementiert werden. Siehe dazu Abschnitt 4.3.1

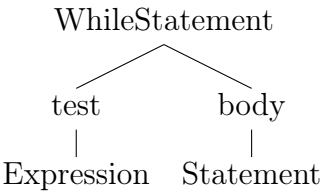
WebAssembly unterstützt bereits eine *if* Instruktion. Daher sind keine expliziten Sprunginstruktionen mehr nötig. Das Template sieht wie folgt aus:

```
(if
 (test)
 (consequent)
 (alternate)
)
```

Beispiel 10: WebAssembly Template für If-else Statements

4.3.3 While-Loop

While Statements werden von BabelJS wie folgt in einen AST abgebildet:



Beim Traversieren von *WhileStatements* wird zuerst die *Expression* und dann das *Statement* visitiert. Im Falle eines mehrzeiligen *WhileStatements* entspricht das *Statement* einem *Block-Statement*. Auch hier muss mit Labels gearbeitet werden. WebAssembly hat hierbei die Einschränkung, dass nur zu Labels von umschliessenden Blöcken gesprungen werden kann [11, Control Instructions, S. 13]. Aus diesem Grund müssen beide Sprunginstruktionen und der Loop-Body in demselben Block zusammengefasst werden. Im Gegensatz zu einem Block wird das Label einer Loop nicht am Ende, sondern am Anfang angehängt [11, Control Instructions, S. 13]. Die genaue Instruktionsfolge, die das Verhalten einer Loop implementiert, ist in der Abbildung 3 ersichtlich. Da beim Nichteintritt gesprungen werden muss und WebAssembly kein br_false kennt, muss die Bedingung des bedingten Sprungs negiert werden.

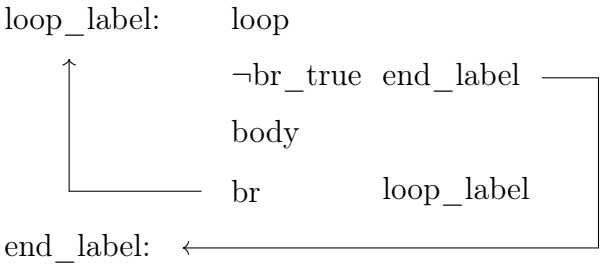


Abbildung 3: While-Loop Ablaufdiagramm

Basierend auf der Abbildung 3 wurde folgendes WebAssembly Template abgeleitet:

```
(loop $label_1
 (block $label_0
  (br_if $label_0
   (i32.eqz (test))
  )
  (block
   (body)
  )
  (br $label_1)
 )
)
```

Beispiel 11: WebAssembly Template für While-Loops

Folgende JavaScript Funktion beinhaltet eine simple While-Loop:

```
function factorial() {
  var i = 1, factorial = 0;

  while (i < 10) {
     factorial *= i;
     i++;
  }
}
```

Beispiel 12: Funktion mit While-Loop

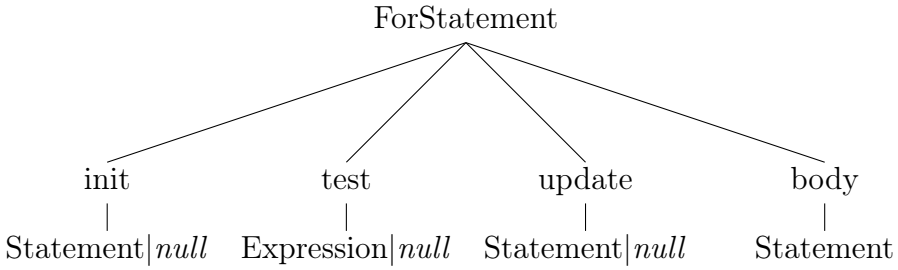
Der Transpiler generiert daraus folgende WebAssembly Instruktionen im Textformat:

```
(module
 (type $factorial (func (result i32)))
 (export "factorial" (func $factorial))
 (func $factorial (type $factorial) (result i32)
  (local $0 i32)
  (local $1 i32)
  (set_local $0 (i32.const 1))
  (set_local $1 (i32.const 0))
  (loop $label_1
   (block $label_0
    (br_if $label_0
     (i32.eqz
      (i32.lt_s (get_local $0) (i32.const 10))
     )
    )
    (block
     (set_local $1
      (i32.mul (get_local $1) (get_local $0))
     )
     (set_local $0
      (i32.add (get_local $0) (i32.const 1))
     )
    )
    (br $label_1)
   )
  )
 )
)
```

Beispiel 13: While-Loop im WebAssembly Textformat

4.3.4 For-Loop

For Statements werden von BabelJS wie folgt in einen AST abgebildet. Die Loop-Parameter unterliegen gewissen Einschränkungen, welche in der Sprachsyntax im Anhang B ersichtlich sind:



Der Loop Aufbau ist identisch mit dem der While-Loop aus dem Abschnitt 4.3.3. Die einzigen Unterschiede sind, dass vor dem bedingungslosen Branch das *update* Statement und vor dem eigentlichen Loop Statement das *init* Statement eingefügt wird. Das daraus resultierende Template ist das Folgende:

```
(init)
(loop $label_1
 (block $label_0
  (br_if $label_0
   (i32.eqz
    (test)
   )
  )
  (block
   (body)
  )
  (update)
  (br $label_1)
 )
)
```

Beispiel 14: WebAssembly Template für For-Loops

4.4 Variablen-Allokation

Bevor der Transpiler mit dem Generieren von Instruktionen beginnen kann, durchläuft er den AST und weist jeder lokalen Variablen und jedem Parameter einen eindeutigen Index zu. Die Indizes werden anschliessend für die Generierung der *get_local* und *set_local* Instruktionen benötigt [11, Indices, S. 14].

4.5 Funktionsaufrufe

Funktionsaufrufe werden von WebAssembly direkt unterstützt. Da eine Funktion möglicherweise noch nicht deklariert wurde, muss Binaryen den Rückgabetyp erhalten. In unserem Fall ist dies entweder *i32* oder *f64*. Der Generator visitiert die Argumente der Reihe nach und führt gegebenenfalls eine numerische Erweiterung A.2 durch. Mit den Binaryen Expression Argumenten kann anschliessend eine *call* Instruktion generiert werden.

