

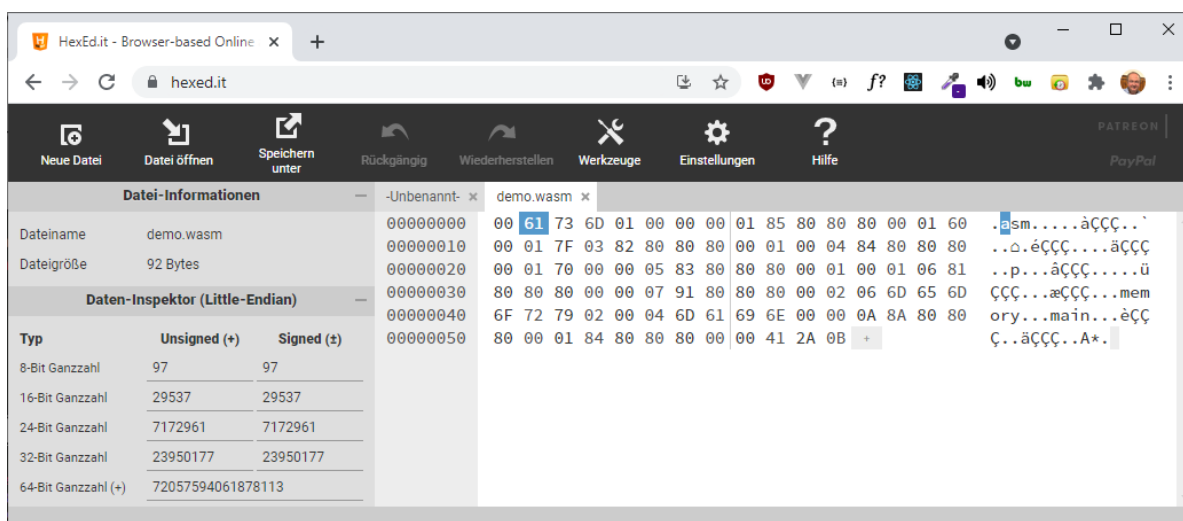
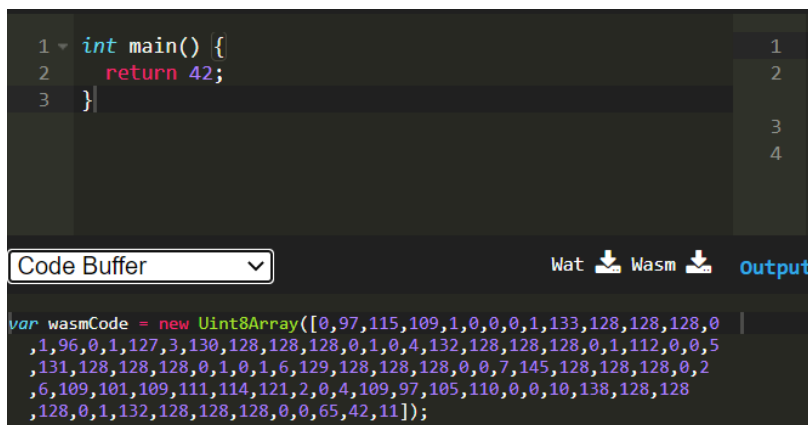
Nachdem wir im ersten Arbeitsblatt erste Schritte mit WebAssembly unternommen haben, wird es Zeit sich etwas näher mit WebAssembly zu beschäftigen. Vermutlich hattest du schon einige Fragen. Zum Beispiel was ist der Unterschied zwischen `.wasm` und `.wat` Files?



