WEBASSEMBLY

Predmet: Programski prevodioci

Student: Rajko Zagorac sw23/2019

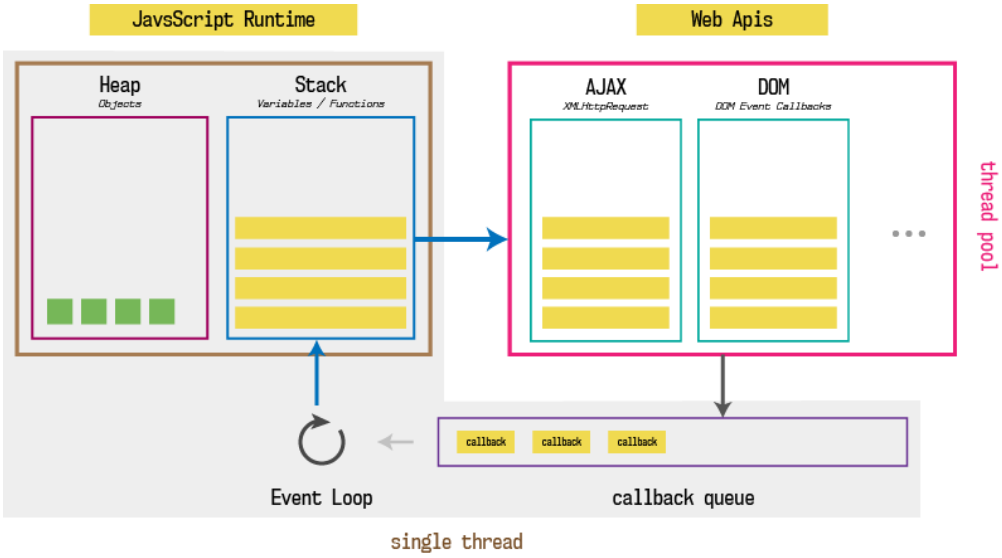
Sadržaj

Javascript i V8

Da bi shvatili kako funkcioniše webassembly i kako je došlo do njegove ideje, prvo se moramo upoznati sa samim načinom funkcionisanja i izvršavanja javascript koda. Fokus ovog rada će biti na V8 enginu, guglovoj implementaciji javascript engina. Webassembly, u nastavku wsm, predstavlja dodatak, i za kasnije smisleno poređenje javascripta i wsma, moramo se upoznati sa osnovama.

Event loop

Da bi se javascript kod izvršavao, potrebno je neko orkuženje. Danas je to uglavnom web browser ili node.js server, ovde ću pričati u kontekstu izvršavanja koda u browseru. Pre samog opisa rada V8 engina, ukratko ću opisati pojmove poput call stacka, event loopa i callback queua, koji predstavljaju važan deo razumevanja izvršavanja javascript koda.

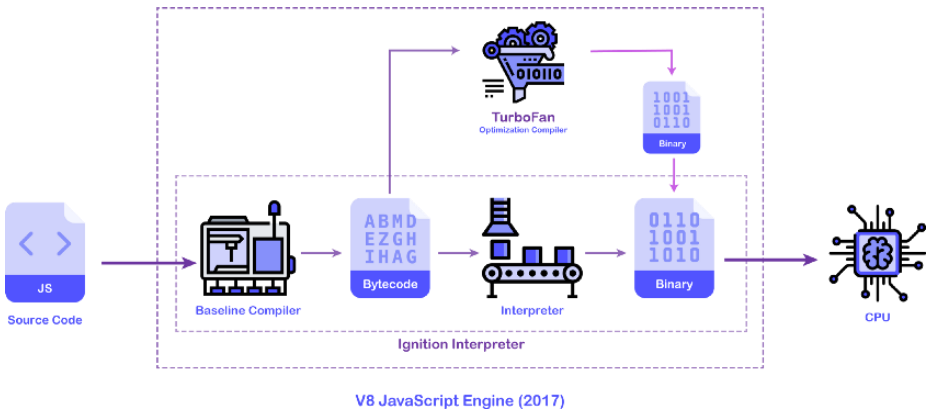


Budući da je javascript single threaded, potrebno je obebediti način izbegavanja blokirajućih funkcija poput zahteva preko mreže i pristupa bazi podataka. Sam browser rešava ovaj problem tako što nudi svoje dodatne niti na raspolaganje (Web Apis). Ovim možemo nastaviti izvršavanje koda, bez blokiranja, tako što ćemo dodati callback funkciju koja će se izvršiti odmah posle blokirajuće metode. Primarne funkcije koje se trenutno izvršavaju se nalaze na call stacku, dok su sve blokirajuće metode koje su završene na redu čekanja. Kada call stack postane prazan, event loop uzima prvu slobodnu funkciju iz reda i stavlja je na stack kako bi se izvršila. Sve ovo je lepo objašnjeno ovde gde je moguće pokrenuti simulaciju i videti kojim redom se izvršavaju funkcije kao i šta se tačno dešava prilikom blokirajućih metoda. Ovo je od daljeg značaja da bi shvatili u kom okruženju se V8 pokreće.