Funktionsaufrufe, die den Rückgabewert nicht weiter benötigen, sind Expression Statements 4.2.3. Bei solchen wird eine zusätzliche *drop* Instruktion generiert. Folgender JavaScript Code illustriert diesen Fall:

```
function isEven(num) {
    return num % 2 == 0;
}

function test(num) {
    isEven(num); // drops the result
    return isEven(num);
}
```

Beispiel 15: Funktionsaufruf ohne Zuweisung

Das generierte WebAssembly im Textformat sieht daher so aus:

```
(module
 (type $isEven (func (param i32) (result i32)))
 (export "isEven" (func $isEven))
 (export "test" (func $test))
 (func $isEven (type $isEven) (param $0 i32) (result i32)
  (i32.eqz
   (i32.rem_s
    (get_local $0)
    (i32.const 2)
   )
  )
 )
 (func $test (type $isEven) (param $0 i32) (result i32)
  (drop
   (call $isEven
    (get_local $0)
   )
  )
  (call $isEven
   (get_local $0)
  )
 )
)
```

Beispiel 16: Funktionsaufruf im WebAssembly Textformat

5 Laufzeitunterstützung

Der Call-Wrapper nimmt die Parameter des effektiven WebAssembly Aufrufs entgegen. Er überprüft, ob diese auf die angegebene Signatur passen. Die Array Parameter schreibt er in das Import Objekt und gibt dann deren Adressen weiter. Sobald der Aufruf der WebAssembly Funktion terminiert, werden die Out-Parameter in den Speicher des JavaScript Laufzeitsystems zurückgeschrieben.

Die einzelnen Schritte, die der Call-Wrapper durchläuft, werden hier noch genauer beschrieben.

5.1 Prüfen der Funktionssignatur

Zuerst wird geprüft, ob die Signatur der aufgerufenen Funktion vorhanden ist und ob die Werte der Parameter den vorgegebenen Typen entsprechen. Bei Arrays werden die Typen sämtlicher Elemente überprüft.

5.2 Kopieren der Array Parameter

Die Elemente von Array Parametern werden der Reihe nach in den voralloziierten Speicher kopiert, der später an die WebAssembly Instanz weitergegeben wird. Damit der WebAssembly Code auf die richtigen Stellen im Speicher zugreifen kann, werden die Adressen der Arrays per Parameter übergeben.

5.3 Konvertieren des zurückgegebenen Resultats

Durch den angegebenen Typen des Rückgabewertes kann entschieden werden, ob das zurückgegebene *i32* bzw. *f64* Resultat noch konvertiert werden muss. Bei boolschen Werten ist dies der Fall. Hierbei werden die Werte 1 und 0 zu *true* bzw. *false* übersetzt. Alle anderen Werte ergeben bei der Konvertierung einen Fehler, falls boolean als Rückgabetyp angegeben wurde.

5.4 Out Parameter

In JavaScript werden Arrays immer per Referenz übergeben. Da das WebAssembly in einer eigenen Sanbox mit dazugehörigem Speicher läuft, sind Arrays nicht effektiv zwischen den Laufzeitsystemen geteilt. In-place Updates an übergebenen Arrays werden ausserhalb der Ausführung des WebAssemblies nicht automatisch sichtbar. Die veränderten Daten müssen dazu zuerst zurückkopiert werden. Da ein automatisches Kopieren sämtlicher Arrays mit hohen Kosten verbunden wäre, geschieht dies nicht implizit. Der Benutzer muss solche Array Parameter als Out Parameter markieren. Ein automatisches Erkennen von Veränderungen an Arrayparametern wäre zwar möglich, aber in gewissen Fällen nicht wünschenswert. In dieser Arbeit wurde aus Performancegründen darauf verzichtet.

Eine asynchrone Ausführung des WebAssembly Codes wird in der momentanen Version des Transpilers nicht unterstützt. Somit sind Race Conditions nicht möglich. Dies könnte sich jedoch in späteren Versionen ändern.

5.5 Speicher

5.5.1 Import

Bevor der eigentliche WebAssembly Code ausgeführt werden kann, muss der Speicher importiert werden. Diese Aufgabe übernimmt der Call-Wrapper, welcher bei jedem Funktionsaufruf ausgeführt wird. Die Grösse des Speichers wird bei jedem Aufruf dynamisch berechnet und auf die nächstgrössere Anzahl Pages aufgerundet. Pro Array wird zuerst dessen Länge und anschliessend alle darin enthaltenen Elemente in das Memory kopiert. Die Arrayreferenz im Parameter wird durch die Adresse ersetzt. Hierbei wird nicht die Adresse der Länge, sondern die des ersten Elementes verwendet, wodurch die Länge an der Position -1 zu liegen kommt.

Der Transpiler beachtet, dass die Arrays speziell aligniert werden müssen. *i32* Werte müssen an einer durch vier und *f64* Werte an einer durch acht teilbaren Adresse abgelegt werden. Um das Alignment von *f64* Arrays zu vereinfachen, nutzt die Länge solcher Arrays acht Bytes an Speicherplatz. Vier Bytes werden somit jeweils nicht genutzt und unnötigerweise alloziiert. Dieses Verfahren wird Padding genannt.

5.5.2 Export

Nach der Ausführung des WebAssembly Codes müssen die Werte des Speichers wieder zurückgeschrieben werden. Dem Call-Wrapper muss daher vor der Ausführung mitgeteilt werden, welche Parameter zurückkopiert werden sollen. Durch das fakultative Zurückschreiben können Laufzeitkosten eingespart werden. Das automatische Erkennen von Änderungen wäre grunsätzlich für den Aufrufer einfacher, jedoch bringt es auch einige Nachteile mit sich. Zum einen ist das richtige Implementieren einer solchen Funktionalität schwierig und mit einem enormen Aufwand verbunden. Zum anderen werden alle Änderungen zurückgeschrieben, auch wenn diese eventuell vom Aufrufer nicht benötigt werden. Aus diesen Gründen haben wir uns gegen eine solche Funktionalität entschieden.

6 Auswertung

Dieses Kapitel beschäftigt sich mit der Auswertung der Messresultate. Es gilt zu untersuchen, ob durch die Verwendung unseres Transpilers effektiv eine Verbesserung der Laufzeit erreicht werden kann. Damit die Messresultate repräsentativ sind, wurden sie auf Maschinen mit einer ähnlichen Hardware durchgeführt. Es wurden jeweils die Laufzeiten unterschiedlicher Algorithmen in den Browsern Firefox, Chromium und Microsoft Edge verglichen.

6.1 Testfälle

Die gewählten Testfälle, die gemessen wurden, sind die folgenden:

Testfall	**Beschreibung**
Fibonacci	Rekursive Berechnung der 41sten Fibonacci Zahl
Sum Array	Aufsummieren der Elemente eines Arrays
Count Primes	Die Anzahl an Primzahlen zwischen 0 und 20000000 berechnen
Newtons Method	Verbessern der Lösung einer quadratischen Gleichung mithilfe des Newton-Verfahrens
gcd	Finden des grössten gemeinsamen Teilers der Zahlen 2147483646 und 978
Quicksort	Sortieren eines Arrays mit 2^{10} Elementen
Mergesort	Sortieren eines Arrays mit 2^{10} Elementen

6.2 Setup

Folgende Auflistung zeigt die für die Messungen verwendeten Browser und Compiler:

- Firefox 62.0.3
- Chromium 70.0.3538.67
- Microsoft Edge 42.17134.1.0
- GCC 6.3.0

In diesen Browsern wurden sämtliche Addons ausgeschaltet, da sie einen negativen Einfluss auf die Laufzeit haben könnten.