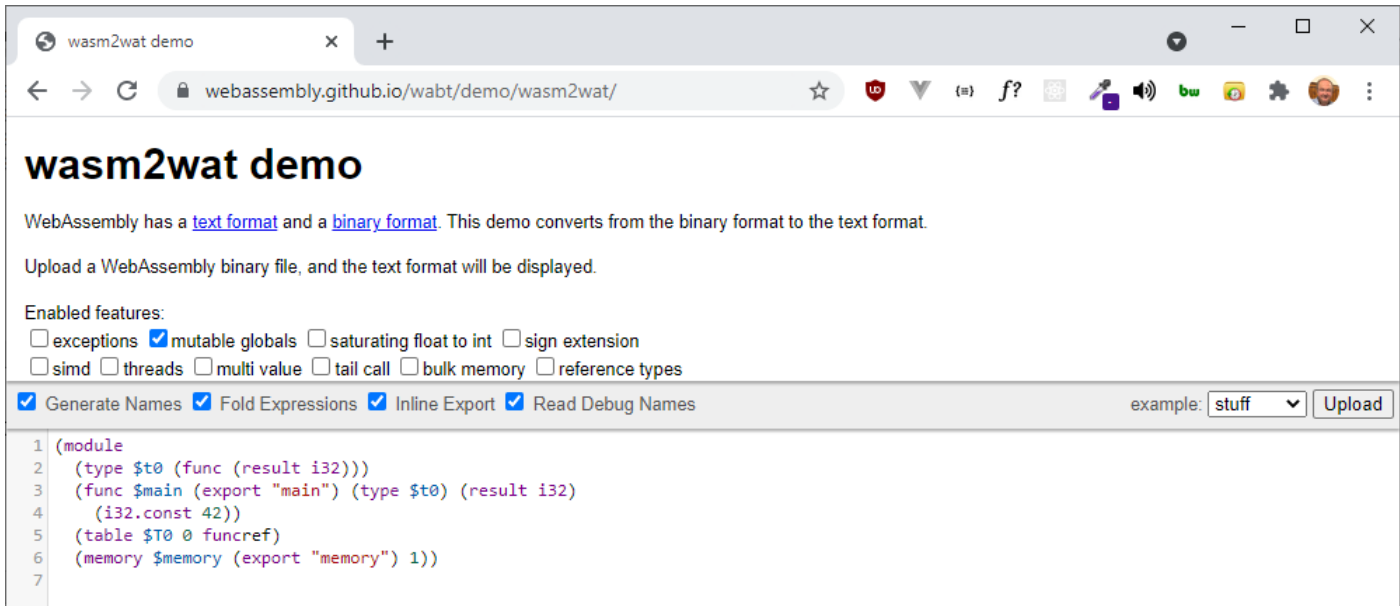
Nun, es gibt zwei Formen von WebAssembly Dateien, die sich 1:1 entsprechen.

- `.wasm` ist das binäre Format von WebAssembly
- `.wat` ist das für uns Menschen lesbare Format von WebAssembly

Das `.wat` Format siehst du oben als Text Format, das `.wasm` Format siehst du unten als Folge von 92 Bytes.



Die beiden sind 1:1 Abbildungen von einander, also kann zwischen den Formaten hin und her konvertiert werden.



Mehr dazu im WebAssembly Binary Toolkit: <https://github.com/WebAssembly/wabt>

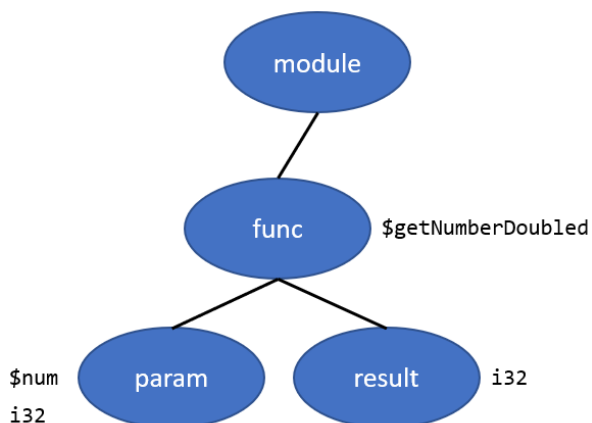
Sehen wir uns die Wat Syntax kurz an (obiges Bild). Lauter Klammern! Das ist schräg! Aber man kann deutlich sehen: Die grundlegende Einheit von WebAssembly ist ein Modul (**module**). In einem Modul können wir Funktionen erstellen, importieren und exportieren.

Ein Modul wird als eine riesige **S-Expression** dargestellt. Was ist das? S-Expressions sind ein sehr altes und sehr einfaches Textformat (Sprache Lisp) zur Darstellung von Bäumen, und so können wir uns ein Modul als einen Baum von Knoten vorstellen, die die Struktur des Moduls und seinen Code beschreiben¹.