V8

Osnovni proces prevođenja i izvršavanja javascript koda bi bio sledeći:

- Leksička analiza
- Sintaksna analiza
- Kreirane apstraktnog sintaksnog stabla (AST)
- Interpreter Ignition prevodi AST u međukod (IR)
- Just in time compiler (JIT) prevodi IR u mašinski kod, tako što ga kompajlira. U ovoj vazi je aktiviran TurboFan, koji koristi prikupljene informacije za vreme prevođenja kako bi optimizovao kod



Leksička analiza

V8 koristi svoju implemetaciju koja se može naći ovde. Lista podržanih tokena se može naći ovde.

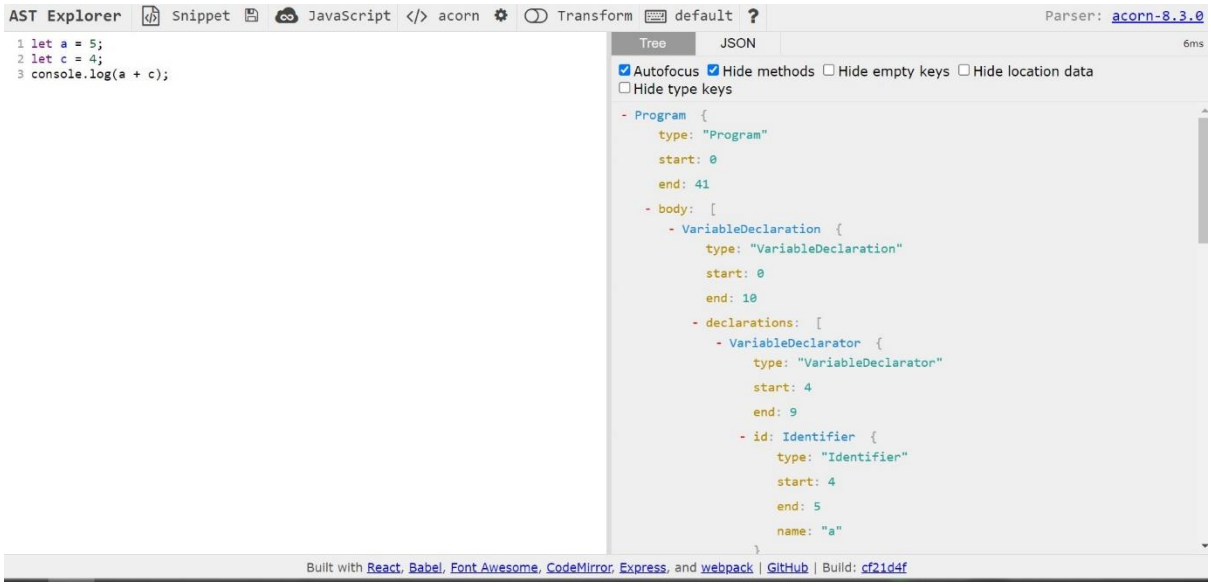
Sintaksna analiza

V8 implementacija se vidi ovde

AST

V8 kod za ast se vidi ovde.

Ovde sam naišao na zgodan alat koji prikazuje kako bi izgledalo generisano AST stablo.



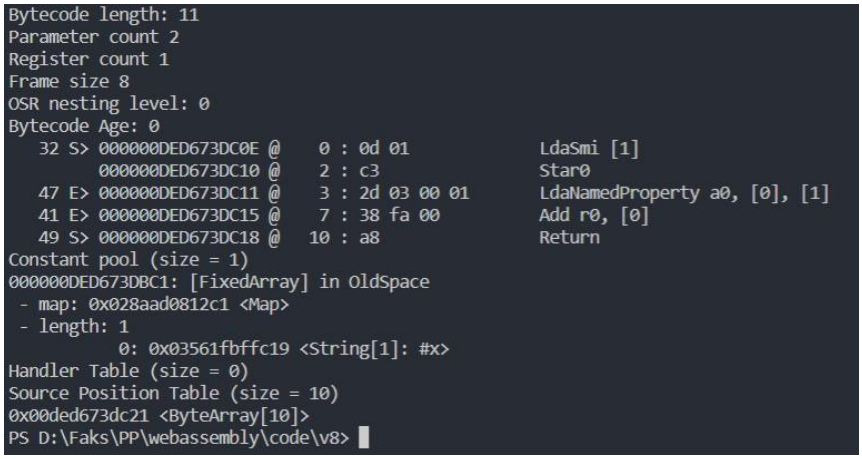
Prevođenje u IR

Kao što je već rečeno, u ovoj fazi se AST prevodi u IR. Zgodna stvar kod node.js je ta što lako možemo da vidimo koji bytecode je generisan korišćenjem komande –print-bytecode. U folderu **v8/bytecodeExample.js** se nalazi jednostavan primer koji ću da testiram. Kako je fokus ovog rada na wsmu, neću previše vremena posvetiti analizi bytecoda, ali ću ostaviti neke korisne linkove na koje sam naišao gde se može shvatiti više.

```
function incrementX(obj) {
    return 1 + obj.x;
  }
incrementX({x: 42});
```

Generisaću bytecode za funkciju incrementX. Potrebno je uneti sledeću komandu: **node –print-bytecode –print-bytecode-filter=incrementX**

Ispod sledi prikaz generisanog bytecoda.



Glavna stvar kod bytecoda je ta što se uvodi dodatna apstrakcija, i što generisanje koda ne mora da zavisi od konkretne arhitekture. Ovaj pristup je danas dominantan u veći popularnih programskih jezika. Uglavnom se pravi takav model koji će simulirati fizički CPU. Zbog toga se interpreteri prave ili kao stek mašine ili registarske mašine. Konkretno, Ignition interpreter koji V8 koristi jeste registarska mašina sa akumulator registrom. Sledi kratka analiza pojedinačnih instrukcija.

LdaSmi[1] – U suštini kaže da se stavi intiger 1 u akumulator. Load small intiger in acumulator od čega sledi naziv.

Star0 – Stavi ono što je u akumulatoru u registar r0.

LdaNamedProperty a0, [0], [1] – Stavi a0 u akumulator, gde je ai redni broj parametra funkcije. [0] nula označava redni broj propertija objekta, u našem slučaju, to će referencirati obj.x. Postoji posebna tabela za brzu pretragu propertija. Tabela se može videti u gornjem primeru u odeljku [FixedArray]. Vidimo polje **0: 0x03561fbffc19 <String[1]: #x>** koje refernica property x. Operand [1] na kraju instrukcije je indeks **feedback vectora** o kojem ću reći nešto više u samom nastavku.

Add r0, [0] – Stavi r0 u akumulator. Budući da smo u prošlom koraku imali 1 u r0 registru, a 42 u akumulatoru, konačna vrednost akumulatora će biti 43. [0] takođe predstavlja deo koji se tiče feedback vektora.

Return – povratna vrednost iz akumulatora, vraća 43 kao povratnu vrednost funkcije.

Spisak V8 bytecodova se može naći ovde. Takođe Chris Hay ima zanimljivu playlistu na yt gde detaljnije objašnjava delove bytecoda idući od jednostavnijih ka složenijim primerima.

Izvršavanje i optimizacija

Ovde ću ukratno opisati na koji to način kod može da se optimizuje i šta je to što suštinski može da uspori izvšavanje javascript koda. U fajlu se **v8/optimization.js** se nalazi mali primeri koda čisto ilustracije radi.

Kako bismo predstavili objekte u javascriptu u memoriji? Mogli bismo napraviti rečnik za svaki objekat i referencirati objekat da gađa taj rečnik iz memorije. Ovo može biti dosta neefikasno prvenstveno zato što oblici objekata mogu da se preklapaju.

```
// shape a
let a= {x : 34}

//shape b
let b1 = { x: 10, y: 10}
let b2 = { x: 5, y: 30}
// ovde nema potrebe da se pravi

// shape c
b2.z = 1
```

Ovde nema potrebe da se pravi novo mesto u memoriji za oblik objekta b2, možemo iskoristiti oblik već kreiranog b1 objekta. Problem nastaje kod dodavanja propertija naknadno kao što je to slučaj sa poslednjom linijom gde menjamo b oblik dodajući z property. Ovde će se napraviti novi oblik, recimo c, koji će imati referencu ka obliku b na koji dodaje dodatan property. Kada se sledeći put bude tražio odgovarajući property, moraćemo se kretati kroz ulančanu listu oblika, što dosta smanjuje vreme pristupa pogotovo u kompleksnijim aplikacijama sa dosta objekata različitih oblika i međusobnih referenci. Zbog ovoga V8 koristi Inline Cahce(IC). IC služi kako bi memorisao informacije o propertijima da bi ih lakše našao sledeći put. Struktura koja se za to koristi jeste feedback vector o kojem je bilo reči nešto ranije prilikom analize bytecoda.

```
function load(a) {
    return a.key
}

let first = {key:'first'}
let iAmFastBeacuseIhaveSameStructureAsFirst = {key: 'second'}
let iAmSlowBecauseIHaveNewStructureWithFooKey = { foo: 'slow'}

load(first)
load(iAmFastBeacuseIhaveSameStructureAsFirst)
load(iAmSlowBecauseIHaveNewStructureWithFooKey)
```

Posmatrajmo primer iznad. Posle izvršavanja prve load funkcije, feedback vector može da izgleda ovako **[{ slot: 0, icType: LOAD, value: MONO(A)}]**. **A** predstavlja oblik objekta koji sadrži jedan property key. Nakon izvršavanja druge load metode, lookup će biti znatno brži jer je već pretpostavljeno da funkcija radi samo sa objektima oblika A. Ovo je jedan primer kako TurboFan može da **optimizuje** kod. Problem može da nastane kada dođe do izvršenja poslednje metode. U tom slučaju će feedback vector struktura biti izmenjena i izgledaće ovako **[{ slot: 0, icType: LOAD, value: POLY[A, B]}]**. Sada svaki put kada treba da se pozove funkcija, moraćemo da prolazimo kroz sve oblike a ne samo jedan oblik kao u prvom slučaju. Ovo je proces **deoptimizacije**, inicijalno smo pretpostavili da će funkcija primati samo objekat oblika **A** tako da je TurboFan mogao da optimizuje kod, ali posle smo joj poslali drugačiji oblik, te prvobitna optimizacija više nije validna. Ovo ukazuje na to da ukoliko želimo brži kod ne bismo trebali previše menjati oblike i strukture. Smenjivanje između optimizacije i deoptimizacije predstavlja važnu stvar za analizu performansi i dalje razumevanje wsm-a.

Zanimljiv video na temu optimizacije i deoptimizacije možete pogledati ovde.

WEBASSEMBLY

Uvod

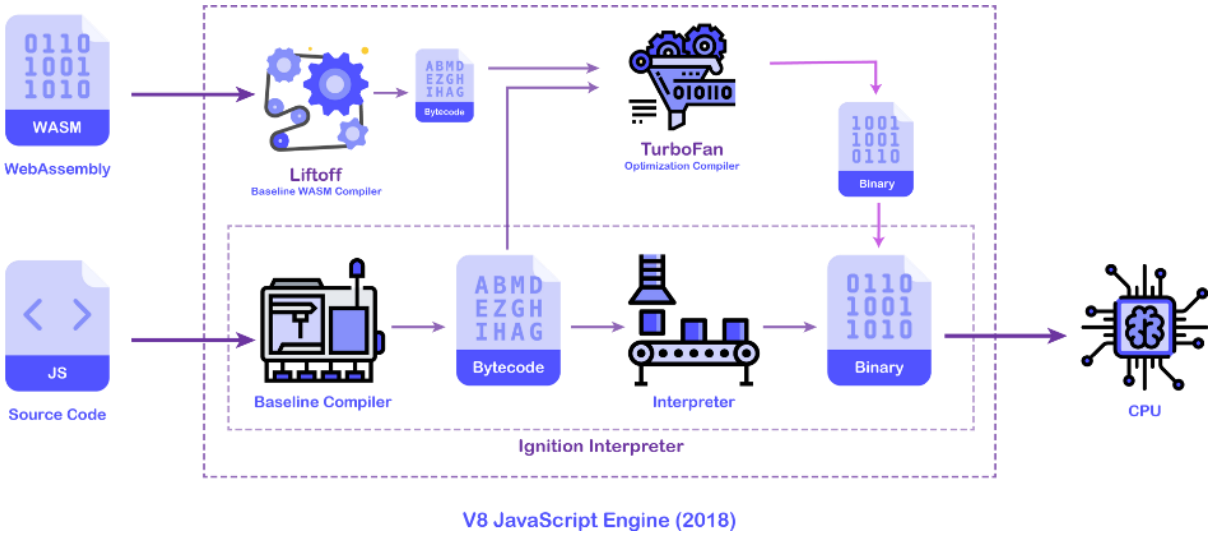
Sada kada imamo osnovu funkcionisanja javascripta i V8 engina, možemo da vidimo kako se wsm uklapa u celu priču.

WebAssembly je inspiran asemblerom i glavni zadatak jeste da omogući brzo izvršavanje koda na webu, i da reši neke probleme koje ima javascript, a koje smo videli u prethodnom poglavlju (deoptimizacija i usporavanje koda npr). Pored toga još jedan od glavnih ciljeva jeste da developerima omogući programiranje u drugim programskim jezicima poput C++ ili Rusta davanjem odgovarajućih alata koji će tako napisan kod prevesti u webassebmly koji potop može da se izvršava na webu.

WebAssembly je inspirisan **asm.js**-om, projektom koji je tim Mozile prezentovao još u 2013. Glavna ideja je bila da se dodatno proširi sintaksa javascripta na pojedinim mestima kako bi se poboljšale performanse. Specifikacija se može videti ovde. Iako je na startu imao početne uspehe, zbog nedostatka standarda i alata koji bi pomogli u radu, projekat je napušten.

Kao što se može videti iz specifikacije wsm-a, glavni ciljevi prilikom dizajna su bili da jezik bude brz, siguran, dobro definisan, nezavisan od hardvera, nezavisan od jezika, nezavisan od platforme i otvoren. Odličan video o kratkom nastanku wsm-a. Wsm je danas podržan u svim modernim browserima. Implementacija je doduše ostavljena na samim enginima. Kao što sam ranije rekao, fokus rada je na V8 enginu, tako da ću dalje pričati u tom kontekstu.

Da bi wsm mogao da se izvršava u V8 enginu, bilo to browser ili node js, potrebno ga je nekako prevesti u bytecode, sličan onom koji generiše Ignition interpreter. Za ovako nešto je potreban dodan alat, kompajler, kakav je Liftoff. Bytecode koji proizvede liftoff se prosleđuje TurboFanu da ga optimizuje i proizvede mašinski kod. Ovog puta, proizvede mašinski kod ima skoro nativnu brzinu. Na slici ispod je prikazan tok izvršavanja.



Wasm fajlovi se završavaju uglavnom ekstenzijom .wasm i sadrže binarne instrukcije propisane standardom. Takođe, pored binarne reprezentacije, postoji i čitljivija tekstualna reprezentacija u obliku .wat fajlova. Wat specifikacija se može pronaći ovde. Iako je wsm dobio veću popularnost korišćenjem bogatih alata, tako da programeri ne moraju direktno da pišu wat fajlove, odlučio sam

da ipak krenem drugačijim pristupom i da napišem po koji jednostavan wat fajl direktno, čisto da steknem osećaj kako stvari funkcionišu. Da ne bih odjednom opisivao glavne koncepte wsm-a, polako ću ih uvodi kroz primere.

Wat2wasm i računanje kvadrata broja

Budući da nam tekstualna reprezentacija binarnih instrukcija kroz wat fajl nije od velikog značaja prilikom izvršavanja, potreban nam je alat koji će odraditi konverziju wat fajla u wasm fajl. Koristio sam **wat2wasm** iz **wabt** tolkita koji je dostupan ovde ovde. Kako koristim windows, upotreba je sledeća:

- git clone --recursive https://github.com/WebAssembly/wabt u trenutnom folderu gde se I nalazi dokumentacija
- cd wabt
- git submodule update –init
- Za cmake i build sam koristio visual studio 2019 koji ima ugrađenu podršku. Otvoriti wabt folder pomoću vs-a.
- Kreirati build folder sa mkdir build
- cd build
- cmake .. -DCMAKE_BUILD_TYPE=DEBUG -DCMAKE_INSTALL_PREFIX=..\ -G "Visual Studio 16 2019"
- cmake --build . --config DEBUG --target install

Sada kada imamo spreman alat, možemo krenuti pisati wat fajlove. U folderu **wasm** će se nalaziti svi primeri u daljem nastavku. Prvi primer je jednostavan i predstavlja računanje kvadrata nekog broja.

square.wat

```
(module
    (func $square (param i32) (result i32)
        local.get 0
        local.get 0
        i32.mul
    )
    (export "square" (func $square))
)
```

Kao što vidimo wat fajlove definišemo kroz module. Napisali smo jednu funkciju koja prima 32-bitni broj i takođe vraća rezultat u 32 bita. **local.get 0** govori da stavimo prvi parametar funkcije na stek. Webassembly funkcioniše po principu stek mašine, o čemu ću više pričati u nastavku. Operacija **i32.mul** zahteva dva parametra na steku, zbog toga smo duplirali liniju na kojoj stavljamo prvi parametar funkcije na stek. Kao povratna vrednost će se uzeti upravo izračunat kvadrat broja, kao proizvod poslednja dva elementa steka. Na kraju u delu za export, definišemo da želimo da eksportujemo funkciju da bismo je mogli koristiti direktno kao poziv javascript funkcije. Pošto javascript ne razume wat fajlove, već samo wasm, koristiću wat2wasm.

PS: D:\Faks\PP\webassembly\code\wasm> **../../wabt/bin/wat2wasm square.wat -o square.wasm**

Sadržaj generisanog koda se može videti pomoću **xxd** komande. Pošto komanda nije po defaultu podržana u windowsu, može se skinuti ekvivalent unixove ovde. Pokretanjem komande xxd .\square.wasm dobijamo sledeći ispis:

```
PS D:\Faks\PP\webassembly\code\wasm> xxd .\square.wasm
00000000: 0061 736d 0100 0000 0106 0160 017f 017f  .asm.......`....
00000010: 0302 0100 070a 0106 7371 7561 7265 0000  ........square..
00000020: 0a09 0107 0020 0020 006c 0b              ..... . .l.
```

Postoji i duža verzija, sa dodatnim objašnjenima sa strane. Potrebno je pokrenuti prethodnu komandu sa -v flagom.

```
PS D:\Faks\PP\webassembly\code\wasm> ../../wabt/bin/wat2wasm square.wat -v
0000000: 0061 736d                                 ; WASM_BINARY_MAGIC
0000004: 0100 0000                                 ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                                        ; section code
0000009: 00                                        ; section size (guess)
000000a: 01                                        ; num types
; func type 0
000000b: 60                                        ; func
000000c: 01                                        ; num params
000000d: 7f                                        ; i32
000000e: 01                                        ; num results
000000f: 7f                                        ; i32
0000009: 06                                        ; FIXUP section size
; section "Function" (3)
0000010: 03                                        ; section code
0000011: 00                                        ; section size (guess)
0000012: 01                                        ; num functions
0000013: 00                                        ; function 0 signature index
0000011: 02                                        ; FIXUP section size
; section "Export" (7)
0000014: 07                                        ; section code
0000015: 00                                        ; section size (guess)
0000016: 01                                        ; num exports
0000017: 06                                        ; string length
0000018: 7371 7561 7265                           square  ; export name
000001e: 00                                        ; export kind
000001f: 00                                        ; export func index
0000015: 0a                                        ; FIXUP section size
; section "Code" (10)
0000020: 0a                                        ; section code
0000021: 00                                        ; section size (guess)
0000022: 01                                        ; num functions
; function body 0
0000023: 00                                        ; func body size (guess)
0000024: 00                                        ; local decl count
0000025: 20                                        ; local.get
0000026: 00                                        ; local index
0000027: 20                                        ; local.get
```

Pokretanje

Wasm fajl možemo da pokrećemo i u browseru i u node js okruženju, budući da oba interno sadrže V8 engine. Prvo ću prikazati pokretanje u node.js a potom i u browseru.

Node

Napraviti novo okruženje sa **node init**. U package json staviti index.js kao entry point, i naznačiti da se radi o modulu, dodavanjem **“type”:”module”** u **package.json**. Sadržaj index.js fajla je dat ispod.

```
import fs from 'fs'

const squarewasm = fs.readFileSync('./square.wasm')

const square = await WebAssembly.instantiate(new Uint8Array(squarewasm))
        .then(res => res.instance.exports)

console.log(square.square(10));
```

Naznačimo da ćemo da koristimo fajl sistem, konvertujemo wasm u odgovarajuću bajt reprezentaciju, i na kraju instanciramo wasm modul. Javascript engine nam nudi **WebAssembly interface**. Ovo predstavlja elegantan način za komunikaciju između js-a i wsm-a. Specifikacije se može videte ovde, kao i na mozilinoj dokumentaciji ovde. Kada instanciramo modul, možemo da ga koristimo za pozivanje svih funkcija kao da koristimo običan javascript. Ovo predstavlja veliku prednost wsm-a u odnosu na njegovog pretka **asm.js**. Ne moramo da menjamo sintaksu javascripta, dovoljno je samo da instanciramo wasm modul, i da ga posle toga direktno pozivamo.

Nakon pokretanja vidimo očekivani ispis.

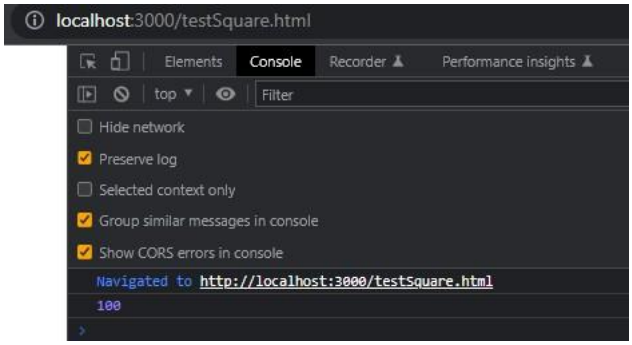
```
PS D:\Faks\PP\webassembly\code\wasm> node index.js
100
```

Browser

Za bismo mogli da serviramo wasm fajlove, koristićemo express server. Instalirati express sa **npm install –save express**. U fajlu **server.js** se nalazi jednostavna konfiguracija. Svi wasm i ostali statički fajlovi će biti locirani u **public** folderu. U **testSquare.html** importujemo wasm module u script tagu

```
<script>
    WebAssembly.instantiateStreaming(fetch('square.wasm'))
    .then(obj => {
        var result = obj.instance.exports.square(10);
        console.log(result);
    })
</script>
```

Primer je sličan kao I kod pokretanja kroz node, samo što je ovde potrebno fetchovati fajl. Kada pokrenemo server sa **node server.js** i pogodimo url u browseru, dobićemo validan rezultat.



O samom jeziku

Nakon jednog jednostavnog praktičnog primera, u ovom delu ću malo više pričati o samoj strukturi wsm-a, i nekim osnovnim teorijskim konceptima. Rad koji predstavlja polaznu tačku o teoriji iza wsm-a je [Bringing the Web up to Speed with WebAssembly](). Ovde se priča o dizajnu i odlukama koje su donošene. Wsm je dizajniran kao stek mašina koju sam spomenuo i ranije prilikom opisa nekih od instrukcija. Odabir ovog dizajna u poređenju sa registarskom mašinom se može obrazložiti zauzećem memorije koje je manje u slučaju stek mašine. Na slici ispod je dat prikaz glavne specifikacije sintakse jezika, slika je preuzeta iz spomenutog rada.

(value types) $t ::= \text{i32} \mid \text{i64} \mid \text{f32} \mid \text{f64}$
(packed types) $tp ::= \text{i8} \mid \text{i16} \mid \text{i32}$
(function types) $tf ::= t^* \to t^*$
(global types) $tg ::= \text{mut}^? \; t$

$unop_{iN} ::= \textbf{clz} \mid \textbf{ctz} \mid \textbf{popcnt}$
$unop_{fN} ::= \textbf{neg} \mid \textbf{abs} \mid \textbf{ceil} \mid \textbf{floor} \mid \textbf{trunc} \mid \textbf{nearest} \mid \textbf{sqrt}$
$binop_{iN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_}sx \mid \textbf{rem\_}sx \mid$
$\textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \textbf{shl} \mid \textbf{shr\_}sx \mid \textbf{rotl} \mid \textbf{rotr}$
$binop_{fN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{min} \mid \textbf{max} \mid \textbf{copysign}$
$testop_{iN} ::= \textbf{eqz}$
$relop_{iN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt\_}sx \mid \textbf{gt\_}sx \mid \textbf{le\_}sx \mid \textbf{ge\_}sx$
$relop_{fN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \textbf{le} \mid \textbf{ge}$
$cvtop ::= \textbf{convert} \mid \textbf{reinterpret}$
$sx ::= \textbf{s} \mid \textbf{u}$

(instructions) $e ::= \textbf{unreachable} \mid \textbf{nop} \mid \textbf{drop} \mid \textbf{select} \mid$
$\textbf{block}\ tf\ e^*\ \textbf{end} \mid \textbf{loop}\ tf\ e^*\ \textbf{end} \mid \textbf{if}\ tf\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid$
$\textbf{br}\ i \mid \textbf{br\_if}\ i \mid \textbf{br\_table}\ i^+ \mid \textbf{return} \mid \textbf{call}\ i \mid \textbf{call\_indirect}\ tf \mid$
$\textbf{get\_local}\ i \mid \textbf{set\_local}\ i \mid \textbf{tee\_local}\ i \mid \textbf{get\_global}\ i \mid$
$\textbf{set\_global}\ i \mid t.\textbf{load}\ (tp\_sx)^?\ a\ o \mid t.\textbf{store}\ tp^?\ a\ o \mid$
$\textbf{current\_memory} \mid \textbf{grow\_memory} \mid t.\textbf{const}\ c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*\ \textbf{func}\ tf\ \textbf{local}\ t^*\ e^* \mid ex^*\ \textbf{func}\ tf\ im$
(globals) $glob ::= ex^*\ \textbf{global}\ tg\ e^* \mid ex^*\ \textbf{global}\ tg\ im$
(tables) $tab ::= ex^*\ \textbf{table}\ n\ i^* \mid ex^*\ \textbf{table}\ n\ im$
(memories) $mem ::= ex^*\ \textbf{memory}\ n \mid ex^*\ \textbf{memory}\ n\ im$
(imports) $im ::= \textbf{import}\ \text{“}name\text{”}\ \text{“}name\text{”}$
(exports) $ex ::= \textbf{export}\ \text{“}name\text{”}$
(modules) $m ::= \textbf{module}\ f^*\ glob^*\ tab^?\ mem^?$

Figure 1. WebAssembly abstract syntax

Kao što se može primetiti, wsm podržava 4 osnovna tipa podataka. 32 bitni I 64 bitni intiger I float brojevi. Što se tiče wat sintakse, ona je predstavljena u obliku S izraza. S izrazi mogu da predstave strukturu wsm modula u obliku stabla. Svaki izraz prvo sadrži tip čvora a potom posle razmaka sledi lista atributa ili ostalih čvorova. Npr za izraz **(module (memory 1) (func))** stablo će biti prezentovano sa korenskim čvorom 'module' i dva čvora deteta, 'memory' sa atributom 1 i 'func' detetom.

Fibonači primer

U ovom delu ću prikazati primer računanja fibonačijevih brojeva u wsm-u pa potom i u js-u. Na kraju ću porediti dobijene rezultate u smislu brzine izvršenja. Fibonačijevi brojevi su čest primer prilikom poređenja performasni, jer je kod jednostavan a razlike u brzini dovoljno velike da možemo doći do nekih zaključaka. Neću odmah krenuti sa korišćenjem alata koji transformišu kod napisan u nekom višem jeziku direktno u wsm, već ću i dalje kod pisati ručno u wat formatu prateći ovaj video .

Ručno pisana iterativna funkcija za traženje n-tog fibonačijevog broja

```
(module
    (func $fibonacci (param $index i32) (result i32) (local $fibn i32)
     (local $fibn1 i32) (local $fibn2 i32) (local $n i32)
        i32.const 0
        local.set $fibn ;; fibn = 0
        i32.const 1
        local.set $fibn1 ;; fibn1 = 1
        i32.const 0
        local.set $fibn2 ;; fibn2 = 0
        local.get $index
        local.set $n ;; n = 1. par. funkcije
        (block (loop ;; n > 0
            local.get $n
            i32.const 0
            i32.eq
            br_if 1 ;; provera uslova za n, 1 kaže da se ide blok iznad
            local.get $fibn1
            local.set $fibn2 ;; fibn2 = fibn1
            local.get $fibn
            local.set $fibn1 ;; fibn1 = fibn

            local.get $fibn1
            local.get $fibn2
            i32.add
            local.set $fibn ;; fibn = fibn1 + fibn2
            ;; n--
            local.get $n
            i32.const 1
            i32.sub
            local.set $n
            br 0   ;; br 0 znaci branch to the top of the stack, llop
        ))
        local.get $fibn
    )
    (export "fibonacci" (func $fibonacci))
)
```

U primeru iznad je ručno napisan wat fajl. Sve lokalne promenljive koje se koriste se moraju naznačiti na vrhu funkcije što vidimo po ključnoj reči local. Kao što se može videti, sve operacije se obavljaju preko steka. Na primer operacija smanjivanja brojaca (n - -), bi se mogla prevesti u nekoliko koraka:

- local.get $n -> postavi vrednost n na stek
- i32.const 1 -> stavi vrednost 1 na stek
- i32.sub -> popuj dve vrednosti sa steka, oduzmi ih i rezultat stavi nazad na stek
- local.set $n -> skini vrednost sa steka i ubaci u n

Ručno pisana rekurzivna funkcija za traženje n-tog fibonačijevog broja

```
(module
  (func $fibTemp (param $n i32) (param $fibn1 i32) (param $fibn2 i32) (result i32)
    (if (result i32)
        (i32.eqz (local.get $n))
        (then (local.get $fibn1))
        (else
            (call $fibTemp
              (i32.sub (local.get $n)(i32.const 1))
              (local.get $fibn2)
              (i32.add (local.get $fibn1) (local.get $fibn2))
            )
        )
    )
  )

  (func $fibonacci (param $index i32) (result i32)
    (call $fibTemp (local.get $index) (i32.const 0) (i32.const 1))
  )

  (export "fibonacci" (func $fibonacci))
)
```

Ovde je prikaz još jedan način implementacije korišćenjem rekurzije. Prilikom poziva funkcije koristimo ključnu reč call.

Fibonači c++ kod generisan u wat

Da ne bismo ručno pisali wat fajlove, postoje alati koji direktno prevode kod napisan u višem programskom jeziku direktno u wat ili wasm fajlove. U slučaju C/C++ koristi se **Emscripten**. Ovaj alat interno koristi LLVM i Binaryen. Takođe postoje i online alati kao što je WebAssemblyExplorer. Konkretno ovde sam koristio online alat, budući da sam imao problema prilikom instalacije emscriptena na windowsu. Fajl **fibonacciInC++.cpp** sadrži c++ implementaciju traženja n-tog fibonačijevog broja, dok se u **fibonacciFromC++.wat** nalazi generisan wat fajl alata. Generisan kod je morao biti malo promenjen da bi radio ispravno.

Poređenje rezultata

U ovom delu ću porediti rezultate prethodnih implementacija fibonačijih brojeva. Ukoliko je express server pokrenut, otići na localhost:3000. Trebalo bi da se prikaže ovakva slika.

Testing Fibonacci(42)

- Testing regular javascript implementation
- Testing wsm hand written iterative implementation
- Testing wsm hand written recursive implementation
- Testing wsm generated from C++

Za primer je korišćeno računanje 42. fib broja. Slede rezultati

----------Regularni javascript----------

Result: 267914296

Time taken: 4210 ms

---------Iterativna implementacija wsm----------

Result: 267914296

Time taken: 0.09999990463256836 ms

--------Rekurzivna implementacija----------

Result: 267914296

Time taken: 0.10000002384185791 ms

--------C++ generisan pomoću alata----------

Result: 267914296

Time taken: 3512.7000000476837 ms

Kao što se može primetiti, wsm je pokazao bolje rezultate u poređenju sa običnim javascriptom. Između rekurzivne i iterativne implementacije nema nikakve razlike. Možemo primetiti da je online alat za generisanje koda dao dosta lošije rezultate u poređenju sa ručno pisanim wat fajlovima. Ovo i jeste mana, ali zbog toga je alat i namenjen više za istraživanje nego za pravo produkciono okruženje. Za bolji kod bi svakako trebalo koristiti alate kakav je Emscripten.

Nešto i o drugim alatima

Budući da je wsm postaje sve popularniji na vebu, razvijaju se i alata koji će olakšati posao programerima. Pored pomenutog Emscriptena, koji ima podršku za C i C++ programe, rust se pokazao kao jezik koji trenutno ima i najveću podršku wasm-a kroz alate kakav je **Yew.** Razlog zbog kog se rust pokazao kao dobar izbor se može naći u činjenici da je dobar deo wasm runtima implementiran baš u rastu.Takođe, još jedan od alata za .NET platformu predstavlja **Blazor WebAssembly.** U slučaju pythona, često se koristi Pyodide.

Današnja primena wsm-a

Zanimljiv, iako malo stariji, video, koji govori o nekim od primena wsm-a. Danas se webassembly koristi na mestima u aplikacijama koje zahtevaju veliki broj računanja i visoko zauzeće CPU-a. Dobri kandidati mogu biti igrice, naučne simulacije visokih performansi, strimovanje muzike, obrada i keširanje slika, aplikacije za enkripciju osetljivih podataka.

Mane wsm-a

Trenutno nema podršku za više niti. Ovo i ne mora biti toliki problem jer se wsm može pokrenuti iz Web Workera. Nema podršku za rukavanje izuzecima što može otežati debagovanje. Nema svoj garbage kolektor, jer ima poseban heap koji ne prati javascript. Nema mogućnost direktnog rukovanja sa DOM stablom. Ovo je glavni razlog zašto se danas i dalje većiniski koristi samo javascript. Većina koda na webu ne zahteva preterano kompleksne operacije, te nema potrebe za wasm-om.

Sumiranje i zaključak

Wsm nije tu da zameni javascript, već da ga dopuni. Što se tiče perfomansi, web assembly uglavnom jeste brži, ali moramo se setiti da se na kraju sve izvršava kroz engine. Ovo je važno shvatiti, jer sam javascript sa TurboFanom može biti identične brzine kao i wsm. Problem kada dobijamo usporenje je

deoptimizacija, što je bilo reči i ranije. Wsm može da smanji te deoptimizacije, jer na samom startu naglašava koji su tipovi promenljivi, i time dobija na brzini.

Iako je tu već 5 godina, wsm nije još u potpunosti ispunio svoj potencijal. Verujem da će pojavom novih slučajeva korišćenja i proširenjem dostupnih alata, rasti i potreba za poznavanjem i upotrebom wasm-a.