Die Firefox und Chromium Tests wurden auf dem Betriebssystem Arch Linux und die Microsoft Edge und C++ Tests auf dem Betriebssystem Windows 10 Home Version 1803 durchgeführt. Beide Rechner haben dieselben relevanten Spezifikationen:

- CPU: Intel Core i7-4790k mit 4.4 GHz
- RAM: 16GB DDR3 mit 1600 MHz
- Dedizierte Grafikkarte

Damit vergleichbare Bedingungen für die Ausführungen des JavaScript Codes und des WebAssemblies herschen, wurde eine Warmup Phase eingeführt. Der zu messende Programmcode wird in dieser Phase ohne Zeitmessung ausgeführt. Mögliche Schwankungen der Laufzeit, die bei den ersten Messungen auftreten könnten, werden dadurch minimiert. Die Schwankungen können beispielsweise durch JIT-Kompilierungen entstehen.

Die meisten Browser bauen bei Zeitmessungen eine Ungenauigkeit ein. Damit verhindern sie mögliche zeitbasierte Side-Channel Angriffe [9]. Die Genauigkeit der Messungen liegt daher im Bereich einer Millisekunde. Da die Ausführungen mehrere Sekunden benötigen, spielt diese Ungenauigkeit kaum eine Rolle.

6.3 Resultate

Die Resultate werden hier objektiv genauer beschrieben. Zuerst gehen wir auf den erreichten Speedup der Testfälle ein und danach wird die Varianz der gemessenen Laufzeiten dargestellt. Ausserdem werden die Messresultate zusätzlich mit denen der Programmiersprache C++ verglichen. Zum Schluss wird der Einfluss des Webpack Development Modus erläutert.

6.3.1 Speedup

Die Abbildung 4 zeigt den Speedup[11], den die Browser erreicht haben, gruppiert nach den Testfällen.

Es wurden jeweils fünf Messungen pro Browser, Testfall und Sprache gemacht. Vor diesen Messungen wurde genau eine Warmup Runde durchgeführt. Die Laufzeit des WebAssemblies beinhaltet auch die Ausführungszeit des Call-Wrappers. Da diese nur einen geringen Anteil der Zeit ausmacht, kann sie vernachlässigt werden.

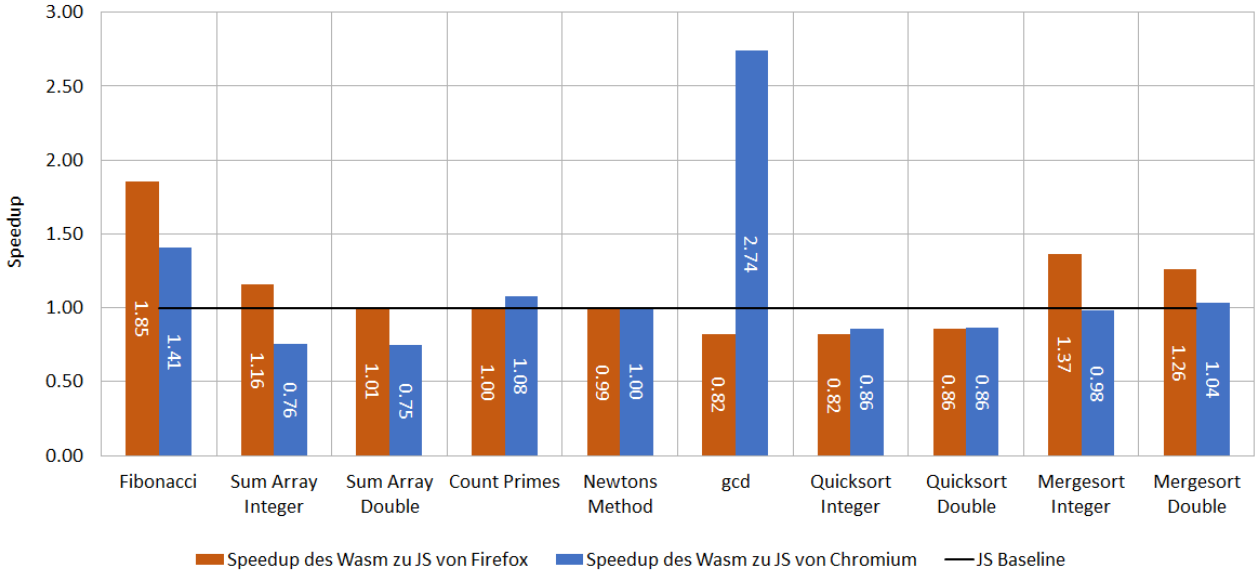


Abbildung 4: Speedup der Benchmarks verglichen nach Browser und Testfall

[11] Der Speedup S entspricht $\frac{T_{old}}{T_{new}}$. T_{old} ist die durchschnittliche Ausführungszeit von JavaScript und T_{new} die von WebAssembly

Ein Testfall ist in beiden Browsern durch die Übersetzung zu WebAssembly schneller geworden. Fünf von zehn Testfällen erreichten unter einem der Browser einen Speedup von mehr als 1. Acht Testfälle wurden in mindestens einem der Browser durch die Transpilierung langsamer. Nur ein Testfall erreicht einen Speedup von mehr als 2.

Die Speedups der vorhergehenden Arbeit [10] blieben leider aus. Es haben sich sowohl die JavaScript- als auch WebAssembly-Engines weiterentwickelt, wodurch nur bei einer erneuten Durchführung der Tests ein wirklicher Vergleich gemacht werden könnte.

Die Gründe für die eher geringen Speedups sind nicht eindeutig bestimmbar. Es ist jedoch auszuschliessen, dass die generierten WebAssembly Instruktionen dafür verantwortlich sind. Führt man die von unserem Transpiler generierten WebAssembly Instruktionen aus, so erhält man ähnliche Zeiten, wie wenn man die des clang Compilers misst. Es ist also anzunehmen, dass die Ausführung des WebAssembly Codes in den Browsern noch Verbesserungspotential aufweist.

Microsoft Edge
Die Testresultate von Microsoft Edge wurden nicht in die obige Abbildung 4 aufgenommen. Sie würden die Lesbarkeit beeinträchtigen. Die Speedups sind in der Tabelle 1 ersichtlich.

Algorithmus	**Speedup**
Fibonacci	15.73
Sum Array Integer	17.29
Sum Array Double	33.73
Count Primes	9.70
Newtons Method	14.69
gcd	5.43
Quicksort Integer	11.03
Quicksort Double	12.61
Mergesort Integer	9.93
Mergesort Double	9.80

Tabelle 1: Speedup Microsoft Edge Browser

Die extrem hohen Speedups sind nicht auf eine speziell tiefe WebAssembly Ausführungszeit zurückzuführen, sondern vielmehr auf aussergewöhnlich langsame Ausführungen des JavaScript Codes. Die Tabelle 2 zeigt die Laufzeiten von Chromium und Microsoft Edge in Millisekunden an. Die Messresultate wurden ohne Aufwärmphase und mit einer Messphase erreicht.

Algorithmus	Chromium		Microsoft Edge	
	JS	Wasm	JS	Wasm
Fibonacci	10828.10	7807.00	193850.30	12188.50
Sum Array Integer	4437.80	5856.40	52218.90	3043.90
Sum Array Double	4359.80	5841.10	170130.90	4964.20
Count Primes	12422.00	11499.80	118478.30	12214.60
Newtons Method	13582.60	13701.40	205673.50	14000.80
gcd	26524.70	9661.00	56274.40	10361.70
Quicksort Integer	5180.70	6153.50	102619.10	9300.70
Quicksort Double	5575.20	6473.60	119648.20	9487.60
Mergesort Integer	9084.70	9347.40	180214.60	18155.50
Mergesort Double	9689.30	9358.80	189203.60	19301.70

Tabelle 2: Laufzeiten in Millisekunden von Chromium und Microsoft Edge im Vergleich

6.3.2 Varianz

Die Varianz[12] wird in dieser Arbeit als Mittel zur Bestimmung der Fluktuation eingesetzt. Durch dessen Ermittlung ist ersichtlich, wie stark die einzelnen Ausführungszeiten vom Mittelwert abweichen. Die Tabelle 3 zeigt die Varianzen von JavaScript und WebAssembly gruppiert nach den Testfällen. Auf Microsoft Edge wurden nicht alle Benchmarks mit mehreren Ausführungsrunden durchgeführt. Diese hätten ca. 15 Minuten pro Benchmark gedauert und das Resultat wäre ähnlich wie bei den vorhergehenden Resultaten ausgefallen.

Bei den Messungen der Varianz wurde jeweils mit einer Aufwärmphase gefolgt von fünf Messphasen gearbeitet. Sowohl die Aufwärm- als auch die Messphasen wurden direkt hintereinander ohne weitere Nutzerinteraktion ausgeführt.

Bei der Berechnung der Varianz wurde mit der Gesamtlaufzeit gearbeitet, welche neben der Laufzeit auch noch die Kompilations und Import- bzw. Exportzeit enthält.

Algorithmus	Varianz Firefox		Varianz Chromium		Varianz Microsoft Edge	
	JS	Wasm	JS	Wasm	JS	Wasm
Fibonacci	167906.64	14.64	48.57	21.03	205707.79	10294.29
Sum Array Integer	124.56	262.00	8.24	62.64	9843.13	59.98
Sum Array Double	82.64	58.80	3.72	4.60	55512.66	11.08
Count Primes	36.40	5.20	101.90	58.82	-	-
Newtons Method	7.76	40.00	23.19	38.49	-	-
gcd	162.96	470.96	129.27	1.91	-	-
Quicksort Integer	17.76	813.84	2.58	1626.74	-	-
Quicksort Double	102.16	1008.64	9.84	2182.25	-	-
Mergesort Integer	188.64	3437.84	112.17	3499.70	-	-
Mergesort Double	243.84	27371.20	73.53	3057.52	-	-

Tabelle 3: Varianz der Benchmarks

[12]Das Quadrat der Standardabweichung.

Aus den Werten ist ersichtlich, dass WebAssembly vor allem bei Berechnungen ohne Speicherzugriffe eine tiefere Varianz aufweist. Sobald mit Speicherzugriffen gearbeitet wird, verschlechtert sich diese im Gegensatz zu JavaScript stark. Dieses Resultat ist unerwartet, da bei der vorhergehenden Masterarbeit [10] vor allem eine geringere Fluktuation bei WebAssembly gezeigt wurde. Die Vermutung liegt nahe, dass die zusätzliche Kompilationszeit die starken Schwankungen der Varianz verursacht. Für die hohen Varianzen bei Speicherzugriffen könnte die zusätzliche Import- bzw. Exportlogik des Call-Wrappers 5.5 verantwortlich sein.

6.3.3 Vergleich zu C++

Zum Vergleich wurden alle Benchmarks in C++ implementiert. Hierbei wurden die Speedups, die in der Abbildung 5 dargestellt sind, gemessen. Die Laufzeit des C++ Codes dient als Ausgangslage für die Berechnung der Speedups.

Anstelle von Arrays wurde die C++ Datenstuktur *vector* genutzt. Die Datenstruktur *array* konnte nicht eingesetzt werden, da sie sämtliche Werte auf dem Stack ablegen würde. Dieser hat nicht genug Speicherplatz für 2^{10} Elemente. Der C++ Code wurde mit der Optimierungsstufe drei[13] kompiliert. Diese Stufe bietet die höchste Laufzeiteffizienz und hält sich gleichzeitig noch an den Standard [1].

Die Zeiten der WebAssembly- und JavaScript-Aufführungen sind die Durchschnittswerte der Browser Firefox und Chromium. Alle Messungen ergeben sich aus einer Messphase. Auf Aufwärmphasen wurde verzichtet.

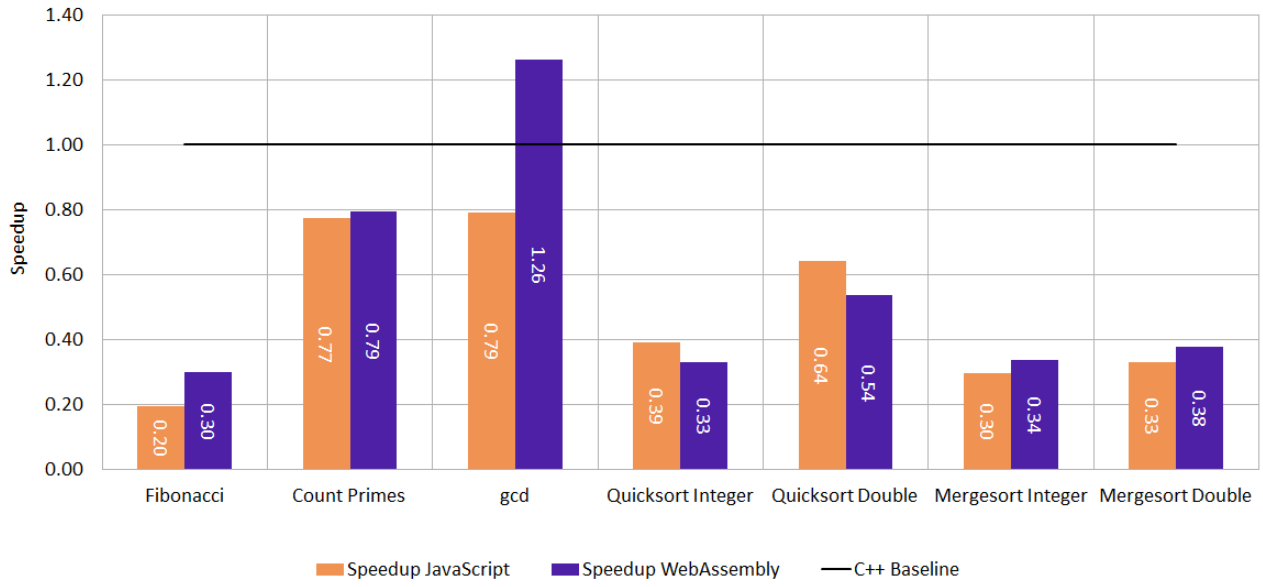


Abbildung 5: Vergleich Speedup von WebAssembly, JavaScript und C++

Die Abbildung 5 zeigt, dass ein ausgereifter Optimizer enorme Speedups mit sich bringen kann. Bei den Benchmarks Sum Array Integer, Sum Array Double und Newtons Method konnte der C++ Compiler die Berechnungen sehr stark wegoptimieren. Dadurch erreichte die Ausführung bei diesen eine Laufzeit von weniger als einer Millisekunde. Die Resultate zeigen, dass die WebAssembly Compiler-Infrastruktur noch viel mehr optimieren könnte. Binaryen könnte ähnlich wie der C++ Compiler gewisse Berechnungen zur Kompilationszeit vereinfachen.

[13] Setzen des -O3 Flags bei der Kompilation.

6.3.4 Webpack Development Modus

Während dem Messen der Laufzeiten ist uns immer wieder aufgefallen, dass bereits kleinste Änderungen am Setup grosse Auswirkungen auf die Performance haben. Webpack[14] verwendet beispielsweise im Development Modus *eval*[15] für die Generierung des Bundles. Durch *eval* ist die JavaScript-Engine in der Wahl der Optimierungen eingeschränkt [12]. Zum Vergleich wurden einige Benchmarks unter Firefox mit eingeschaltetem Development Modus erneut durchgeführt. Ein Vergleich der Durchschnittsgeschwindigkeiten aus fünf Messungen ist in der Abbildung 6 ersichtlich.

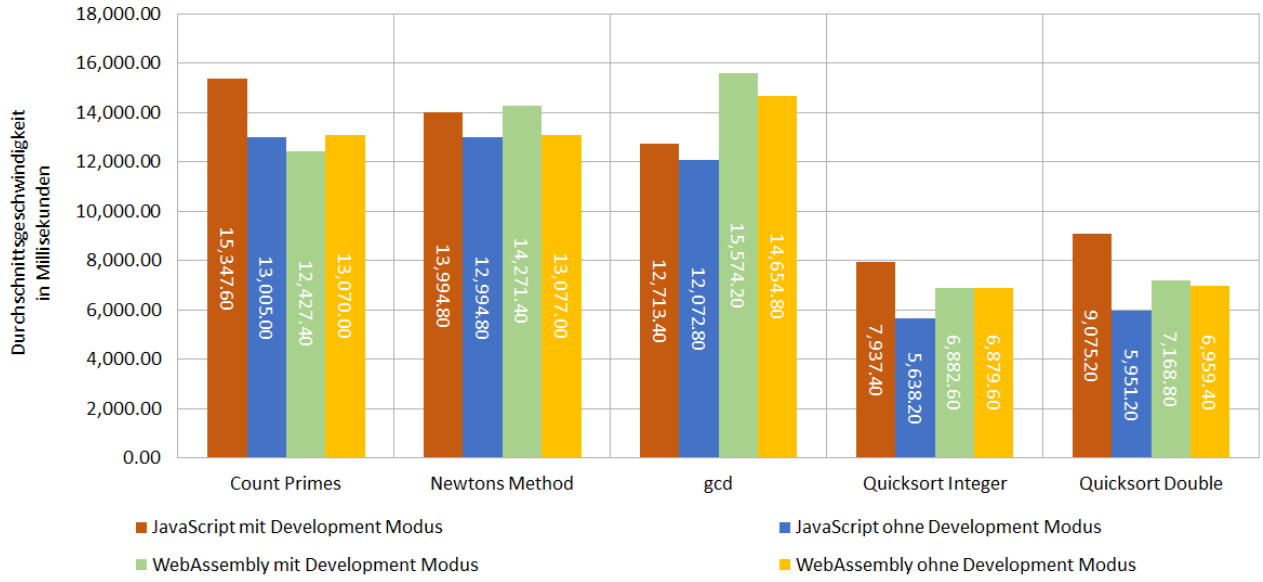


Abbildung 6: Vergleich der Durchschnittsgeschwindigkeiten mit und ohne Development Modus

Eine grosse Abweichung der Webassembly Laufzeit konnte nur bei den Benchmarks gcd und Newtons Method festgestellt werden. Die JavaScript Ausführungszeit hingegen ist bei allen Benchmarks um 1 bis 2 Sekunden angestiegen.

6.4 Fazit

Die Arbeit hat gezeigt, dass sich das Übersetzen zu WebAssembly nur in gewissen Fällen lohnt. Entwickler sollten vor dem Umstieg auf WebAssembly Messungen durchführen und die Laufzeit ausführlich vergleichen. Es ist denkbar, dass sich die Laufzeit des WebAssembly Binärcodes in Zukunft weiter verbessert. Die Ladezeit von WebAssembly in der V8 JavaScript Engine wurde beispielsweise Mitte dieses Jahres enorm verkürzt [7].

Die hohen Varianzen sind sehr unerwartet. Die Vermutung liegt nahe, dass die zusätzliche Export- und Importzeit die Schwankungen verursacht. Damit diese Vermutung verifiziert werden könnte, müssten die Tests erneut durchgeführt werden, wobei die Varianz nur auf der effektiven Ausführungszeit berechnet werden müsste.

Aus den Vergleichen mit C++ wird klar, dass die WebAssembly Compiler-Infrastruktur mehr optimieren könnte. Dies war von Anfang an anzunehmen und wird sich mit Sicherheit in Zukunft ändern.

[14]Der für dieses Projekt verwendete JavaScript-Bundler. Dieser nimmt mehrere JavaScript Datein entgegen und kombiniert sie in eine einzelne Datei.

[15]Eine Funktion, die JavaScript Code als String entgegennimmt und diesen ausführt.

7 Schlussfolgerung

Zusammenfassend können wir sagen, dass es möglich ist JavaScript zur Laufzeit im Browser in WebAssembly umzuwandeln. Der Protoyp ist fähig ein Subset von JavaScript zu übersetzen. Ausserdem erfüllt er die meisten nicht funktionalen Anforderungen. Bei der Performance konnte der Transpiler nur in gewissen Fällen punkten. Einige Fälle lassen sich mit WebAssembly substantiell beschleunigen. Andere weisen eine ähnliche Laufzeit auf, wie ihr JavaScript Äquivalent. Dies liegt aber nicht an ungünstigen Instruktionen, die generiert werden. Vielmehr liegt es an den aktuellen Browsern, die das WebAssembly zu wenig optimiert ausführen.

Die Arbeit hat gezeigt, dass sich rechenintensive Programmbeispiele optimal auf die WebAssembly Sprache abbilden lassen. Gewisse Konstrukte wie zum Beispiel die Objektorientierung sind dafür eher weniger geeignet. Der Grundssatz des Subsets wurde also durch die Umsetzung der Arbeit bestätigt. Somit wurden sinnvolle funktionale Anforderungen gewählt und diese auch erfüllt.

Die übersetzbare Sprache ist noch sehr eingeschränkt. Andere Compiler unterstützen um einiges komplexere Sprachen [10] [16] [13]. Dennoch können die wichtigsten Konzepte, die in prozeduralen Sprachen vorkommen, übersetzt werden.

Würden wir das Projekt nochmals durchführen, so würden wir eventuell auf Binaryen verzichten und unseren WebAssembly Generator selber schreiben. Einerseites konnten wir durch die Library viel Zeit und Programmcode einsparen, andererseits hatten wir immer wieder Probleme mit der API. Bei einer erneuten Durchführung würden wir eine vertiefte Evaluation durchführen und danach einen Entscheid basierend auf der aktuellen Problemstellung treffen.

7.1 Ausblick

In Zukunft könnte die Sprache, die übersetzt wird, erweitert werden. Dabei ist es wichtig, dass man sich strikt an den Grundsatz der Sprache hält. Ansonsten würde die Gefahr bestehen, dass man Sprachkonstrukte einführt, die die Performance negativ beeinträchtigen.

Die aktuelle Typinferenz könnte erweitert werden, sodass gar keine Typangaben durch den Benutzer nötig sind. Sämtliche Typen wären dann spätestens beim Aufruf einer Funktion bestimmbar. Der Benutzer würde dadurch entlastet werden und die API würde sich vereinfachen.

Neben diesen Erweiterungen des Transpilers, denken wir, dass sich die WebAssembly Infrastuktur noch verbessern könnte. Die Binaryen Library könnte beispielsweise den generierten WebAssembly Code noch mehr optimieren und einige Konzepte in der Dokumentation ausführlicher beschreiben. Die JIT-Compiler der Browser, die das WebAssembly in Maschinencode umwandeln, scheinen noch zu wenig zu optimieren. Die Ausführung des JavaScript Codes ist in den meisten Fällen schneller. Hier können sich die Browser Engines noch verbessern.

Um interessierte Webentwickler für das Thema WebAssembly und unseren Transpiler zu begeistern, könnte eine Vergleichswebseite erstellt werden. Auf dieser könnte unter anderem die Laufzeit des JavaScript und des WebAssembly Codes verglichen werden. Hierbei könnte die bereits bestehende Benchmark Infrastruktur wiederverwendet werden.

Damit der Transpiler in anderen Projekten verwendet werden könnte, müsste dieser unter npm[16] veröffentlicht werden. Dies würde die Einbindung und das Experimentieren mit dem Transpiler vereinfachen.

[16] Node package manager, ein Repository zur Verwaltung von JavaScript Libraries

Anhang

A Erläuterung Language Set

A.1 Typen

Die im Language Set 3.2 angegebenen Typen können mit den WebAssembly Typen *i32* bzw. *f64* abgebildet werden. Boolsche Werte werden mit einem *i32* Wert dargestellt. `true` wird auf 1 und `false` auf 0 gemappt. Arrays fungieren als Pointer und haben daher den Typen *i32*. Die Grösse einer Adresse wurde vom WebAssembly Standard festgelegt [11, Indices, S. 14].

A.2 Numerische Erweiterung

Binäre Operatoren und Shorthand Assignments unterstützen Operanden von unterschiedlichen Typen. Die WebAssembly Instruktionen benötigen jedoch Werte desselben Typs. Um die Typkompatibilität während der Ausführung sicherzustellen, ist in gewissen Fällen eine Typkonvertierung nötig. Der Wert des kleineren Typen muss zu einem Wert des grösseren Typen umgewandelt werden.

A.3 Abweichungen der binären Operatoren zu JavaScript

Division Durch die strikte Typisierung der Subsprache ergibt das Teilen von zwei *i32* Werten wieder einen *i32* Wert. Das Resultat einer Division wird gegen 0 gerundet.

Modulo Das Ausführen einer Modulo Operation ist nur auf *i32* Werten unterstützt. WebAssembly verfügt über keine Modulo Instruktion, die mit *f64* Werten rechnet.

A.4 Pre- und Post-Increment/Decrement

Pre- und Post-Increments sowie Decrements können nur als Top-Level Statements verwendet werden. Das bedeutet, dass sie nicht in anderen Expressions vorkommen können. Grund für diese Einschränkung ist die Template-based Code Generation. Bei dieser Art von Generierung konzentriert man sich nur auf den aktuellen AST Node und kümmert sich nicht um den umgebenden Kontext (Parent- oder Siblingnodes). Somit ist bei der Generierung nicht klar, ob sich die aktuelle Increment/Decrement Expression in einer anderen Expression befindet. Ohne dieses Wisssen ist es nicht möglich zu entscheiden, ob das Resultat auf dem Laufzeitstack bleiben soll oder nicht. Hätte man diese Art von Expression zugelassen, so wäre die Codegenerierung um einiges unübersichtlicher geworden.

Pre- und Post-Increment/Decrement sind sowohl bei Variablen als auch bei Arrayelementen erlaubt.

```
value++;
--value;
array[0]++;
--array[0];
```

Beispiel 17: Erlaubte Pre- und Post-Increment/Decrement Beispiele

```
value++ + 1;
array[0]++ + 1;
array++;
```

Beispiel 18: Nicht erlaubte Pre- und Post-Increment/Decrement Beispiele

A.5 Assignment Expression

Zuweisungen sind nur als eigene Statements erlaubt. Sie können nicht in anderen Expressions vorkommen. Diese Einschränkung existiert aus dem im Abschnitt A.4 genannten Grund.

```
var x = 1;
x = 2;
```

Beispiel 19: Erlaubtes Assignment Beispiel

```
var x;
var y = 1 + (x = 2);
```

Beispiel 20: Nicht erlaubtes Assignment Beispiel

A.6 Funktionsaufrufe

Bei einem Funktionsaufruf kann der Rückgabewert ignoriert bzw. nicht weiter verwendet werden. Ausserdem ist es möglich Funktionen rekursiv aufzurufen.

```
function fibonacci(current) {
    if (current <= 2) {
        return 1;
    }

    return fibonacci(current - 2) + fibonacci(current - 1);
}
```

Beispiel 21: Rekursives Fibonacci Beispiel

A.7 Return Statement

Funktionen müssen zwingend einen Wert vom Typen Boolean, 32-bit Integer oder 64-bit Float zurückgeben. Void wird als Rückgabetyp nicht unterstützt. WebAssembly würde dies nicht zulassen. Arrays können nicht direkt zurückgegeben werden. Änderungen an Arrays werden nur dann sichtbar, wenn es sich bei dem Array um einen Out Parameter handelt. Siehe dazu Kapitel 5.4.

```
function func() {
    return 10;
}
```

Beispiel 22: Erlaubtes Return Statement Beispiel

```
function func() {
    return 'Test';
}

function otherFunc() {
    var x = [1, 2, 3];
    return x;
}
```

Beispiel 23: Nicht erlaubtes Return Statement Beispiel

A.8 If-else Statement

Es ist möglich, ein if Statement ohne ein dazugehöriges else Statement zu verwenden. Else-if Branches sowie Branches, die nur aus einem einzigen Statement bestehen, werden unterstützt.

```
var x;

if (...) {
    x = 1;
}

if (...) {
    x = 2;
} else {
    x = 3;
}

if (...) {
    x = 4;
} else if (...) {
    x = 5;
}

if (...)
    x = 6;
else if (...)
    x = 7;
else
    x = 8;
```

Beispiel 24: Erlaubte if-else Verzweigungs Beispiele

A.9 While Statements

Es werden sowohl einzeilige, als auch mehrzeilige While Statements unterstützt.

```
var i = 0;

while (i < 10) {
    i++;
}

var j = 0;

while (j < 10) j++;
```

Beispiel 25: Erlaubte While Statement Beispiele

A.10 For Statements

Es werden sowohl einzeilige, als auch mehrzeilige For Statements unterstützt. Alle drei Expressions der For-Loop sind optional, wobei bei einer leeren Bedingung die Schleife endlos läuft. Die Art von Expressions, die übergeben werden dürfen, ist eingeschränkt. Diese Limitation entsteht durch die Unterscheidung zwischen dem Expression- und Statementstack in der Implementation. Weitere Informationen dazu befinden sich in der Syntaxspezifikation im Anhang B. For-Of und For-In Statements werden nicht unterstützt.

```
for (var i = 0; i < 10; i++) {
    loopCounter++;
}

for (var i = 0; i < 10; i++) loopCounter++;

for (;;) loopCounter;
```

Beispiel 26: Erlaubte For-Loop Beispiele

```
for (var element in array) { }

for (var element of array) { }
```

Beispiel 27: Nicht erlaubte For-Loop Beispiele

A.11 Variablen Deklarationen

Deklarationen werden, wie es in JavaScript üblich ist, gehoisted. Das bedeutet, sie werden an den Anfang der Funktion verschoben. Variablen sind somit zu jedem Zeitpunkt deklariert, solange mindestens eine Deklaration existiert. Bevor eine Variable verwendet werden kann, muss ihr ein Wert zugewiesen werden. Variablen können nie den Wert *undefined* annehmen. Nach der initialen Zuweisung darf sich der Typ einer Variablen nicht mehr ändern. Das spezielle Verhalten der *const* und *let* Deklarationen wird nicht angeboten.

```
function func() {
    x = 10;
    var x;

    y = 10;

    if (false) {
        var y;
    }
}
```

Beispiel 28: Variablen Hoisting Beispiele

A.12 Arrays

Arrays können nur per Parameter übergeben werden. Änderungen an Arrays werden im Speicher von JavaScript nur sichtbar, wenn das Array als Out Parameter 5.4 deklariert wurde. Sie können weder vergrössert noch verkleinert werden. Das Property *length* wird nur lesend über einen direkten Aufruf unterstützt. Neben dem *length* Property wird nur der `[]` Operator auf Arrays angeboten. Das Lesen und Schreiben eines Wertes ausserhalb der Arraygrösse kann zu Speicherfehlern führen. Das genaue Verhalten ist nicht definiert.

Mehrdimensionale Arrays werden in unserer Sprache nicht unterstützt. In JavaScript werden solche Arrays als Jagged Arrays[17] abgelegt. Die Indirektion der Zugriffe hat einen beachtlichen Einfluss auf die Performance. Dies würde dem Prinzip des Language Sets widersprechen und wurde deshalb weggelassen.

```
function func(array) {
    var length = array.length;

    if (length >= 1) {
        array[0] = 5;
    }

    return array[0];
}
```

Beispiel 29: Erlaubtes Array Beispiel

[17] Ein Array von Arrays bei dem die einzelnen Arrays unterschiedlich gross sind.

```
function func() {
    array.push(4);
    var x = array.pop();

    var length = array['length'];

    var y = array[array.length];
    array[array.length] = 5;

    func2([1, 2, 3]);

    if ([1, 2, 3][0] == 4) { }

    array[0][1];

    return 0;
}
```

Beispiel 30: Nicht erlaubtes Array Beispiel

A.12.1 Array-Literale

Array-Literale sind in einem separaten Git-Branch verfügbar, da sie nur mit einigen Einschränkungen funktionieren. Es wurde noch kein dynamisches Vergrössern bzw. Verkleinern implementiert und Array-Literale können nur bei direkten Zuweisungen verwendet werden. Die momentane Implementation ist nicht performant, da statisch für jedes Literal Speicher alloziert wird, welcher eventuell gar nie benötigt wird. Des Weiteren ist ihr Nutzen fragwürdig, da Arrays auch per Parameter übergeben werden können. Aus diesen Gründen wurden die Array-Literale nicht zur aktuellsten Version hinzugefügt.

```
function func() {
    var array2 = [1, 2, 3];

    var array3;
    array3 = [1, 2, 3];
}
```

Beispiel 31: In speziellem Git-Branch erlaubtes Array-Literal Beispiel

B JS2Wasm EBNF

Program	=	FunctionDeclaration { FunctionDeclaration }.
FunctionDeclaration	=	FunctionHead BlockStatement.
FunctionHead	=	`"function"` Identifier `"("` ParameterList `")"` .
ParameterList	=	`[` Identifier { `","` Identifier } `]`.
BlockStatement	=	`"{"` { Statement } `"}"`.
Statement	=	`";"` | ExpressionStatement | IfStatement | ReturnStatement | VariableDeclaration | WhileStatement | ForStatement.
ExpressionStatement	=	(AssignmentExpression | Expression | ShorthandAssignmentExpression | UpdateExpression) `";"`.
IfStatement	=	`"if"` `"("` Expression `")"` (BlockStatement | Statement) `[` `"else"` (BlockStatement | Statement) `]`.
ReturnStatement	=	`"return"` Expression `";"`.
VariableDeclaration	=	`"var"` VariableDeclaratorList.
VariableDeclaratorList	=	VariableDeclarator { `","` VariableDeclarator } `";"`.
VariableDeclarator	=	Identifier `[` `"="` Expression `]`.
WhileStatement	=	`"while"` `"("` Expression `")"` (BlockStatement | Statement).
ForStatement	=	`"for"` `"("` `[` (AssignmentExpression | VariableDeclaration | ShorthandAssignmentExpression | UpdateExpression) `]` `";"` `[` Expression `]` `";"` `[` (AssignmentExpression | ShorthandAssignmentExpression | UpdateExpression) `]` `")"` (BlockStatement | Statement).
AssignmentExpression	=	Designator `"="` Expression.
ShorthandAssignmentExpression	=	Designator ShorthandAssignmentOperator Expression.
ShorthandAssignmentOperator	=	`"+="` | `"-="` | `"*="` | `"\="`.
UpdateExpression	=	(Designator UpdateExpressionOperator) | (UpdateExpressionOperator Designator).
UpdateExpressionOperator	=	`"++"` | `"--"`.
Expression	=	LogicTerm { `"||"` LogicTerm }.
LogicTerm	=	LogicFactor { `"&&"` LogicFactor }.
LogicFactor	=	SimpleExpression { CompareOperator SimpleExpression }.
CompareOperator	=	`"=="` | `"!="` | `"<"` | `"<="` | `">"` | `">="`.
SimpleExpression	=	Term { (`"+"` | `"-"`) Term }.
Term	=	Factor { (`"*"` | `"/"` | `"%"`) Factor }.
Factor	=	Operand | UnaryExpression | `"("` Expression `")"`.
UnaryExpression	=	(`"!"` | `"+"` | `"-"`) Factor.
Operand	=	Literal | Designator | CallExpression.
CallExpession	=	Identifier `"("` ParameterList `")"`.
Designator	=	Identifier | Identifier `[` Expression `]`.
Literal	=	Integer | Double | Boolean.

Identifier [14, Identifier Names, S. 207], *Integer* [14, Numeric Literals, S. 209], *Double* [14, Numeric Literals, S. 209] und *Boolean* [14, Boolean Literals, S. 209] verwenden dieselben Regeln wie die des ECMAScript 262 Standards.

C API

C.1 Transpiler

- Erzeugen einer Instanz der Transpiler Klasse.

```
new Transpiler(): void
```

- Erzeugen einer Instanz der Transpiler Klasse. Hierbei wird die TranspilerHook Instanz genutzt.

```
new Transpiler(hook: TranspilerHooks): void
```

- Erstellt ein Mapping zwischen dem Funktionsnamen und Returntyp bzw. Parametertypen. Die Parametertypen müssen in der gleichen Reihenfolge angegeben werden, wie sie in der Funktionsdeklaration vorkommen.

```
setSignature(name: string, returnType: WebAssemblyType, ...
    parameterTypes: WebAssemblyType[]): Transpiler
```

- Transpiliert den übergebenen String. Die Signatur wird hierbei auf ihre Korrektheit überprüft.

```
transpile(content: string): CallWrapper
```

C.2 WebAssemblyType

- Der WebAssembly *i32* Typ:
  INT_32
- Der WebAssembly *f64* Typ:
  FLOAT_64
- Ein Array mit Werten vom Typ *i32*:
  INT_32_ARRAY
- Ein Array mit Werten vom Typ *f64*:
  FLOAT_64_ARRAY
- Der Boolean Typ:
  BOOLEAN

C.3 CallWrapper

- Namen der initialen Funktion setzen.

```
setFunctionName(functionName: string): CallWrapper
```

- Angeben der Parameter, dessen veränderte Werte wieder zurück ins JavaScript geschrieben werden sollen.

```
setOutParameter(...outParameter: any[]): CallWrapper
```

- Die initiale Funktion mit den angegebenen Parametern aufrufen.

```
call(...parameters: any[]): any
```

C.4 TranspilerHooks

Das TranspilerHooks Interface enthält Hook-Methoden, welche vom Transpiler und CallWrapper aufgerufen werden. Die *NullTranpilerHooks* implementiert das Interface und enthält leere Implementationen. Der leere Konstruktor der Transpiler Klasse nutzt standardmässig diese Implementation.

- Wird vor der Transpilation aufgerufen.

```
beforeCompilation(): void
```

- Wird nach der Transpilation aufgerufen.

```
afterCompilation(): void
```

- Wird vor dem Import aufgerufen.

```
beforeImport(): void
```

- Wird nach dem Import aufgerufen.

```
afterImport(): void
```

- Wird vor der Ausführung des WebAssemblies aufgerufen.

```
beforeExecution(): void
```

- Wird nach der Ausführung des WebAssemblies aufgerufen.

```
afterExecution(): void
```

- Wird vor dem Export aufgerufen.

```
beforeExport(): void
```

- Wird nach dem Export aufgerufen.

```
afterExport(): void
```

D Abbildungsverzeichnis

E Tabellenverzeichnis

F Beispielverzeichnis

G Literaturverzeichnis

[1] GCC Optimize Options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Zugegriffen am 14.12.2018.

[2] Babel. Babel Parser. https://babeljs.io/docs/en/babel-parser/. Zugegriffen am 04.12.2018.

[3] Wikipedia contributors. Tree traversal. https://en.wikipedia.org/wiki/Tree_traversal. Zugegriffen am 08.12.2018.

[4] LLVM Developer Group. The LLVM Compiler Infrastructure. http://llvm.org/. Zugegriffen am 04.12.2018.

[5] WebAssembly Community Group. Binaryen. https://github.com/WebAssembly/binaryen. Zugegriffen am 04.12.2018.

[6] WebAssembly Community Group. Semantics. https://webassembly.org/docs/semantics/. Zugegriffen am 04.12.2018.

[7] Clemens Hammacher. Liftoff: a new baseline compiler for webassembly in v8. https://v8.dev/blog/liftoff. Zugegriffen am 09.12.2018.

[8] Microsoft. Typescript. https://blog.rust-lang.org/2016/12/22/Rust-1.14.html. Zugegriffen am 04.12.2018.

[9] Mozilla. Reduced time precision. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now#Reduced_time_precision. Zugegriffen am 13.12.2018.

[10] Micha Reiser and Luc Bläser. Accelerate Javascript Applications by Cross-Compiling to Webassembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2017, pages 10–17, New York, NY, USA, 2017. ACM.

[11] Andreas Rossberg. WebAssembly Specification. W3C, Oktober 2018.

[12] Kyle Simpson. You don't know js: Scope & closures. https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/ch2.md#performance. Zugegriffen am 06.12.2018.

[13] Rust Core Team. Announcing Rust 1.14. https://blog.rust-lang.org/2016/12/22/Rust-1.14.html. Zugegriffen am 04.12.2018.

[14] Brian Terlson. ECMAScript standard. ECMA ECMA-262, Juni 2018.

[15] Luke Wagner. Host Binding Proposal for WebAssembly. https://github.com/WebAssembly/host-bindings/blob/master/proposals/host-bindings/Overview.md. Zugegriffen am 07.12.2018.

[16] Alon Zakai. Emscripten. https://github.com/kripken/emscripten. Zugegriffen am 04.12.2018.