Jeder Knoten im Baum befindet sich innerhalb eines Paares von Klammern (**()**). Das erste Wort innerhalb der Klammer sagt dir, um welche Art von Knoten es sich handelt, und danach folgt eine durch Leerzeichen getrennte Liste von entweder Attributen oder Child-Knoten. Das bedeutet also der WebAssembly S-Ausdruck:

```
(module
  (func $getNumberDoubled (param $num i32) (result i32))
)
```

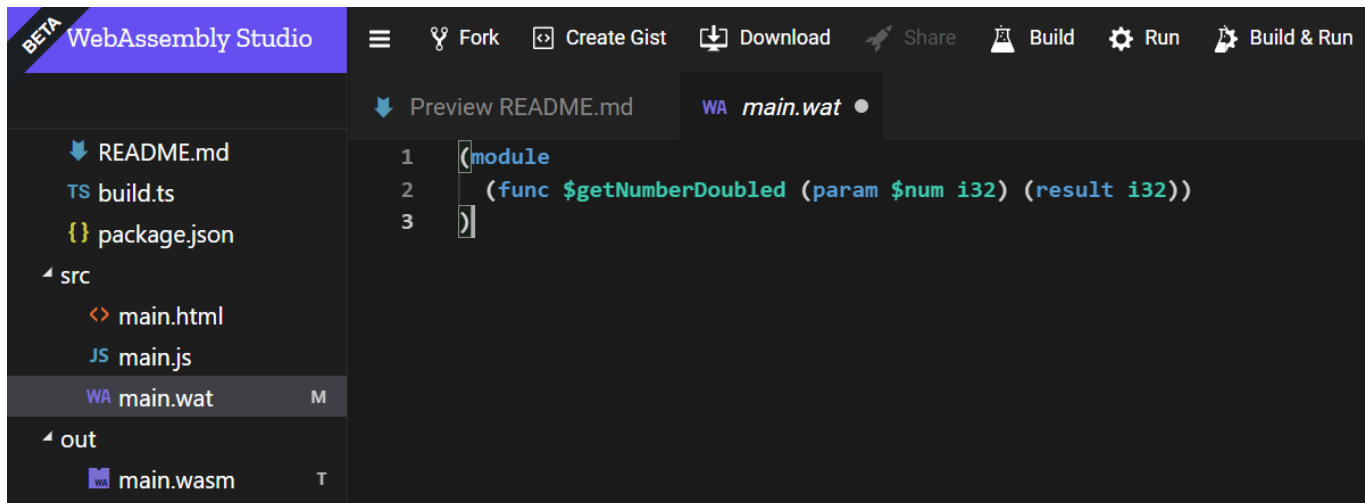
lässt sich darstellen als:



¹ Siehe: <https://en.wikipedia.org/wiki/S-expression>

Wie du auch erkennen kannst, beginnen Namen und Parameter mit einem `$`-Zeichen. Da WebAssembly mit Bitfolgen direkt arbeitet, müssen wir immer auch einen Typ dazu angeben, sonst wäre ja nicht bekannt wie lang die Bitfolge ist. So ist eben `i32` Integer 32 (4 Bytes) und `i64` Integer 64 (8 Bytes). Erinnere dich: WebAssembly kennt als Datentypen nur Integer und Float!

Gehe auf: <https://webassembly.studio/> und erstelle ein leeres Wat Projekt. Löschen den Code und tippe:



Funktionen können Parameter haben (in WebAssembly `locals` genannt. Wir können diese Parameter mit `get_local` bzw `local.get` ansprechen.

```
(module
  (func $getNumberDoubled (param $num i32) (result i32)
    get_local $num
    return
  )
)
```

Hier wird natürlich nichts verdoppelt. Im ersten Schritt wird die Zahl wieder zurückgeliefert.

Damit das auch funktioniert, müssen wir die Funktion `$getNumberDoubled` exportieren.

```
(module
  (func $getNumberDoubled (param $num i32) (result i32)
    get_local $num
    return
  )
  (export "getNumberDoubled" (func $getNumberDoubled))
)
```

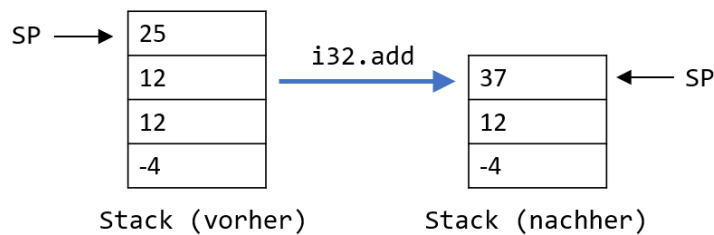
Vergiss nicht zu speichern! Ändere auch `main.js`!

```
fetch('../out/main.wasm').then(response =>
  response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
  instance = results.instance;
  document.getElementById("container").textContent = instance.exports.getNumberDoubled(3);
}).catch(console.error);
```

Bevor wir den Code für das Verdoppeln schreiben können, müssen wir verstehen, wie WebAssembly ausgeführt wird. Wir müssen also die virtuelle Maschine kennenlernen (nur ein wenig).

Die Art der WebAssembly virtuellen Maschine (VM)

Die VM ist eine auf einem Stack basierte Maschine². Du kennst ja Stacks aus C# (Stack) und JavaScript (dort sind es Arrays). Stacks sind LIFO Datenstrukturen (**L**ast **I**n **F**irst **O**ut). Hier bedeutet es, dass es einen Speicherbereich gibt, der wie ein Stack funktioniert, wo Variable gespeichert und gelesen werden, während das Programm von der VM abgearbeitet wird. Die WebAssembly Befehle (Operation Codes) manipulieren unter anderem den Stack. So addiert `i.32.add` die beiden obersten Elemente des Stacks³.



SP = Stackpointer (zeigt auf das oberste Element)

Beachte, dass die Berechnung in Postfixnotation (auch Reverse Polish Notation, RPN genannt⁴) erfolgt: $25, 12, * \rightarrow 37$. Infixnotation wäre $15 + 12 \rightarrow 37$.

Jede Expression in Infix Notation kann auf Postfix Notation und vice versa umgeschrieben werden. Der Vorteil der RPN ist, dass keine Klammern benötigt werden und die Reihenfolge der Operationen eindeutig ist.

Beispiel:

Infix: $(A+B)*C - (D-E)*(F+G)$

Hier musst du wissen, dass Punkt- vor Strichrechnung erfolgt. Daher wäre die Expression ohne dieses Wissens: $((A+B)*C) - ((D-E)*(F+G))$

Postfix:

$A \ B \ + \ C \ * \ D \ E \ + \ F \ G \ + \ * \ -$

1. Arbeite dieses Beispiel mittels Stack ab!

Zurück zu der VM. Die Instruktionen (Op Codes) benötigen jeweils nur ein Byte.

Siehe: <https://pengowray.github.io/wasm-ops/>

Zum Beispiel ist ein `if 0x04`, ein `i.32 load 0x28` und ein `i.32 mul 0x6C`.

2. Ergänze nun das `.wat` Programm von oben, um die Zahl zu verdoppeln.

3. Erstelle ein `.wat` Programm um eine Zahl um 1 zu verringern.

Nun aber genug vom low-level coden. WebAssembly ist als Ziel für ein Programm in einer höheren Programmiersprache gedacht wie.

² Siehe: <https://stanford-cs242.github.io/f18/lectures/04-1-webassembly-practice.html>

³ Idee für ein ITP Projekt 5.Klasse. Baue deinen eigenen WebAssembly Compiler für deine eigene Sprache!

⁴ Siehe SYT Unterricht

Interessanterweise kannst du auch JavaScript nach WebAssembly konvertieren. Nun, nicht genau JavaScript, sondern eine TypeScript Variante. Warum TypeScript? Erwinnere dich, dass du Datentypen benötigst. Die Sprache heißt daher Assembly Script.



Damit werden wir uns im nächsten Arbeitsblatt beschäftigen!

Dieses Arbeitsblatt war kurz. Jedoch:

