Budućnost weba: Web Assembly

Bajivić, Antonio

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:816919

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-21**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij računarstva

BUDUĆNOST WEBA: WEBASSEMBLY

Diplomski rad

Antonio Bajivić

Osijek, 2021.

SADRŽAJ

1. UVOD 1

1.1. Zadatak diplomskog rada 2

2. UVOD U WEB I WEBASSEMBLY 3

2.1. Povijest Weba 3

2.2. Struktura Web-a 5

2.2.1. Prikaz stranica i korisničko iskustvo (engl. User Experience) 5

2.2.2. Struktura koda 6

2.3. WebAssembly 8

2.3.1. Uvod 8

2.3.2. Povijest 11

2.3.3. Upotrebljivost 12

3. PRIMJENA WEBASSEMBLY-A 13

3.1. Značajke 13

3.1.1. Buduće značajke 15

3.2. Sigurnost 17

3.3. Performansa 19

3.3.1. WebAssembly protiv JavaScripta - teorijski 19

3.3.2. WebAssembly protiv JavaScripta – eksperiment 23

3.4. Ugrađivanje u web tehnologije 27

3.4.1. Frontend 28

3.5. Ugrađivanje u ne-web tehnologije 29

3.5.1. Video igre 30

3.5.2. Obrada slike 31

3.5.3. Podatkovne znanosti 31

3.5.4. Aplikacije bez servera 32

4. ZAKLJUČAK 35

LITERATURA 36

SAŽETAK 39

ABSTRACT 40

ŽIVOTOPIS **41**

PRILOZI **42**

1. UVOD

Svakidašnjim korištenjem web stranica i aplikacije, a i same tehnologije općenito tako došlo je do potrebe za preoblikovanjem same „jezgre“ *weba* te je potrebno ukloniti određena ograničenja. Trenutno se za stvaranje dinamičnih stranica, odnosno za dodavanje funkcionalnosti na elemente neke stranice ili aplikacije koriste pomalo zastarjeli skriptni jezici *PHP* ili vrlo popularan *JavaScript*. Oba ova jezika su se toliko razvila da su dobili svoje softverske okvire (*engl.* framework) koji ubrzavaju proces stvaranja stranice. No, kako je počela rasti potreba za stvaranjem web aplikacija, a i svakim danom raste sve veći broj korisnika Interneta, potrebno je bilo pojačati infrastrukturu mreže sa boljom arhitekturom servera, a i uvođenjem novih generacija mreže. Nakon pojačavanja infrastrukture problem na stranicama ili aplikacijama nastaje u ograničenjima PHP-a i JavaScripta. Oni kao jezici ostvaruju interakciju s elementima koja je vrlo jednostavna za korištenje, ali problem nastaje kada se želi postići veća brzina koju je lako ostvariti korištenjem programskih jezika niske razine (*engl.* low-level).

Programski jezici niske razine koji su prilikom obrade koda najbliži računalnim komponentama, posebno mikroprocesoru. Takvi su jezici direktno implementirani u hardver računala. Programski jezici niske razine se dodatno dijele na asemblerske jezike i strojne jezike. Oni su ovisniji više o hardveru jer su namijenjeni za manevriranjem komponentama. Asemblerski jezik znači da se koriste vrlo jednostavne naredbe kao što su MOV (pomakni), ADD (dodaj), SUB (oduzmi). Takve naredbe su zadužene za izvođenje jednostavnih operacija, npr. pomicanje vrijednosti u memorijske registre i izvode se određene kalkulacije. Strojni jezik je najniža razina programskih jezika zato što sadrži binarni kôd koji je generiran obradom visokih programskih jezika, tipa Python. Programeri koji su zaduženi za izradu softverskih prevoditelja (*engl.* compiler) i operacijskih sustava moraju biti upoznati s ovom kategorijom jezika.

Obzirom na prednost programskih jezika niske razine zbog „blizine“ računalnim komponentama došlo je do potrebe implementiranja sličnog načina razvoja aplikacija i u svijetu weba općenito. O ideji izradi standarda za web koji će, u teoriji, koristiti bilo koji programski jezik te pomoću asemblera obraditi naredbe i približiti ih strojnom jeziku pričalo se još u 2010., a prvo pojavljivanje takvoga je 2015. pod nazivom „WebAssembly“.

U drugom poglavlju opisan je generalno Web, kako i otkud te zašto je nastao te strukturu web stranica i koda. Dalje u nastavku drugog poglavlja dat će se kratak opis WebAssembly-a, a i njegovu povijest te što on zapravo je i što doprinosi današnjem svijetu weba te neke od njegovih prednosti. U trećem poglavlju demonstrirane su performanse jednostavne aplikacije za računanje

Fibonaccijevog niza pojedinačnim korištenjem JavaScripta i WebAssembly tekstualnog formata. Osim eksperimenta dani su konkretni primjeri koji su uz pomoć WASM-a sada na webu i lako dostupni svim ljudima. Četvrtim poglavljem se zaključuje ovaj rad te je objašnjeno na koji je način ovo budućnost weba i kako će web kakav se danas poznaje biti izmijenjen i poboljšan. Nakon detaljnijeg upoznavanja ovog standarda puno je jednostavnije za shvatiti zašto postoji potreba za njegovim korištenjem.

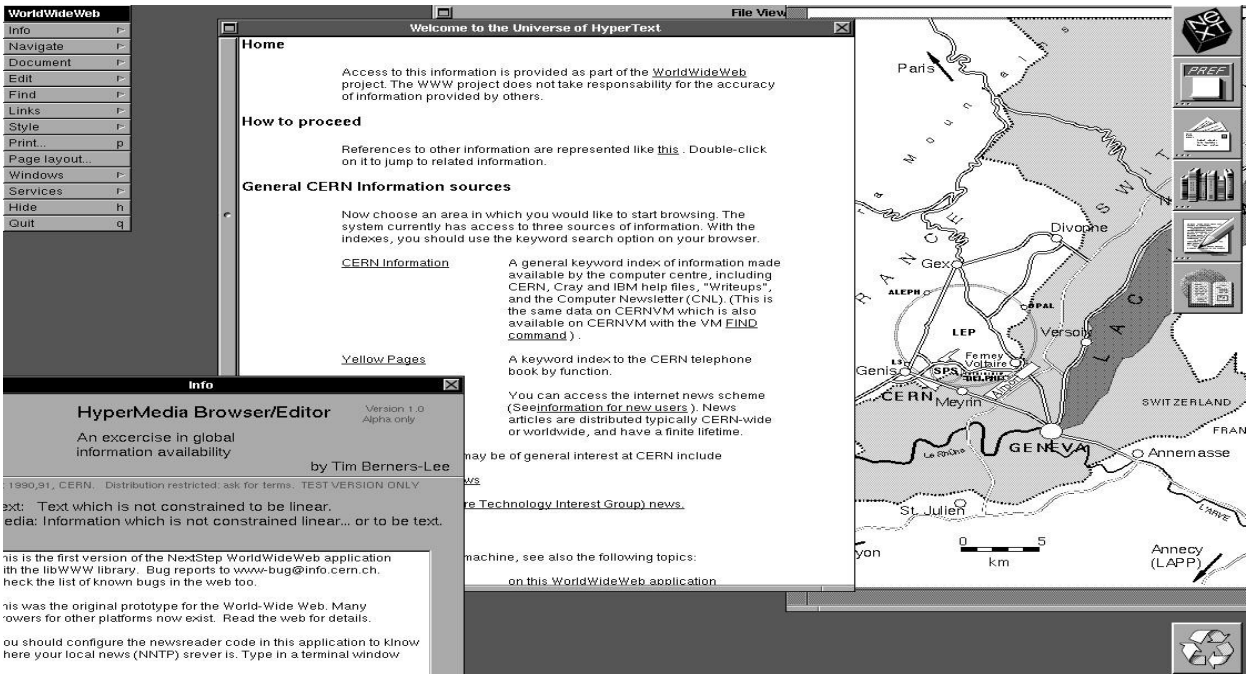
1.1. Zadatak diplomskog rada

Zadatak ovog rada je opisati na koji način će ovaj standard, odnosno ovaj pristup izrade web aplikacija i/ili stranica poboljšati budućnost weba i omogućiti raznim programerima jednostavniji „ulaz“ u web svijet. Ponekad se određeni razvojni programer (*engl.* developer) van „web svijeta“ nemaju vremena posvetiti savladavanju potpuno novog jezika, a korištenjem prethodnog znanja u drugim jezicima to će im bit olakšano korištenjem WebAssembly-a. Osim što će se objasniti prednosti ovog standarda, objasnit će se pristup kroz front-end dio weba i što i na koji način donosi u tom sektoru. Objasnit će se utjecaj i kako će unaprijediti ostale ne-web tehnologije. Na temelju određenih mjerenja u performansama pri izvršavanju zadataka izvršit će se usporedbe između JavaScript-a i drugih jezika kojima je omogućen direktniji pristup web-u. Neke matematičke ili slične operacije je bolje odraditi korištenjem nekog drugog programskog jezika poput C-a. U konačnici zadatak je demonstrirati prednosti ovog standarda i pokazati kako će budućnost weba biti promijenjena koristeći WASM.

2. UVOD U WEB I WEBASSEMBLY

2.1. Povijest Weba

Današnji web nastao je 1989. godine. Tim Berners-Lee je izmislio „*World Wide Web*“ dok je radio u CERN-u[1]. Web je izvorno zamišljen i razvijen kako bi udovoljio potražnji za automatiziranom razmjenom informacija između znanstvenika sa sveučilišta i instituta širom svijeta. CERN nije izolirani laboratorij, već je žarište široke zajednice koja obuhvaća više od 17 000 znanstvenika iz preko 100 zemalja. Iako obično provode neko vrijeme na mjestu CERN-a, znanstvenici obično rade na sveučilištima i nacionalnim laboratorijima u svojim matičnim zemljama. Pouzdani komunikacijski alati su stoga neophodni. Osnovna ideja WWW-a bila je spojiti nove tehnologije računala, podatkovnih mreža i hiperteksta u moćan i jednostavan za korištenje globalni informacijski sustav. Tim Berners-Lee napisao je prvi prijedlog za World Wide Web u ožujku 1989., a svoj drugi prijedlog u svibnju 1990. zajedno s belgijskim inženjerom sustava Robertom Cailliauom, ovo je formalizirano kao prijedlog upravljanja u studenom 1990. Takvim prijedlogom iznijeli su se glavni koncepti i definirali važni pojmovi iza weba. U dokumentu je opisan „hipertekstualni projekt“ nazvan „WorldWideWeb“ u kojem su „preglednici" mogli pregledati „mrežu hipertekstualnih datoteka“. Krajem 1990. Tim Berners-Lee imao je prvi web poslužitelj i preglednik podignut i pokrenut u CERN-u koji je demonstrirao njegove ideje. Kôd za svoj web poslužitelj razvio je na NeXT računalu. http://info.cern.ch bila je adresa prve web stranice i web poslužitelja na svijetu koja je radila na NeXT računalu u CERN-u (***Slika 2.1.***)



Sl. 2.1.Prvi (NeXT) www poslužitelj

Adresa prve web stranice bila je koja je sadržavala poveznice na informacije o samom WWW projektu bila je http://info.cern.ch/hypertext/WWW/TheProject.html. Navedena stranica je uključivala i opis hiperteksta, tehničke detalje za stvaranje web poslužitelja i veze do drugih web poslužitelja čim su postali dostupni. Tako je nakon nekog vremena nastao pojam *Web 1.0*. Iako se u to vrijeme zapravo nikada nije koristio za opisivanje weba, ovaj se izraz danas koristi za osnovne web stranice koje pružaju ograničeno ili statično korisničko iskustvo[2]. Ovaj bi se izraz koristio za opisivanje jednostavnih web stranica s prikazom proizvoda neke trgovine. Naravno, e-trgovina je bila glavno područje eksplozije koje je web prvi put koristilo na komercijalni način. To je potom evoluiralo tako da uključuje web stranice poput ebay-a za mrežne dražbe. Uključio je i prvi val radnih mjesta kao što su Monster i Hot Jobs. Kako se tehnologija razvijala, web stranice su mogle pružiti bogatije i interaktivnije korisničko iskustvo. *Web 2.0* korišten je za opis novijeg web fenomena. Primjeri takvih web stranica su bile stranice društvenih mreža kao što su Facebook i MySpace, koje su korisnicima omogućavale stvaranje vlastitih profila, dijeljenje datoteka i veću međusobnu interakciju. YouTube je omogućio ljudima da objavljuju vlastite medijske sadržaje, „bloganje“ je omogućilo prosječnom mrežnom korisniku da postane kućni novinar, a Wikipedia je omogućila mrežnoj zajednici da razvije najopsežniju enciklopediju ikad stvorenu. Na tehničkoj razini, tehnologija koja pruža „kralježnicu“ mnogim web adresama u ovo doba omogućila je web stranicama da glatko ažuriraju sadržaj i bez osvježavanja stranica. Ukratko, ako se Web 2.0 zalaže za bilo što, to je stvaranje i dijeljenje informacija isporučenih putem weba. Budući korak evolucije weba je *Web 3.0* koji je trenutno u ranim počecima, ali još nije u potpunosti definiran. Pojam Web 3.0 stvarno opisuje trenutnu evolucijsku fazu weba i poprima brojne oblike, od umjetno inteligentnih aplikacija koje predviđaju buduće trendove do inovativnih web usluga s profitabilnim poslovnim modelima i čitavih trodimenzionalnih virtualnih svjetova koji omogućavaju ljudima interakciju sa svakim poput igra Second Life. Jedan od glavnih koncepata Web 3.0-a je „mreža podataka“ ili „semantička mreža“, koja u principu uključuje dostupnost strukturiranih podataka na *Internetu*. Postojeće web stranice dizajnirane su da ih ljudi čitaju, a ne strojevi. To znači da čovjek može na webu tražiti neki proizvod, npr. kalkulator koji je jeftin i nudi određene mogućnosti, dok bi se računalo mučilo da to učini. Kako se tehnologija bude razvijala, tako će inteligentni softver moći izvršavati ove zadatke umjesto vas. RSS (skraćeno Really Simple Syndication) feedovi i dijeljenje podataka također će se i dalje razvijati i igrati glavnu ulogu, kao i noviji koncepti poput mikro bloganja, odnosno bloganje s mobilnog telefona. Korisnik će moći fotografirati gdje god se nalazio i ta fotografija će se automatski prenijeti na korisnikov blog i biti označena GPS-om kako bi korisnik znao gdje je bio i kad je snimljena. Poslovnim aplikacijama kojima se prije moglo pristupiti isključivo kupnjom i instalacijom sada se lako može pristupiti putem web preglednika.

U osnovi se pojavljuju sve više niša web usluga koje sve mogu dijeliti i pristupati „oblaku“ podataka. To je u velikoj mjeri olakšano internetskim standardima kao što je XML, a posebno HR-XML za regrutnu industriju.

2.2. Struktura Web-a

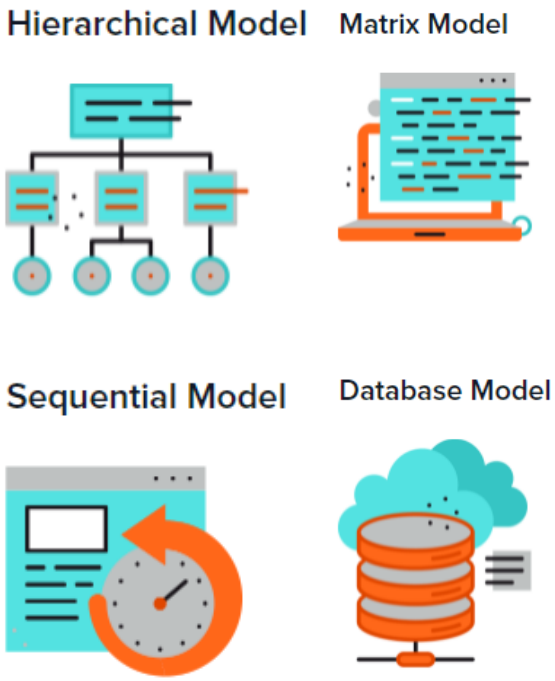
2.2.1. Prikaz stranica i korisničko iskustvo (engl. User Experience)

Struktura web stranice pomaže u oblikovanju razumljivih, otkrivenih i predvidljivih uzoraka. Ispravna struktura web stranice pomaže posjetiteljima web stranice da lako pronađu informacije putem dosljednosti. Korisnici se osjećaju zadovoljno kada brzo pronađu informacije, a čvrsta i relativna struktura bitna je za iskoristivost web stranice. Korisničko iskustvo, skraćeno na engleskom UX, dizajneri mogu riješiti šire probleme u dizajnu i upotrebljivosti korisničkog sučelja dobrom strukturiranjem web stranica. Za grupiranje i katalogizaciju sadržaja potrebna je dobra struktura web stranica. Kad razmatraju potencijalne arhitekture, dizajneri obično mogu birati između pristupa odozgo prema dolje ili odozdo prema gore na temelju potreba svojih korisnika i poslovnih ciljeva [3]. Pristup od vrha prema dolje usmjeren je prvo na opće kategorije sadržaja, tipa vijesti. Dizajneri mogu logički podijeliti sadržaj postupnim razbijanjem u kategorije. To može pomoći u informiranju taksonomije ili hijerarhijske strukture web mjesta. Pristup odozdo prema gore suprotan je pristupu odozgo prema dolje. Tamo gdje se pristup odozgo prema dolje fokusira na katalogizaciju sadržaja u kategorije, pristup odozdo prema gore fokusira se prvo na stvaranje strukture koja se temelji na sadržaju koji je dostupan web mjestu, npr. određeni članak. Prvo se grupiraju elementi u kategorije najniže razine, a zatim njihovim grupiranjem kategorije u one više razine.

Tipovi modela prikaza su:

- Hijerarhijski model - često se koristi u web aplikacijama koje sadrže veliku količinu podataka
- Sekvencijalni model – popularni su kada se korisnike vodi kroz niz stranica, npr. stvaranje novog korisničkog računa
- Matrični model – popularan u ranim fazama weba, a daje korisnicima mogućnost odabira kamo žele dalje

- Model baze podataka - dinamičan je pristup strukturi web stranice, koristeći ovu strukturu korisnici mogu stvoriti iskustva na temelju svojih upita za pretraživanje



Sl. 2.2. Tipovi web struktura

2.2.2. Struktura koda

Za stvaranje web stranice potreban je HTML datoteka koja koristi tri tri osnovna elementa, odnosno oznake koje se koriste na svim stranicama, a to su *html*, *head* i *body*. HTML (*HyperText Markup Language)* je osnovni jezik za izradu web stranica. Koristeći HTML web preglednici dobivaju podatke o sadržaju i strukturi stranice[4]. Preglednik zatim oblikuje strukturu i u konačnici prikazuje web stranicu krajnjem korisniku. Obzirom da on služi za stvaranje hipertekstualnih datoteka i da ga se koristi za prikaz stranice on nije programski jezik[5]. HTML5 je najnovija verzija HTML-a i koriste ga oni koji stvaraju današnje web stranice. HTML datoteke počinju deklaracijom *doctype*. Ovaj redak koda omogućuje pregledniku da zna verziju HTML-a na kojoj je stranica napisana. Prva linija svake HTML datoteke, a i pomoću kojeg označavamo najnoviju verziju, ovdje će biti korišten HTML5, je:

Linija ***Kod***

```
1:      <!DOCTYPE html>
```

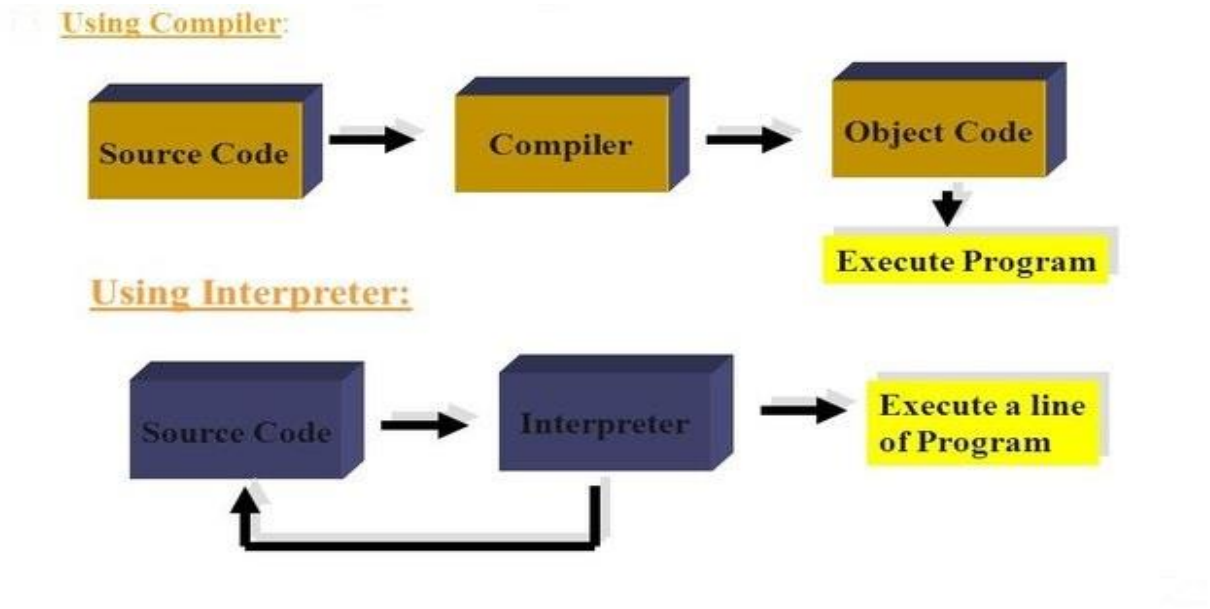
Sl. 2.3. Početak HTML datoteka

Nakon deklaracije verzije potrebno je koristiti *html* oznaku kako bi preglednik znao da je ostatak koda napisan u html formatu, a kada se završava kôd onda je bitno zatvoriti html tag.

Sl. 2.4. Struktura HTML datoteke

Unutar svakog *head* oznake se dodaje naslov stranice, unutar *title* oznaka, koji će biti prikazan na kartici stranice. Dodaju se metapodaci koji olakšavaju pronalaženje vaše stranice pomoću optimizacije pretraživača (Search Engine Optimization, skraćeno SEO), tip simbola koji će biti korišteni za izradu, najčešće korišten je „UTF-8“. Također, unutar *heada* se dodaju *link* tagovi koji uključuju CSS i JavaScript datoteke, iako se JavaScript datoteke mogu dodati i unutar *body* oznake. *Cascading Style Sheet*, skraćeno CSS, označava prezentacijski jezik pomoću kojeg se dodaje stil strukturi stranice element po element. Obzirom da su se HTML elementi u prošlosti mogli oblikovati to je dovelo do problema prilikom izrade kompleksnijih stranica jer se svaki put moralo sve iznova pisati i postavljati na poslužitelj. Pojavom CSS-a se riješio problem odvajanja stila od kostura stranice. Kako se web dalje razvijao potrebno je bilo napraviti stranice i njihove elemente interaktivnima. To je bilo moguće *JavaScriptom* i *PHP-om*, moguće je i *Pythonom*, ali se on nije popularizirao do nedavno i najčešće ga se koristi u druge svrhe. Ovi jezici omogućuju

izradu određenih funkcionalnosti i na backend i frontendu weba. Odmicanjem u budućnost nastali su razni softverski okviri što znači da je svaki od ovih jezika ugrađen u konceptualnu strukturu koja olakšava izradu web stranice u kompletu. Ograničenost ovih jezika je što su oni skriptni. Skriptni jezici podržavaju skripte, odnosno programe za automatsko izvršavanje nekih zadataka[6]. Kako bi se kôd preveo i obradio na stranici koristi se tumač (*engl*. interpreter), a ne prevodilac [7]. Razlika je što prevoditelji pretvaraju izvorni kôd u strojni jezik te ga obrađuje procesor računala, a pomoću interpretera sustav automatski izvodi potrebne akcije.[8] Prepreka kôd interpretera je ta što se prilikom okidanja funkcije kôd svaki puta mora tumačiti na visokoj razni. Prevodilac za razliku od njega jednom prevede cijeli kôd u strojni te on po potrebi radi što treba i ne izvodi se ponovna pretvorba koda u strojni jezik.



Sl. 2.5. Usporedba ciklusa prevoditelja i interpretera prilikom izvršavanja koda

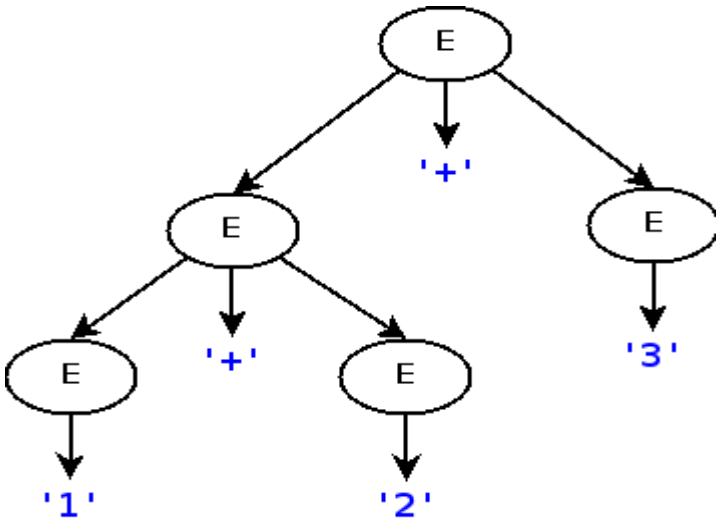
Iako skriptni jezici olakšavaju programiranje jer je korak prevođenja izostavljen, potreba za sistemskim jezicima poput C-a, C++, itd. se javlja razvojem weba generalno, pogotovo za fazu Web 3.0.

2.3. WebAssembly

2.3.1. Uvod

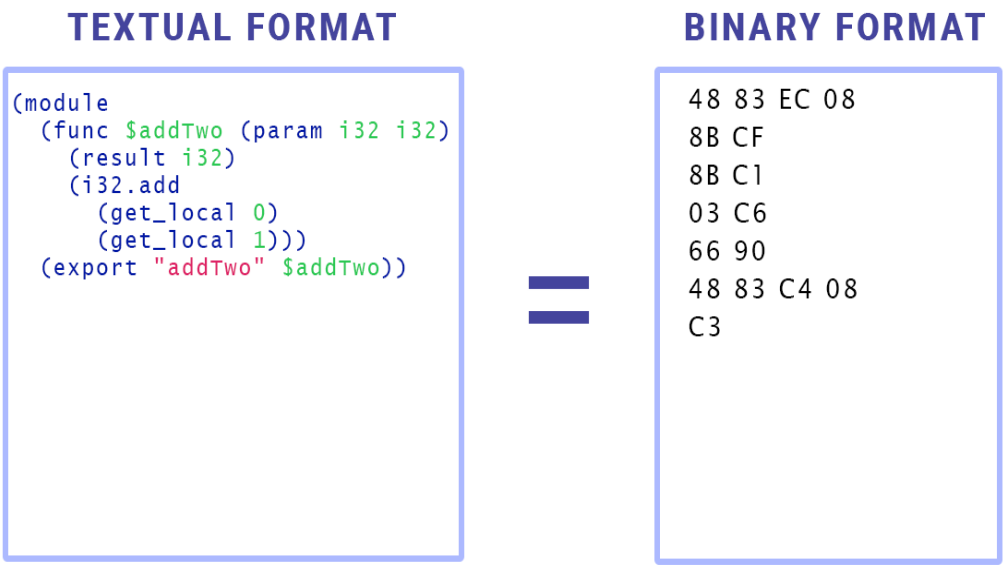
WebAssembly (skraćeno Wasm) je binarni format instrukcija za virtualni stroj zasnovan na stogu (*engl.* stock-based). WebAssembly zapravo definira apstraktno sintaksno stablo koje se

pohranjuje u binarnom formatu[9]. Apstraktno sintaksno stablo je stablo čiji su čvorovi označeni operatorima, a listovi predstavljaju operande (***Slika 2.6.***).



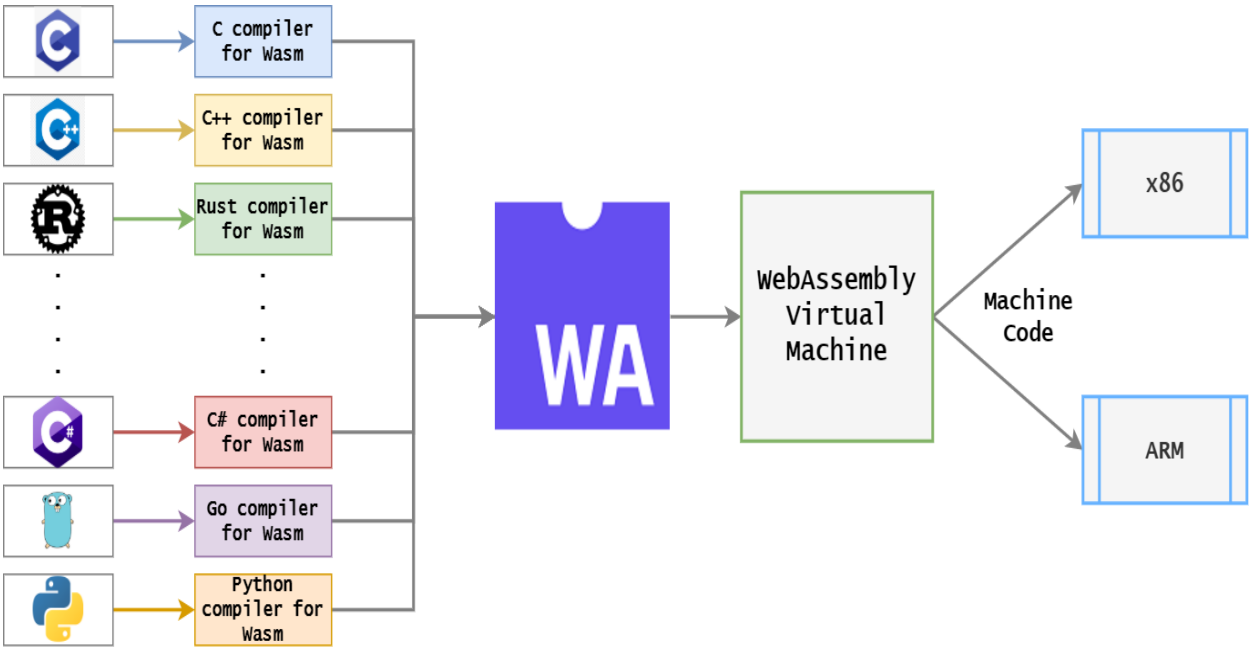
Sl. 2.6. Apstraktno sintaksno stablo

Jezik je asemblerski-baziran, odnosno programski jezik niske razine koji je baziran na asembleru [10]. WebAssembly je dizajniran kao „prijenosni prevodilac“ za programske jezike, omogućavajući upotrebu na webu za klijentske i poslužiteljske aplikacije što znači da se ne piše direktno taj jezik nego je on posrednik, odnosno „Intermedijarni Zastupnik“ (*engl.* Intermediate Representation, skraćeno IR), između određenih programskih jezika i asemblera. Znači da se može izvoditi u modernim web preglednicima, a prevodilac prevodi određeni jezik u strojni jezik pomoću posebnog virtualnog stroja koji pretvara IR kôd u strojni kôd. Za *debugiranje*, odnosno ispravljanje grešaka, u budućnosti će se pojaviti program za ispravljanje tih grešaka, a do tada je apstraktno sintaksno stablo predstavljeno u umjereno „prijateljskom“ formatu teksta.



Sl. 2.7.Pretvaranje tekstualnog formata u binarni za zbroj dva broja

Radi gotovo sa izvornim performansama i pruža jezike kao što C, C++, C# i Rust[11]. Uz neke od ovih mogućnosti dizajniran je za pokretanje zajedno s JavaScriptom omogućavajući oboma rad zajedno. Suradnja s JavaScriptom omogućuje da se iskoriste performanse i snaga WebAssembly-ja te ekspresivnost i fleksibilnost JavaScript-a u istim aplikacijama čak i ako je korisnicima nepoznato pisanje WASM koda. Njegove datoteke imaju *.wasm na kraju imena, odnosno wasm format.



Sl. 2.8. Pretvaranja jezika visoke razine kroz WebAssembly u strojni kôd

WebAssembly odbor također „gura“ ideju i radi na tome da pruži standardni set sučelja tako da - WASM aplikacije napisane na različitim jezicima mogu pokretati platforme, međusobno neometano komunicirati, moći pristupiti potrebnim API-jevima sustava i sigurno se izvršiti unutar zaštićenog okruženja (s kontroliranim pristupom, memorijom, mrežom, datotečnim sustavom itd.). Zaštićeno okruženje je izolirano testno okruženje koje omogućava korisnicima pokretanje programa ili izvršavanje datoteka bez utjecaja na aplikaciju, sustav ili platformu na kojoj rade. To sustavno sučelje se naziva „WebAssembly Sytem Interface“, skraćeno WASI.

2.3.2. Povijest

WebAssembly kao ideja još nije postojala, ali u periodu 2008.-2009. godine počelo se razmišljati o brzini koju će JavaScript kao jezik moći postići i kako će biti primjenjiv za druge tipove zadataka koje zahtijevaju puno računskih operacija. Tih godina se JavaScript počeo posebno jako i brzo razvijati, kao i njegova zajednica, zbog toga što je tada izbačen Google Chrome s njegovim mehanizmom za razvoj i upravljanje V8 koji je ubrzavao cijeli proces obrade naredbi. Razvojni programer Alon Zakai, trenutno radi za Google, počeo se baviti pokušajem rješavanja skriptnih jezika tako što je preveo C++ u JavaScript. U počecima je to eksperimentirao na računanju π. Takav prevodilac je tada nazvao *Emscripten* zbog asocijacije na popularnu seriju „Simpsons“, a koristio je Low Level Virtual Machine, odnosno njegov IR. U 2013. je u pozadini LLVM-a napisan novi backend s nadimkom "fastcomp". Dizajniran je za emitiranje asm.js-a, što je hakirano u ranijoj JS pozadini, no to nije bilo dobro rješenje. Kao rezultat toga došlo je do velikog poboljšanja u kvaliteti koda i vremenu sastavljanja. Obzirom da su JavaScript Just In Time, skraćeno JIT, optimizacije bile nepredvidive u 2013. Alon uz Luke Wagnera i David Hermana dolaze na ideju za razvitak *asm.js*. To je razvilo temelj današnjeg WebAssemblyja, a koristio bi sustav tipa *asm.js* tako što bi se u JavaScript mehanizmom za razvoj i upravljanje dodao izraz „use asm“ i dodala bi se Ahead Of Time, skraćeno AOT, kompilacija ako se želi. Inicijalne optimizacije u Firefoxu su dostizale oko 50% izvorne brzine, a s vremenom je i ta brzina rasla. U 2014. Unity objavljuje podršku za *Emscripten/asm.js* kao budućnost Unityja koji će se naći na Webu. Prvi je put najavljen 17.6.2015., a prva demonstracija je bila izvršavanje Unityjevih „Angry Bots“ u Firefoxu, Google Chromeu i Microsoft Edgeu[12]. Prethodne tehnologije bile su asm.js iz Mozille i Google Native Client, a inicijalna implementacija temeljila se na skupu značajki asm.js. Tehnologija asm.js već pruža gotovo „izvorne“ brzine izvršavanja koda i može se smatrati održivom alternativom za preglednike koji ga ne podržavaju ili su ga onemogućili iz sigurnosnih

razloga. U ožujku 2017. proglašen je završnim dizajn minimalno održivog proizvoda (*engl.* minimum viable product, skraćeno MVP) i završena faza pregleda. Krajem rujna 2017. godine izdan je Safari 11 koji podržava WASM. U veljači 2018. godine WebAssembly Working Group objavila je tri javna radna nacrta za osnovnu specifikaciju, JavaScript sučelje i Web API. Od 2019. godine postaje službeni web jezik uz HTML, CSS i JavaScript.

2.3.3. Upotrebljivost

WebAssembly već utječe, a i posebno će utjecati na razne grane izrade softvera, konkretnije: aplikacija, programa, igara i sličnih tehnoloških rješenja koja će biti i unutar preglednika, a i van preglednika. Primjeri na koje će utjecati unutar preglednika su:

- Bolje izvršavanje za jezika i alate koji su trenutno unakrsno prevedeni na webu (C / C++, …)
- Uređivanje slika i video zapisa
- Igre: manje igre koje trebaju brzo započeti, AAA igre koje će imati puno zadataka obrade, portali za igre, odnosno one koje „surađuju“
- Peer-to-peer aplikacije, tj. one koje imaju komunikaciju direktno između korisnika
- Streaming i sve što se izvodi „uživo“: glazba, videozapisi, VR
- Prepoznavanje slika
- CAD aplikacije
- Znanstvene vizualizacije i simulacije, platforme za takve zadatke
- Alati za programere poput prevoditelja, urednika, virtualni strojevi i interpreteri jezika
- VPN
- Enkripcija
- „Debeli“ klijent, odnosno zahtjeve za poslovne aplikacije tipa baze podataka

Uz ove kategorije primjera utjecat će i na mnoge druge unutar preglednika, a neki od slučajeva na koje utječe van preglednika su:

- Usluga distribucije igara (prijenosna i sigurna)
- Izvođenje koda na poslužitelju koji još nije pouzdan
- Aplikacije na samom poslužitelju
- Hibridne izvorne (*engl.* native) aplikacije na mobilnim uređajima
- Simetrični proračuni na više čvorova, odnosno i na više procesora

3. PRIMJENA WEBASSEMBLY-A

3.1. Značajke

WebAssembly generalno je vrlo efikasna tehnologija, a zbog asemblerske funkcionalnosti kao tehnologija pridonosi veliki značaj u brzini izvođenja koda. Također, donosi promjene i u sigurnosti koja se odnosi na memoriju. Sigurnost je jedna od njegovih prednosti zato što se kod izvodi u zaštićenom okruženju što znači da će se moći izvoditi i u postojećim JavaScript virtualnim strojevima. Kada ga se kroz kod ugradi u web primjenjuje sigurnosna pravila istog podrijetla i dozvola koja zadaje preglednik. Dizajniran je da bude ljudski čitljiv, odnosno u tekstualnom formatu zbog otklanjanja pogrešaka, testiranje, eksperimentiranja, optimiziranja, učenja, podučavanja i ručnog pisanja programa. Tekstualni format koristit će se za gledanje izvora WASM modula na webu dok će binarni biti obrađen. Osmišljen je kako bi održao prirodu weba, bazirano na značajkama što znači da će biti baziran na posebnim mogućnostima koje se budu izrađivali, a bit će unazad kompatibilan zbog funkcionalnosti koje će trebati odraditi na samoj web aplikaciji ili stranici. Njegovi moduli moći će se pozivati u, a i izvan JavaScript konteksta i pristupiti funkcionalnosti preglednika putem istih web API-ja kojima se može pristupiti iz JavaScript-a. WASM također podržava ugrađivanje izvan weba što znači da se može nešto napraviti van web okruženja te ako je potrebno primijeniti na webu. Ciljevi dizajna ovog jezika se mogu podijeliti u dvije kategorije na kojima se želi poraditi i pridonijeti webu, a želi se poraditi na semantici i prezentaciji. Cilj im je da semantika bude brza, sigurna i prijenosna, pod prijenosna se misli da bude primjenjiva u svim okolinama i da nije ovisna o programskom jeziku, hardveru i slično. Kod prezentacije se cilja na to da bude efikasna i prijenosna. Kako bi se ostvarili ciljevi na semantici na ovaj način rade na ovim stvarima i na ove načine:

- Brzina: izvršava se s gotovo izvornim performansama koda, iskorištavajući mogućnosti zajedničke gotovo svakom suvremenom hardveru.
- Sigurno: kôd je provjeren i izvršava se u memorijski sigurnom okruženju također, u zaštićenom okruženju, (*engl.* sandboxu) sprječavajući oštećenje podataka ili narušavanje sigurnosti.
- Dobro definirano: cjelovito i precizno definira valjane programe i njihovo ponašanje na način o kojem je lako neformalno i formalno zaključiti.
- Neovisno o hardveru: može se prevoditi na svim modernim arhitekturama, stolnim ili mobilnim uređajima i ugrađenim sustavima.

- Neovisno o jeziku: ne privilegira nijedan određeni jezik, programski model ili objektni model.
- Neovisno o platformi: može se ugraditi u preglednike, pokretati kao samostalni virtualni stroj ili integrirati u druga okruženja.
- Otvoreno: programi mogu surađivati sa svojim okruženjem na jednostavan i univerzalan način.

Na ostvarenju ciljeva efikasne i prijenosne prezentacije kategorizirane su ove značajke i rađeno je na ove načine:

- Kompaktan: ima binarni format koji se brzo prenosi jer je manji od uobičajenih formata teksta ili izvornog koda.
- Modularno: programi se mogu podijeliti na manje dijelove koji se mogu zasebno prenositi, predmemorirati i trošiti.
- Učinkovito: može se dekodirati, provjeriti i kompilirati u brzom jednom prolazu, podjednako ili s pravodobnim (JIT) ili prije vremena (AOT) prevođenjem.
- Mogućnost prijenosa uživo: omogućuje dekodiranje, provjeru valjanosti i kompilaciju što je prije moguće, prije nego što se vide svi podaci.
- Omogućava paralelizam: omogućuje dekodiranje, provjeru valjanosti i kompilaciju u mnoge neovisne paralelne zadatke.
- Prijenosni, odnosno široko primjenjiv: ne donosi arhitektonske pretpostavke koje nisu široko podržane u modernom hardveru.

Kako bi bio široko primjenjiv potrebno je opisati pretpostavke za efikasno izvršavanje koda. Izvršna okruženja koja, unatoč ograničenom, lokalnom, nedeterminizmu, ne nude sljedeće karakteristike, ipak mogu izvršiti module WebAssembly-a. U nekim će slučajevima možda morati oponašati ponašanje koje hardver ili operativni sustav domaćina (*engl.* hosta) ne nudi, tako da se moduli izvršavaju kao da je ponašanje podržano. To će ponekad dovesti do loših performansi. Kako njegova standardizacija napreduje, nastojat će se formalizirati ovi zahtjevi i prilagodba WASM-a novim platformama koje nisu nužno postojale kad je prvi put dizajniran. Neke od karakteristika koje ovaj jezik pretpostavlja da izvršna okruženja imaju su 8-bitni bajtovi, adresabilnost u granulaciji bajtne memorije, podržavanje neusklađenog pristupa memoriji ili nepouzdano hvatanje koji omogućava softversku emulaciju, little-endian poredak bajtova, wasm64 dodatno podržava linearnu memoriju veću od 4GiB s 64-bitnim pokazivačima ili indeksima i mnoge druge. Također, ne navodi API-je ili sustavne pozive, već samo uvozni

mehanizam gdje skup raspoloživih uvoza definira okruženje hosta. U web okruženju, funkcijama se pristupa putem web API-ja definiranih web platformom. Okruženja koja nisu web-okruženja mogu odabrati implementaciju standardnih web API-ja, standardnih ne-web-API-ja, npr. POSIX ili izmišljanje vlastitih. Prenosivost na izvornoj razini (*engl.* source-level) C/C ++ može se postići programiranjem standardnog API-ja (npr. POSIX) i oslanjanjem na prevodilac i/ili biblioteke za mapiranje standardnog sučelja na dostupni uvoz okruženja domaćina ili u vrijeme prevođenja, putem #ifdef, ili izvorno okruženje (*engl.* runtime) što znači putem otkrivanja značajki i dinamičkog učitavanja / povezivanja[13].

3.1.1. Buduće značajke

WebAssembly je i dalje mlada tehnologija koja se tek razvija, a na slici 3.1. su prikazane samo neke od mogućnosti koje se planiraju uvesti u preglednike, Wasmtime, Wasmer i Node.js, a navedeni su i prijedlozi koji su još uvijek na razmatranju.

	Your browser	Chrome 91	Firefox 89	Safari 14.1	Wasmtime 0.22	Wasmer 2.0	Node.js 15.0
Standardized features							
JS BigInt to Wasm i64 integration	✔	✔	✔	✔	n/a	n/a	✔
Bulk memory operations	✔	✔	✔	⌛	✔	✔	✔
Multi-value	✔	✔	✔	✔	✔	✔	✔
Import & export of mutable globals	✔	✔	✔	✔	✔	✔	✔
Reference types	✘	⌛	✔	⌛	✔	✔	⌛
Non-trapping float-to-int conversions	✔	✔	✔	⌛	✔	✔	✔
Sign-extension operations	✔	✔	✔	✔	✔	✔	✔
Fixed-width SIMD	✔	✔	⌛	✘	⌛	✔	⌛
In-progress proposals							
Exception handling	✘	⌛	✘	✘	✘	✘	✘
Module Linking	✘	✘	✘	✘	⌛	✘	✘
Tail calls	✘	⌛	✘	✘	✘	✘	✘
Threads and atomics	✔	✔	✔	⌛	✘	⌛	⌛

Sl. 3.1. *Roadmap* i značajke koje se planiraju uvesti

JavaScriptov BigInt je ugrađeni objekt čiji konstruktor vraća *bigint* primitiv, odnosno vraća vrijednost veću od 2^{53}-1. Za operacije skupne memorije (*engl.* bulk memory operations) mogućnost se smatra da će se uvesti „obilazak“ instanciranja modula svaki put kada ih je potrebno koristiti između višestrukih „agenata“[14]. Takav obilazak će biti omogućen spremanjem

segmenata podataka u zasebni modul koji je instanciran samo jednom, a zatim izvozom ove memorije na upotrebu drugog modula koji sadrži samo kod. Djelotvorno je rješenje, ali je glomazno jer zahtijevaju dva modula gdje jedan treba biti dovoljan. Varijable s više vrijednosti će otvoriti vrata objektno orijentiranom programiranju i stvaranju struktura poput korisničkog računa. Globalne varijable će biti moguće mijenjati, a u web-vezivanju izvezene globalne varijable su tipa WebAssembly.Global, a ne pretvoreni u JavaScriptov Number[15]. Navedene su varijable lokalne agentu, a ne mogu biti dijeljene između agenata što znači da mogu bit korištene kao lokalno skladište niti. Za referenciranje tipova omogućit će se modulima da sadrže reference na JS/DOM objekte pritom da ih prosljeđuje kao argumente, spremajući ih u lokalna i globalna područja i pohranjujući ih u WebAssembly.Table objekte[16]. Što se tiče konverzije *float* i *int* tipova varijable bit će moguće obaviti konverziju jer trenutna LLVMova konverzija daje nedefenirani (*undefined*) rezultat umjesto da daje nedefinirano ponašanje[17]. Za prijedlog SIMD-a, više je sličan hardverskom SIMD-u ako nema SIMD-ovih zamki. Ovim bi se prijedlogom uspostavila konvencija za zasićenje operacija koje bi SIMD mogao dijeliti, kako bi se izbjeglo uvođenje zamki[18]. Koristeći operatore povećanja broja znakova (*engl.* sign-extension operators) mogućnost znači da će se podržavati „produžavanje znakova“, a za to su se dodali novi operatori: *i32.extend8_s*, *i32.extend16_s*, *i64.extend8_s*, *i64.extend16_s*, *i64.extend32_s*. Produžavanje znakova znači da se povećava broj bitova od binarnog broja, a da mu se pritom sačuva „potpisana“ vrijednost (pozitivna/negativna). Jedan od ciljeva visoke razine prijedloga za SIMD je uvođenje vektorskih operacija u specifikacije WASM-a, na način koji jamči prijenosne performanse. Svrha „Fixed-width SIMD“ prijedloga za SIMD je uvođenje vektorskih operacija u specifikacije jezika na način koji jamči prijenosne performanse. Skup operacija uključenih u prijedlog sastoji se od operacija koje su dobro podržane na širokom spektru platformi i dokazano su učinkovite. U tu svrhu trenutni je prijedlog ograničen na standardizaciju SIMD operacija fiksne širine 128-bitnih. Prijedlog je u fazi 4 u kojem V8 i lanac alata (*engl.* toolchain) već imaju funkcionalne implementacije. Ovim će se prijedlogom najviše će se očitovati napredak u obradi slika. Trenutni prijedlog uvodi novi tip vrijednosti *v128* i niz novih operacija koje djeluju na tom tipu.. Kriteriji koji se koriste za određivanje ovih operacija su:

- operacije bi trebale biti dobro podržane u više modernih arhitektura
- pobjede u performansi (*engl.* performance wins) bi trebale biti pozitivne u više relevantnih arhitektura unutar grupe instrukcija
- odabrani skup operacija trebao bi minimalizirati „litice“ izvedbe, ako ih ima

3.2. Sigurnost

Sigurnosni model WebAssembly-a ima dva važna cilja: zaštititi korisnike od programskih pogrešaka ili zlonamjernih modula i pružiti programerima korisne primitive i ublažavanja (*engl.* mitigations) za razvoj sigurnih aplikacija, unutar ograničenja prvog cilja. Njegov dizajn promiče sigurne programe uklanjanjem opasnih značajki iz njegove semantike izvršenja, istovremeno zadržavajući kompatibilnost s programima napisanim za C/C++. Moduli moraju deklarirati sve dostupne funkcije i pripadajuće tipove tijekom učitavanja, čak i kada se koristi dinamičko povezivanje. To omogućuje implicitno provođenje integriteta kontrolnog toka (*engl.* control-flow integrity, skraćeno CFI) kroz strukturirani kontrolni tok. Budući da je prevedeni kôd nepromjenjiv i da ga se ne može primijetiti tijekom izvođenja, programi WASM-a zaštićeni su od napada otmice (*engl.* hijacking attacks) na kontrolni tok. Pozivi funkcije moraju navesti indeks cilja koji odgovara važećem unosu u prostoru indeksa funkcije ili indeksnom prostoru tablice. Neizravni pozivi funkcije podliježu provjeri potpisa tipa tijekom izvođenja; potpis tipa odabrane neizravne funkcije mora odgovarati potpisu tipa navedenom na mjestu poziva. Zaštićeni stog poziva koji je neranjiv za preljeve međuspremnika (*engl.* buffer-overflows) u hrpi modula osigurava siguran povratak funkcije. Podružnice (*engl.* branches) moraju ukazivati na valjana odredišta unutar funkcije zatvaranja. Varijable u C/C++ mogu se smanjiti na dva različita primitiva koristeći ovaj jezik, ovisno o njihovom opsegu. Lokalne varijable s fiksnim opsegom i globalne varijable predstavljene su kao vrijednosti fiksnog tipa pohranjene indeksom. Prvi primitivi su prema zadanim postavkama inicijalizirani na nulu i pohranjeni u zaštićenom nizu poziva, dok su drugi smješteni u globalni indeksni prostor i mogu se uvesti iz vanjskih modula. Lokalne varijable s nejasnim statičkim opsegom, npr. koristi ih operator adrese, ili su tipa *struct* i vraćaju se vrijednošću, pohranjuju se u zasebnom stogu kojemu se može adresirati korisnik u linearnoj memoriji u vrijeme asembliranja. Ovo je izolirano memorijsko područje s fiksnom maksimalnom veličinom koja je prema zadanim postavkama inicijalizirana na nulu. Reference na ovu memoriju izračunavaju se s beskonačnom preciznošću kako bi se izbjeglo umatanje i pojednostavila provjera granica. U budućnosti će se implementirati podrška za više dijelove linearne memorije i preciznije memorijske operacije, npr. zajednička memorija, zaštita stranica, velike stranice itd.. Zamke (*engl.* traps) se koriste za trenutno prekidanje izvršenja i signaliziranje abnormalnog ponašanja okolini izvršenja. U pregledniku je ovo predstavljeno kao JavaScript iznimka. Podrška za modularno definirane rukovatelje zamki bit će implementirana u budućnosti. Operacije koje mogu zarobiti uključuju: specificiranje nevažećeg indeksa u bilo kojem indeksnom prostoru, izvođenje neizravnog poziva funkcije s neusklađenim potpisom, premašuje maksimalnu veličinu zaštićenog niza poziva, pristup adresama izvan granica

u linearnoj memoriji, izvršavanje ilegalnih aritmetičkih operacija, npr. dijeljenje ili ostatak nulom, potpisano prelijevanje dijeljenja itd.. Sve prethodno navedeno će se odnositi na sigurnost za razvojne programere.

Sljedeća razina sigurnosti koju WebAssembly nudi je memorijska sigurnost. U usporedbi s tradicionalnim programima C/C++, ove semantike uklanjaju određene klase sigurnosnih grešaka memorije. Prelijevanje međuspremnika, koje se događa kada podaci prelaze granice objekta i pristupaju susjednim memorijskim regijama, ne mogu utjecati na lokalne ili globalne varijable pohranjene u indeksnom prostoru, one su fiksne veličine i adresirane indeksom. Podaci pohranjeni u linearnoj memoriji mogu prebrisati susjedne objekte, jer se provjera granica izvodi na granularnosti linearne memorijske regije i nije ovisna o kontekstu. Međutim, prisutnost integriteta kontrolnog toka i zaštićenih hrpa poziva sprječava izravne napade ubrizgavanja koda (*engl.* direct code injection attacks). Dakle, uobičajena ublažavanja poput sprječavanja izvršavanja podataka (*engl.* data exectuion, prevention, skraćeno DEP) i zaštite od razbijanja stoga (*engl.*stack smashing protection, skraćeno SSP) nisu potrebni za programe WASM-a. Druga uobičajena klasa sigurnosnih pogrešaka memorije uključuje nesigurnu upotrebu pokazivača i nedefinirano ponašanje. To uključuje pokazivače za preusmjeravanje na neraspoređenu memoriju, npr. NULL ili oslobođene dodjele memorije. U njemu je eliminirana semantika pokazivača za pozive funkcija i varijable s fiksnim statičkim opsegom, što omogućava upućivanje na nevažeće indekse u bilo kojem indeksnom prostoru da pokrenu pogrešku provjere valjanosti tijekom učitavanja ili u najgorem slučaju zamku tijekom izvođenja. Pristupi linearnoj memoriji provjeravaju se granice na razini regije, što potencijalno rezultira zamkom tijekom izvođenja. Ta su područja memorije izolirana od interne memorije izvornog okruženja i prema zadanim postavkama postavljena su na nulu, ako nije drugačije inicijalizirano. Ipak, druge vrste grešaka njegova semantika ne uklanja. Bez obzira što napadači ne mogu izvoditi izravne napade ubrizgavanja koda, moguće je oteti kontrolni tok modula pomoću napada ponovne upotrebe koda protiv neizravnih poziva. Međutim, konvencionalni napadi povratom orijentirano programiranje (*engl.* return-oriented programming, skraćeno ROP), koji koriste kratke sekvence uputa zvani naprave (*engl.* gadgets) nisu mogući u WebAssembly-u jer integritet kontrolnog toka osigurava da su ciljevi poziva valjane funkcije deklarirane u vrijeme učitavanja. Isto tako, uvjeti utke (*engl.* race conditions), poput ranjivosti provjere vremena do vremena upotrebe (*engl.* time of check to time of use, skraćeno TOCTOU), su mogući jer nisu osigurana nikakva jamstva izvršenja ili raspoređivanja osim izvršavanja po redoslijedu i primitiva atomske memorije nakon MVP-a. Slično tome, mogu se dogoditi napadi na bočni kanal (*engl.* side channels attacks) poput vremenskih napada na module. U budućnosti se

dodatnim zaštitama mogu pružiti vremena izvođenja ili lanac alata poput diverzifikacije koda ili odnosno nasumičnog miješanja memorije, slično slučajnom rasporedu adresnog prostora (*engl.* address space layout randomization, skraćeno ASLR) ili ograničenih pokazivača, a ovdje se misli na „debele“ pokazivače.

Učinkovitost integriteta kontrolnog toka (CFI) može se izmjeriti na temelju njegove cjelovitosti. Općenito postoje tri vrste vanjskih prijelaza kontrolnog toka koje treba zaštititi jer se pozivu možda ne može vjerovati, odnosno može biti maliciozan, a ovo su prijelazi:

- izravni pozivi funkcija,
- neizravni pozivi funkcija,
- povrat (*engl.* return)

Izravni i neizravni pozivi funkcija se obično nazivaju prednji rub (*engl.* forward-edge) jer odgovaraju usmjerenim rubovima u usmjerenom grafu upravljanja kontrole toka. Slično pozivima, povrat se obično naziva stražnji rub (*engl.* back-edge) jer odgovara stražnjim rubovima u usmjerenom grafu kontrole toka. Posebni pozivi funkcija poput tragački pozivi (*engl.* tail calls) mogu se promatrati kao kombinacija izravnih poziva funkcija i povrata. Tipično se to provodi pomoću instrumentacije izvornog okruženja. Tijekom prevođenja, prevodilac generira očekivani graf kontrolnog toka izvršavanja programa i uvodi instrumentacije izvornog okruženja na svako mjesto poziva kako bi potvrdio da je prijelaz siguran. Skupovi očekivanih ciljeva poziva grade se iz skupa svih mogućih ciljeva poziva u programu, jedinstveni identifikatori dodjeljuju se svakom skupu, a instrumentacija provjerava je li trenutni cilj poziva član očekivanog skupa ciljeva poziva. Ako je ova provjera uspjela, izvornom pozivu je dopušten nastavak izvođenja, inače se izvršava obrađivač kvara (*engl.* failure handler) koji obično prekida program.

3.3. Performansa

3.3.1. WebAssembly protiv JavaScripta - teorijski

U usporedbi s JavaScriptom WebAssembly je definitivno pobjednik što se tiče brzine izvođenja naredbi. JavaScript je napisan dinamično što znači da nije eksplicitno određen tip određenih podataka do same interpretacije dok se za WASM moraju navesti tipovi podataka koji je zatim „dostavljen“ u malom binarnom formatu. JavaScript je skriptni jezik što znači da se prije samog izvršavanja u pregledniku mora parsirati, prevoditi, optimizirati prije nego što se uopće izvrši. Koristeći ovaj jezik memorijom se ručno upravlja što znači da nema gomile „smeća“ što

bi utjecalo na performanse. WebAssembly datoteke kada se nalaze u predmemoriji (*engl.* cache) brže će se učitati nego JavaScript datoteka. Iako, dok se JavaScript u potpunosti optimizira na Microsoft Edgeu pobijedit će on. Također u izvršnog okruženja performansi pobjeđuje JavaScript. No, prilikom usporedbe manjih i većih polja elemenata događa se nešto neobično, ali vrlo logično. JavaScript je brži u odrađivanju polja podataka malih veličina dok je WASM pobjednik prilikom izvođenja operacija na većim poljima[19]. Usporedba u rezultatima u obrađivanju polja se provelo kroz eksperiment množenja matrica. Eksperiment uključuje pisanje nekoliko implementacija množenja matrica u JavaScript i C ++. Množenje matrica je jednostavno i računski intenzivno ($O(n^3)$ za „naivne“ implementacije). Prva implementacija (*ijk*) pruža polaznu osnovu za usporedbu ostalih implementacija, a gdje jednostavno prelazimo kroz retke i stupce obje matrice.

```
// Javascript - ijk (order of the loops)
function ijk(matrixA, matrixB, matrixC, length) {
  for(let i = 0; i < length; i++){
   for(let j = 0; j < length; j++) {
    for(let k = 0; k < length; k++) {
     matrixC[i * length + j] += (matrixA[i * length + k] * matrixB[j
* length + k]);
    }
   }
  }
 }


// C++ -  ijk (order of the loops)
void ijk(double* matrixA, double* matrixB, double* matrixC, int
length) {
  for(unsigned int i = 0; i < length; i++){
    for(unsigned int j = 0; j < length; j++) {
      for(unsigned int k = 0; k < length; k++) {
        matrixC[i * length + j] += (matrixA[i * length + k] *
matrixB[j * length + k]);
      }
    }
  }
}
```

Sl. 3.2. Implementacija *ijk* za testiranje brzine izvođenja operacija na raznim veličinama polja

Druga implementacija (*kji*) slična je prvoj implementaciji, ali zloupotrebljava privremeni lokalitet, također je sporiji algoritam zbog izmjene najudaljenije petlje (i) s najdubljom petljom

(k). Ova implementacija učinkovito pogoršava performanse hijerarhije memorije uzrokujući više promašaja predmemorije (*engl.* cache misses).

```
// Javascript - kji (order of loops)
 function kji(matrixA, matrixB, matrixC, length) {
  for(let k = 0; k < length; k++) {
   for(let j = 0; j < length; j++) {
    for(let i = 0; i < length; i++){
     matrixC[i * length + j] += (matrixA[i * length + k] * matrixB[j
* length + k]);
    }
   }
  }
 }


// C++ - kji (order of loops)
void kji(double* matrixA, double* matrixB, double* matrixC, int
length) {
  for(unsigned int k = 0; k < length; k++) {
    for(unsigned int j = 0; j < length; j++) {
      for(unsigned int i = 0; i < length; i++){
        matrixC[i * length + j] += (matrixA[i * length + k] *
matrixB[j * length + k]);
      }
    }
  }
}
```

Sl. 3.3. Implementacija *kji* za testiranje brzine izvođenja operacija na raznim veličinama polja

Treća implementacija, množenje blokova matrice (*engl.* block matrix multiplication, skraćeno *bmm*) iskorištava privremeni lokalitet. Ova implementacija koristi prednost tehnike koja se naziva blokiranje. Blokiranje koristi privremenu lokalnost za povećanje predmemorijskih pogodaka (*engl.* cache hits). Veličina bloka odabrana za ovaj eksperiment je 32 jer su veličine polja djeljive sa 32, a poznato je da male veličine blokova dobro funkcioniraju. Uz veličinu blokova od 32 korištene su i veličine blokova od 8 i 16. Implementacije *kji* i *bmm* pokušavaju utvrditi hoće li dobro napisana funkcija u JavaScript-u nadmašiti loše napisanu funkciju u C++.

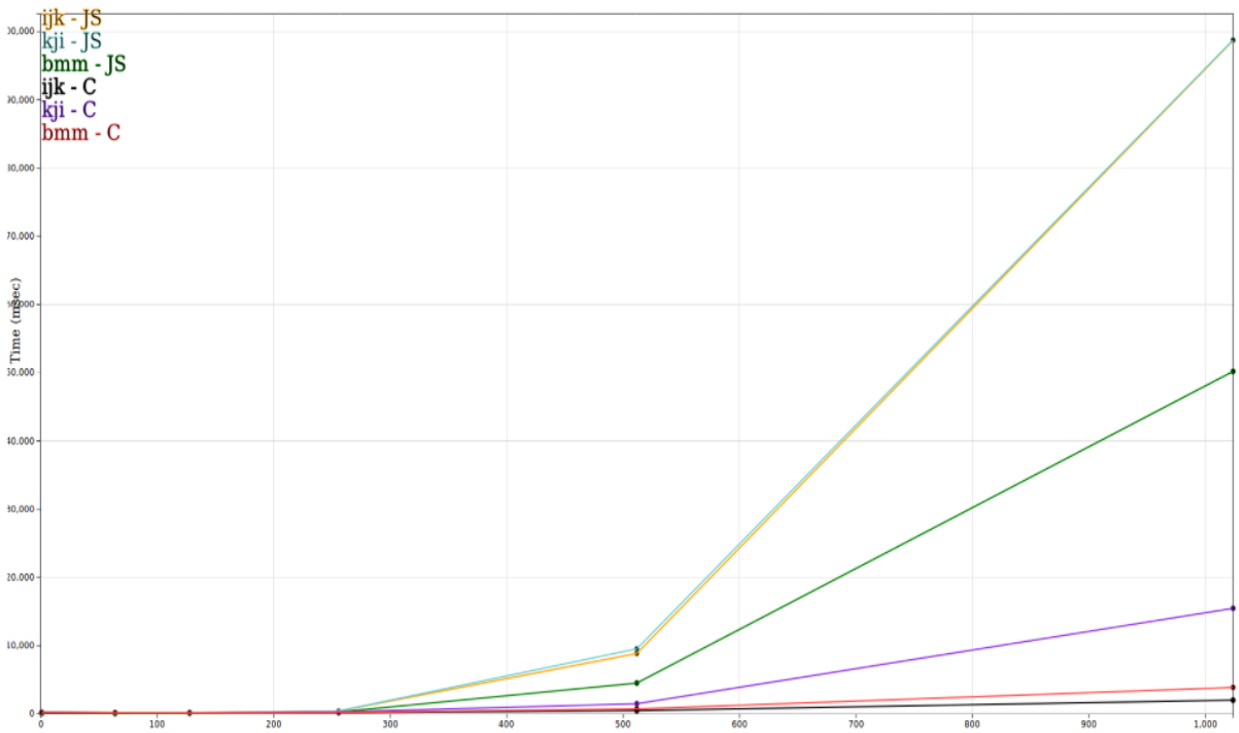
```
// Javascript - bmm (blocked matrix multiplication)
function bmm(matrixA, matrixB, matrixC, length, blockSize) {
  let block = blockSize * (length/blockSize);
  let sum;
  for(let kk = 0; kk < block; kk += blockSize) {
   for(let jj = 0; jj < block; jj += blockSize) {
    for(let i = 0; i < length; i++) {
     for(let j = jj; j < jj + blockSize; j++) {
      sum = matrixC[i * length + j];
      for(let k = kk; k < kk + blockSize; k++) {
       sum += matrixA[i * length + j] * matrixB[i * length + j];
      }
      matrixC[i * length + j] = sum;
     }
    }
   }
  }
 }


// C++ - bmm (blocked matrix multiplication)
void bmm(double* matrixA, double* matrixB, double* matrixC, int
length, int blockSize ) {
  unsigned int block = blockSize * (length/blockSize);
  double sum;


  for (unsigned int kk = 0; kk < block; kk += blockSize) {
    for (unsigned int jj = 0; jj < block; jj += blockSize) {
      for (unsigned int i = 0; i < length; i++) {
        for (unsigned int j = jj; j < jj + blockSize; j++) {
          sum = matrixC[i*length+j];
          for (unsigned int k = kk; k < kk + blockSize; k++) {
            sum += matrixA[i*length+k] * matrixB[k*length+j];
          }
          matrixC[i*length+j] += sum;
        }
      }
    }
  }
}
```

Sl. 3.4. Implementacija *bmm* za testiranje brzine izvođenja operacija na raznim veličinama polja

Konačni rezultati su prikazani na grafovima za svaku od ovih implementacija s legendama gore lijevo u kojima se vidi koji jezik i koja točno implementacija je označena kojom bojom. Rezultati grafova koji su prikazani se odnose na veličinu blokova između 64x64 i 1024x1024.



Sl. 3.5. WebAssembly protiv JavaScripta u računanju matrica

Na ovom grafu je dokazana pobjeda JavaScripta na manjim poljima jer je za implementaciju *bmm* u C-u odma na na početku vidljivija crvena crta od ostalih što znači da mu je trebalo više vremena da obradi te veličina. Na veličinama većim od 500 JavaScript je naglo skočio, a C se jedva odmaknuo od osi apscise koja ovdje predstavlja veličinu polja. Po ovom grafu ispada da je *ijk* implementacija bolja od *bmm* što u teoriji ne bi trebalo biti tako, ali izgleda da se provoditelju testa dogodila greška u samom prevođenju.

3.3.2. WebAssembly protiv JavaScripta – eksperiment

Za ovaj rad se koristio eksperiment u kojem će se usporediti brzina izvođenja Fibonaccijevog niza. Leonardo Fibonacci je bio poznati matematičar iz Republike Pise. Postavio je i riješio problem koji uključuje rast populacije zečeva na temelju idealiziranih pretpostavki. Tako je nastavio Fibonaccijev niz, u kojem je rezultat broja zbroj dva prethodna, a formula glasi:

$$Fn = Fm + Fk$$
$$n > m > k;\ m = n\text{-}1,\ k{=}n\text{-}2;\ n = 0, 1, 2, 3, \ldots \qquad (3.4.2.1.)$$

Naime, 0 i 1 su uvijek postavljeni kao početni brojevi. Za eksperiment se tekstualni format (*wat*) pretvorio *.wasm* tip datoteku koja će biti učitana u JavaScript datoteku i pomoću NodeJS-a

biti pokrenuta, a rezultati će biti ispisani u konzoli. Vrijeme će se mjeriti u milisekundama. Prvi korak je pisanje kod u tekstualnom formatu.

Obzirom da je eksperiment vrlo jednostavan, za ovaj zadatak se koristio online pretvarač koji se nalazi na ovoj poveznici: https://webassembly.github.io/wabt/demo/wat2wasm/. U WAT odjeljku se zalijepi kôd sa slike 3.6. te se stisne na gumb Preuzmi (*engl.* Download) kako bi se preuzela datoteka u *.wasm* formatu.

Linija ***Kod***

```
1:      (module
2:       (export "fibonacciWASM" (func $fib))
3:       (func $fib (param $n i32) (result i32)
4:       (if
5:        (i32.lt_s
6:         (get_local $n)
7:         (i32.const 2)
8:        )
9:        (return
10:        (i32.const 1)
11:       )
12:      )
13:      (return
14:       (i32.add
15:        (call $fib
16:         (i32.sub
17:          (get_local $n)
18:          (i32.const 2)
19:         )
20:        )
21:        (call $fib
22:         (i32.sub
23:          (get_local $n)
24:          (i32.const 1)
25:          )
26:         )
27:        )
28:       )
29:      )
30:     )
```

Sl. 3.6. Tekstualni format WebAssembly-a za računanje Fibonaccija

Kôd najprije kreće s izgradnjom modula koji će se koristiti. Zatim se navodi varijabla koja se izvozi kao rezultat i koja će se koristiti unutar JavaScript datoteke, a to je *fibonacciWASM*, a u

nastavku je napisana funkcija iz koje će se spremiti rezultat (funkcija *fib*). Funkcija kreće s ključnom riječju *func* te se navodi naziv i parametar koji će primiti i tip parametra. Nešto što do sada nije često korišteno u programskim jezicima je to da se navodi tip rezultata. Zatim se provjerava je li broj „potpisan“ s manje od, što označava *lt_s* (signed less than). Parametar se dohvaća ključnom riječju *get* i uz sebe ima *local* što znači da je unutar ovog opsega (*engl.* scopea) funkcije. Kada se koristi konstanta koristi se ključna riječ *const* i navodi se broj. Taj dio koda provjerava je li trenutni broj *n* manji od 2, a ako je istina vratit će 1. U nastavku koda slijedi izvedba Fibonaccijevog niza s naredbom *return* koja vraća zbroj dva broja. Zbroj se u tekstualnom formatu radi koristeći operaciju *i32.sub* što znači da zbraja dva broja tipa *i32*. Rekurzivni poziv se odvija koristeći *call* naredbu. Funkcija se repetitivno ponavlja dok god se parametar *n* ne umanji do 1.

JavaScript datoteka se sastoji od tri funkcije: *mainFunction, measurePerformanceTime* i *fibonacciJS* te određenih sustavnih varijabli koje su potrebne za izvedbu eksperimenta. Sustavne varijable su potrebne za čitanje dohvaćanje datoteka, čitanje datoteka i provedbu mjerenja. Unutar *mainFunction* se dohvaća *.wasm* datoteka i instancira rezultat iz nje te se pokreće funkcija za mjerenje vremena izvođenja računanja Fibonaccijevog niza za WebAssembly, a zaim JavaScript. Mjerenje se provodi u *measurePerformanceTime* tako da se 30 puta izvede niz i pritom računa vrijeme, a za rezultat se ispiše vrijeme svih 30 iteracija i prosječno vrijeme jedne iteracije. Unutar *fibonacciJS* funkcije se računa Fibonacci.

Linija ***Kod***

```
1:      const fs = require('fs')
2:      const util = require('util')
3:      const path = require('path')
4:      const readFile = util.promisify(fs.readFile)
5:      const { PerformanceObserver, performance } = require('perf_hooks')
6:
7:      async function mainFunction () {
          const wasmFile = await readFile(path.resolve(__dirname,
8:        ./fibonacci.wasm'))
          const res =
9:        await WebAssembly.instantiate(new Uint8Array(wasmFile.buffer))
10:       const { fibonacciWASM } = res.instance.exports
11:       measurePerformanceTime('WebAssembly RESULTS', fibonacciWASM)
12:       measurePerformanceTime('JavaScript RESULTS', fibonacciJS)
13:     }
14:
15:     function measurePerformanceTime(languageLabel, fn) {
16:        const ITERATIONS_NUMBER = 29
17:        const observations = []
```

Sl. 3.7. Usporedba brzine izvođenja Fibonaccijevog niza u JavaScript datoteci (1)

Linija ***Kod***

```
18:        const observationObject = new PerformanceObserver(list => {
19:          const entries = list.getEntries()
20:          observations.push(...entries)
21:          performance.clearMarks()
22:        })
23:        observationObject.observe({ entryTypes: ['measure'] })
24:       for (let i = 0; i <= ITERATIONS_NUMBER; ++i) {
25:         performance.mark('TimeBefore')
26:         fn(40)
27:         performance.mark('TimeAfter')
            performance.measure('TimeBefore to TimeAfter', 'TimeBefore',
28:         'TimeAfter')
29:       }
          const totalTime =
30:         observations.reduce((total, e) => total + e.duration, 0)
31:       const averageTime = totalTime / observations.length
32:       const totalTimeFloat = Number.parseFloat(totalTime).toFixed(2)
33:       const averageTimeFloat = Number.parseFloat(averageTime).toFixed(2)
          languageLabel === 'WebAssembly RESULTS' ?
           console.log("WEBASSEMBLY vs JAVASCRIPT Fibonacci experiment
           \n-----------------------------------------------------    ")
34:        : console.log();
         console.log(`${languageLabel} \n
       Total execution time for all iterations: ${totalTimeFloat} ms
35:    \nAverage execution time for one iteration: ${averageTimeFloat} ms`)
         console.log("-----------------------------------------------------
36:    ")
37:    }
38:
39:    function fibonacciJS (inputNumber) {
40:      if (inputNumber <= 1) return 1
41:      return fibonacciJS(inputNumber - 1) + fibonacciJS(inputNumber - 2)
42:    }
43:
44:    mainFunction()
```

Sl. 3.8. Usporedba brzine izvođenja Fibonaccijevog niza u JavaScript datoteci (2)

Kako bi uspjeli pokrenuti testiranje potreban je NodeJS na računalu koji se može preuzeti na njihovoj službenoj stranici s ove poveznice: https://nodejs.org/en/. NodeJS je JavaScript izvorno okruženje izgrađeno na Chromeovom V8 JavaScript mehanizmu. Datoteka se pokreće naredbom unutar terminala, odnosno konzole, *node index.js*. Obzirom da se provodi 30 iteracija za Fibonaccijev niz do broja 40 treba se sačekati neko vrijeme kako bi se provele kalkulacije, a konačni rezultat izgleda ovako:

```
WEBASSEMBLY vs JAVASCRIPT Fibonacci experiment
-----------------------------------------------------
WebAssembly RESULTS
Total execution time for all iterations: 33364.86 ms
Average execution time for one iteration: 1112.16 ms
-----------------------------------------------------

JavaScript RESULTS
Total execution time for all iterations: 52635.46 ms
Average execution time for one iteration: 1754.52 ms
-----------------------------------------------------
```

Sl. 3.9. Rezultati eksperimenta u brzini izvođenja Fibonaccijevog niza

Zaključak eksperimenta je kako je WebAssembly, sada i dokazano, značajno brži za izvršavanje računskih operacija i to za gotovo dvostruko. Obzirom da je ovo vrlo jednostavan eksperiment i jednostavniji zadatak za napraviti WebAssembly-eva brzina i efikasnost se očituje u raznim granama kao što je već navedeno. U sljedećem potpoglavlju će se spomenut konkretna primjena na web i ne-web sektore.

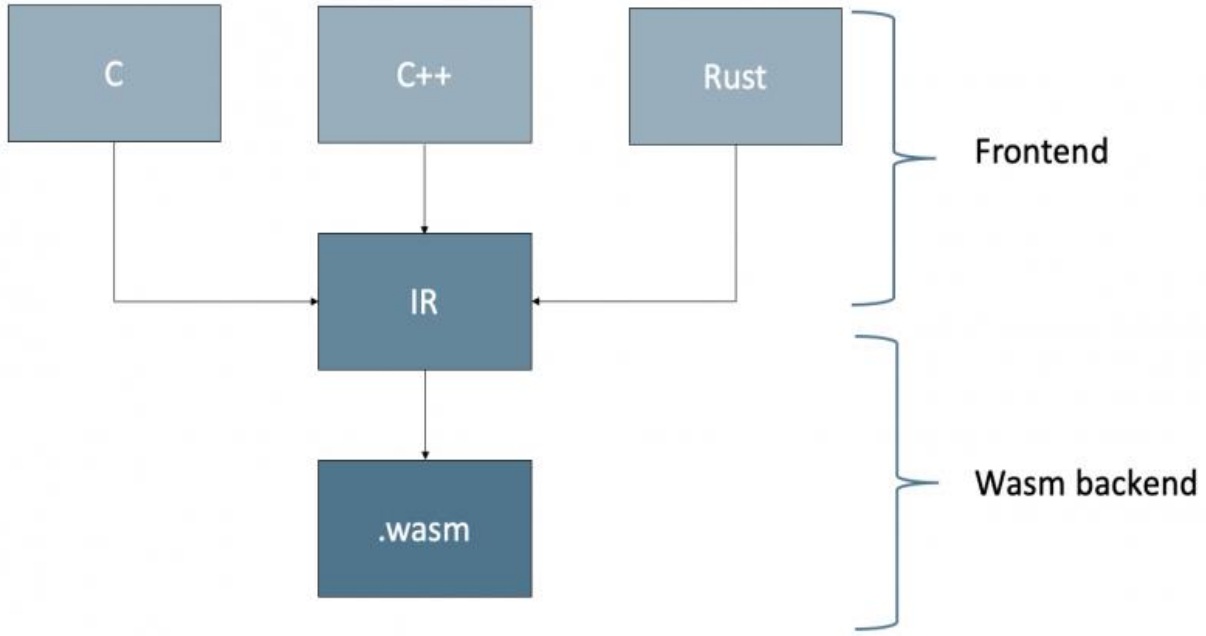
3.4. Ugrađivanje u web tehnologije

Nije iznenađujuće što je jedna od primarnih svrha WebAssembly-a pokretanje na webu, npr. da je ugrađen u web preglednike, iako to nije njegova jedina svrha. To znači integriranje s web ekosustavom, iskorištavanje web API-ja, podrška web modelu sigurnosti, očuvanje prenosivosti weba i osmišljavanje prostora za razvoj. Mnogi od tih ciljeva jasno se odražavaju u ciljevima na visokoj razini. Konkretno, WASM-ov MVP neće biti lošiji sa sigurnosne točke gledišta u usporedbi s JavaScript modulom. Omogućen je JavaScript API koji JavaScriptu omogućuje prevođenje njegovih modula, izvođenje ograničenih refleksija na prevedenim modulima, spremanje i preuzimanje prevedenih modula iz izvanmrežne pohrane, instanciranje prevedenih modula s JavaScript uvozom, pozivanje izvezenih funkcija instanciranih modula, itd. Preglednici, JavaScript motori i izvanmrežni alati imaju uobičajene načine upućivanja na JavaScript artefakte i jezične konstrukcije. Na primjer, lokacije u izvornom kodu JavaScript ispisuju se u tragovima stogova ili porukama pogreške i prirodno su predstavljene kao linije i stupci decimalnog formata

u tekstualnim datotekama. Imena funkcija i varijabli preuzeta su izravno iz izvora. Primjera radi, iako se točan format nizova Error.stack ne podudara uvijek, lokacije su lako razumljive i iste u svim preglednicima. Također, moduli omogućuju prirodnu integraciju sa sustavom ES6 modula.

3.4.1. Frontend

Kako je koristeći WebAssembly olakšan pristup drugim razvojnim programerima koji nisu koristili JavaScript i njegove softverske okvire, zajednica frontend razvojnih programera se počela povećavati. Naročito su se počeli priključivati razvojni programeri koji koriste vrlo popularan Rust programski jezik, C, Go i ostali. Već postoji mnogo primjera u kojemu se za frontend koristi Rust koji je inače sintaktički sličan C++, a napravljen je kako bi osigurao bolju memorijsku sigurnost, a da pritom održava visoke performanse. On je preveden pomoću Rusta kroz *rustwasm toolchain* kojeg Mozilla promovira. Uz Rust postoji opcija korištenja lanac alata za prevođenje datoteka u *asm.js* i WASM format koristeći LLVM koji omogućava korištenje C i C++. Pomoću ove opcije se Unity ubacio u web sektor s Godot Game Engineom i Unreal Engineom. Dodatna opcija je korištenje Blazor softverskog okvira koji je baziran na WASM-u, a koristi .NET, C# i HTML. Softverski okvir je *open souce* tipa, a uz zajednicu održava ga Microsoft[20]. Blazor je softverski okvir koji fokusiran na izradu jednostrane aplikacije (*engl.* single page applications). Jedan od jednostavnijih primjera koji je korišten za izradu aplikacije je kombinacija s Rusta uz Yew web softverski okvir. Yew je Rustov softverski okvir za stvaranje višenitnih frontend web aplikacija uz pomoć WASM-a. Sadrži softverski okvir koji je zasnovna na komponentama što olakšava stvaranje interaktivnog korisničkog sučelja, nešto slično već postojećim JavaScript softverskim okvirima tipa VueJS, Nuxt itd. Postiže izvrsne performanse minimiziranjem DOM API poziva i pomažući programerima da lako prebace obradu u pozadinske niti pomoću Web Workers API-ja. Web Workers omogućuje pokretanje naredbe skripte u pozadini niti koja je odvojena od glavne niti za izvršenje web aplikacije. Prednost je toga što se naporna obrada može izvesti u zasebnoj niti, omogućujući glavnoj, obično UI, niti da radi bez usporavanja. Yew podržava JavaScript interoperabilnost, omogućavajući programerima da iskoriste NPM pakete i integriraju se sa postojećim JavaScript aplikacijama. Obzirom da Yew koristi Rust, koji je vrlo popularan jezik i proglašen najdražim jezikom pet uzastopnih godina za redom na popularnoj stranici za pronalaženje rješenja programerskih problema StackOverflow, vrlo je efikasan i brz. Generalno frontend aplikacije funkcioniraju na način da se kod piše u C, C++ ili Rustu te se kroz IR obrađuju i pretvaraju u *.wasm* datoteke (***Slika 3.10.***).



Sl. 3.10. Shema pretvaranja C, C++ i Rusta u wasm datoteku

Unutar frontenda u moderno vrijeme još jedan jezik je došao do izražaja, a to je TypeScript. To je programski u kojemo se moraju navoditi tipovi podataka kao što su *string, int* i slično. Obzirom na postojanje i tog skriptnog poslužio je za primjer stvaranja AssemblyScripta. U osnovici to je JavaScript s tipovima WebAssembly-a koji su prevedeni statički u hrpu izvoza i uvoza što znači da se razlikuje od pokretanja dinamički pisanog JavaScripta[21]. Bez obzira na tu razliku, namjerno je pisan da bude vrlo sličan JavaScriptu i baš zato ima potencijal integriranja s postojećim konceptima web platforme za proizvodnju manjih WASM binarnih datoteka dok u isto vrijeme olakšava pisanje koda za one koji su upoznati s JavaScriptom i TypeScriptom.

3.5. Ugrađivanje u ne-web tehnologije

Iako je WebAssembly dizajniran za pokretanje na Webu, također je poželjno da se može dobro izvršavati u drugim okruženjima, uključujući sve, od minimalnih ljuski (*engl.* shellova) za testiranje do cjelovitih aplikacijskih okruženja, npr. na poslužiteljima u podatkovnim centrima, do IoT uređaja ili čak mobilnim/stolnim aplikacijama. Možda je čak poželjno izvršiti WASM ugrađen u veće programe. Ne-web okruženja mogu pružati različite API-je od onih u web okruženjima koja će testiranje značajki i dinamičko povezivanje učiniti vidljivim i korisnim. Ne-web okruženja

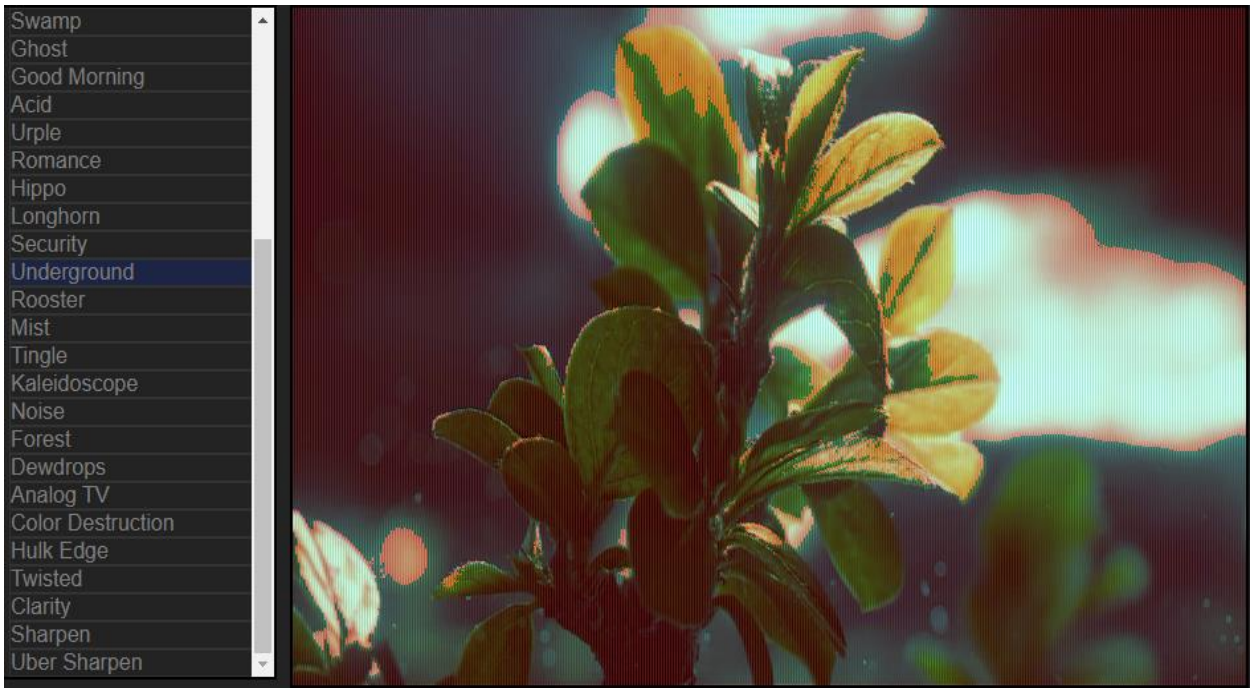
mogu uključivati JavaScript VM-ove, npr. Node.js, međutim WebAssembly je također dizajniran tako da može biti izveden bez prisutnog JavaScript VM-a. Sama specifikacija neće pokušati definirati bilo koju veliku prijenosnu biblioteku nalik *libc*-u. Međutim, određene značajke koje su jezgre semantike ovog jezika su slične funkcijama koje se nalaze u izvornom libc bile bi dio osnovne specifikacije kao što su primitivni operatori, npr. operator *grow_memory*, koji je sličan funkciji *sbrk* na mnogim sustavima i u budućnosti operatori slični *dlopenu*. Tamo gdje postoji preklapanje weba i popularnih ne-web okruženja mogu se predložiti zajedničke specifikacije, ali one bi bile odvojene od njegovih specifikacija. Simetrični primjer u JavaScript-u bila bi trenutna specifikacija *Loader*-a koja je predložena za Web i NodeJS okruženje i razlikuje se od JavaScript specifikacije. Međutim, u većini slučajeva očekuje se da će zajednice, kako bi postigle prenosivost na razini izvornog koda, graditi biblioteke koje su preslikane iz sučelja na razini izvora u ugrađene mogućnosti okruženja domaćina, bilo u vrijeme izrade ili u vrijeme izvođenja. On pružio sirove građevne blokove (*engl.* raw building blocks) kao što su: ispitivanje značajki, ugrađeni moduli i dinamičko učitavanje kako bi te biblioteke bile moguće. Dva rano očekivana primjera su POSIX i SDL. Općenito, ako se za ne-web tehnologije bude išlo u smjeru tako da mu nisu potrebni web API-ji, on se može koristiti kao prijenosni binarni format na mnogim platformama, donoseći velike prednosti u prenosivosti, alatima i jezičnoj agnostici (*engl.* language-agnosticity) obzirom da podržava C/C ++ semantike.

3.5.1. Video igre

Neki od primjera za korištenje WebAssemby-a kroz ne-web tehnologije su video igre, obrada slike, programi za crtanje i slično. Moderne 3D video igre zahtijevaju tonu procesorske snage kako bi izgledale dobro i brzo reagirale na unos igrača. Zbog toga je većina web-igara koje danas vidite u najboljem slučaju skinute verzije svojih računala ili konzola. Popularni primjeri za igrice koje koriste WASM su AngryBots i Game of life. AngryBots je Unity-eva igrica koja ga koristi i s njom su se kao proizvođači igrica priključili webu generalno, inače je 3D igra dostupna na: https://beta.unity3d.com/jonas/AngryBots/. Game of life je 2D igra u kojoj ne postoje igrači nego je kretanje u igrici ovisno o inicijalnom početnom stanju i ne zahtijeva unose za nastavak igre. Doom 3 je jedna vrlo popularnih igara koja se posebno istaknula kao postavljena na browser. Doom 3 je pucačina iz prvog lica horor igra koja je izvorno objavljena za Microsoft Windows 2004. Doom 3 koristi id Tech 4 mehanizmu za razvoj i upravljanje igre, objavljen pod GNU General Public License 2011[22]. Igra je imala kritičan i komercijalni uspjeh, s više prodanih više od 3,5 milijuna primjeraka igre što će dodatno popularizirati njegovo korištenje u ovom sektoru.

3.5.2. Obrada slike

Kako je WebAssembly otvorio vrata mnogim ne-web tehnologijama da se priključe webu tako se pridružila jedna od često korištenih grana u programerskom svijetu, a to je obrada medijskih podatak. Jedan od popularnih načina obrade digitalnih signala je korištenje procesora digitalnih signala (*engl.* digital signal processors). Kako bi to bilo omogućeno na vrlo jednostavan način koristi se WebDSP. WebDSP je kolekcija visokoučinkovitih algoritama, koji su dizajnirani za izgradnju blokova za web aplikacije koje imaju za cilj rad na medijskim podacima[23]. Metode su napisane u C ++ i sastavljene na OSM, a izložene kao jednostavne „vanilla JavaScript“ funkcije, *vanilla* žargonski znači izvorni JavaScript bez dodatnih paketa, koje programeri mogu pokretati na strani klijenta. Primjer obrade videa s određenim filterima i mogućnosti usporavanja i ubrzanja pokreta nalazi se na ovoj poveznici: https://d2jta7o2zej4pf.cloudfront.net/. Prvo što se na stranici vidi je video koji se obrađuje koristeći WASM, a ispod nje se nalazi rezultat obrade podataka koji koristi JavaScript. JavaScript ne uspije toliko dobro primijeniti filter na video. Rezultat za filter „Underground“ izgleda ovako:



Sl. 3.11. Obrada medijskih podataka koristeći WebDSP filterom Underground

3.5.3. Podatkovne znanosti

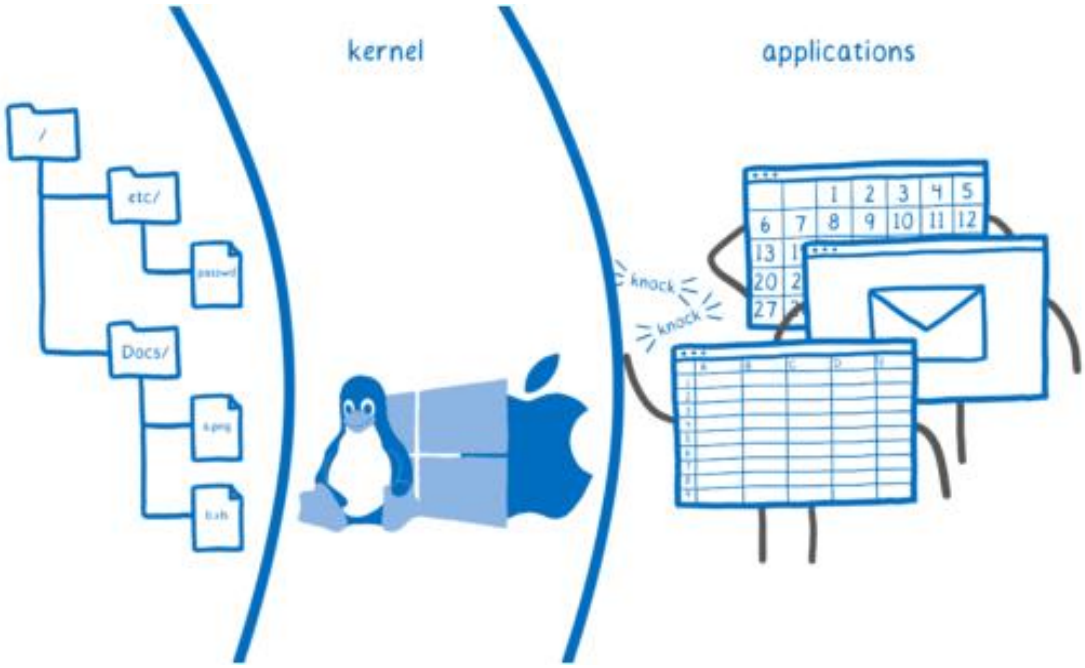
Podatkovne znanosti su se do sad najčešće koristile na računalu zbog JavaScriptove spore obrade veće veličine podataka. Neki primjeri toliko velikog broja podataka su slike iz aviona ili

Internacionalne Svemirske Postaje. Obzirom da se slike koriste i za druge razne stvari, ponekad se trebaju izvršiti dodatne funkcionalnosti na tim slikama, kao npr. detekcija objekata. Jedan od eksperimenata je bio učitavanje slike od 3GB u pregledniku, a to je slika koja se slikala iz aviona. Slika je poprilično velika zbog veće količine podataka, a kako bi bila kvalitetno prikazana koristile su se inkrementalne konične funkcije (*engl.* Incremental Conic Functions, skraćeno ICF), algoritam koji se koristi za rješavanja problema razvrstavanja na temelju matematičkog programiranja. Algoritam poboljšava prethodnu verziju konstrukcije klasifikatora koničnih funkcija kako bi se ubrzala računalna brzina. Inkrementalni korak izbjegava a-priori poznavanje broja podklasa, koji je potreban parametar u koraku sistematiziranja ovog klasifikacijskog algoritma. ICF je dokazano u prosjeku 3 puta brži od prethodnih verzija algoritama koji su koristili JavaScript bez žrtvovanja točnosti. Rezultati se nalaze na ovoj poveznici: http://deeplearning.magellium.fr/conference/2017/12/11/Datascience_from_the_browser_with_WebAssembly.html. Još jedna od većih prekretnica što je omogućeno s WebAssembly-em, a to je postavljanje AutoCAD-a na web kao web aplikacija, a ne više kao izvorna aplikacija. AutoCAD je vrlo stara, ali popularna i često korištena aplikacija za izradu tehničkih crteža. Koriste ga većinom inženjeri, a vrlo je jednostavan za učenje i uporabu. Aplikacija je prije dolaska na web bila vrlo zahtjevna i brzina crtanja i učitavanja je ovisila o hardveru što predstavlja velik problem za njegove korisnike. WASM je uz pomoć C-a i C++ riješio taj problem postavljanjem na webu. I dalje se može koristiti izvorna aplikacija, ali je preporučljivije za računalo kroz web preglednik.

3.5.4. Aplikacije bez servera

WebAssembly osim što je namijenjen da bi se omogućio pristup webu raznim programerskim sektorima i jezicima općenito, ne znači da je zatvorio vrata drugim sektorima. Može ga se koristiti i za izradu aplikacija bez servera. Kako bi to bilo moguće koristi se WebAssembly System Interface, skraćeno WASI. WASI je sučelje sustava koje je orijentiran sposobnošću (*engl.* capability-oriented) skup API-ja dizajnirani za standardizaciju izvršavanja zaštićenog okruženja WASM-ovih modula van preglednika[24]. Općenito sučelje sustava omogućava izradu, izmjenu podataka operacijskog sustava, za C programski jezik se smatra da već ima taj pristup, ali zapravo nema. Kada bi imao taj pristup problem bi bio taj što ako jedan program napravi izmjenu u nekim sistemskim objektima koje su ključne za održavanje cijelog sustava ili poremeti resurse drugog sustava, mogao bih srušiti cijeli sustav. Zbog tog problema stvoren je zaštitni prsten sigurnosti (*engl.* protection ring security). Uz prstenovu sigurnosnu zaštitu, operativni sustav u osnovi stavlja zaštitnu barijeru oko resursa sustava. Ovo je kernel.

Kernel je jedina stvar koja obavlja operacije kao što je stvaranje nove datoteke ili otvaranje datoteke ili otvaranje mrežne veze. Korisnički programi pokreću se izvan ovog kernela u načinu rada koji se naziva korisnički način rada. Ako program želi otvoriti datoteku, mora „pitati“ kernel da otvori datoteku za to. Ovdje nastupa koncept sistemskih poziva. Kada program treba zatražiti od kernela da napravi neku od takvih izmjena ili čitanja objekata operacijskog sustava, program to traži pomoću sistemskog poziva. U tom koraku kernel nastoji shvatit koji korisnik šalje zahtjev i tada može vidjeti ima li korisnik pristup datoteci prije otvaranja.



Sl. 3.12. Ilustracija zaštitnog prstena sigurnosti

Konkretno, WASI ima za cilj biti zajednički sloj koji će omogućavati WebAssembly modulima kako bi se mogli koristiti s domaćinovim izvornim okruženjima i dobiti granulirani pristup objektima operacijskog sustava kao što su datoteke, varijable okruženja ili utičnice (*engl.* sockets). NodeJS je prošle godine dodao eksperimentalnu podršku za WASI što znači da će se izvorno pokrenuti WASI instanca i pokrenuti modul u okruženju, a ujedno prosljeđivati objekti operacijskog sustava. Problem nastaje što ako svaki operacijski sustav ima svoje pozive, ali to je riješeno apstrakcijom. Većina programskih jezika nudi standardnu biblioteku, a tijekom kodiranja programer ne mora znati na koji će sustav djelovati jer koriste sučelje. Prilikom postavljanja lanac alata će birati implementaciju sučelja na temelju ciljanog sustava. Implementacija sučelja koristi funkcije API-ja operacijskog sustava da bude specifična za njega. Tu nastupa sučelje sustava primjer je *printf* naredba koja se prevodi za Windows i može se koristiti Windows API za

interakciju s uređajem. U slučaju prevođenja za Mac ili Linux koristit će se POSIX. Ovdje nastaje problem za WASM jer koristeći njega on ne zna kakav operacijski sustav je ciljan čak i kad se prevodi. Što znači da se niti jedno sučelje operacijskog sustava ne može koristiti unutar njegove standardne knjižnice. Zbog ovih razloga se koristi WASI. Razvojni programeri su za uklanjanje prethodno navedenih problema odlučili stvoriti modularni skup standardnih sučelja tako što će standardizirati najosnovniji modul, a to je *wasi-core*. *wasi-core* će sadržavat osnove koje su potrebne svim programima. Pokrivat će veći temeljni dio kao i POSIX, uključujući stvari poput datoteka, mrežnih veza, satova i slučajnih brojeva i za mnoge od ovih stvari bit će potreban vrlo sličan pristup POSIX-u. Na primjer, upotrijebljen je POSIX-ov pristup usmjeren na datoteke, gdje postoje sistemski pozivi kao što su otvaranje, zatvaranje, čitanje i pisanje i ostalo. Naravno da neće koristiti sve operacije kao u POSIX-u jer za korištenje ovakvog sučelja nisu potrebne i zato modularni pristup ovdje igra ulogu. Pomoću modularnog pristupa standardizirat će se veći dio stvari, a ostalim platformama će biti omogućeno koristiti iz sučelja ono što je njima potrebno.

4. ZAKLJUČAK

Web je tehnologija koju koristi danas gotovo svaki čovjek. Moguće je pristupiti web stranicama svih elektroničkih uređaja koji imaju pristup Internetu, od mobitela do tableta pa i osobnih računala, a u moderno vrijeme čak i automobili. WebAssembly je programski jezik koji je objavljen prije 6 godina što znači da ima još vremena za napredak, no obzirom za koliko je tehnologija otvorila vrata prema webu vrlo je perspektivna. Za istraživanje ovog rada i shvaćanje ovog jezika, bilo je prvo potrebno shvatiti što je Web općenito, njegovu povijest i evoluciju kroz vrijeme. Kada se pogleda kroz vrijeme unazad može se shvatiti zašto je web „morao“ nastati i što je pridonio svijetu. Komunikacija među ljudima je bila originalna namjena weba, ali su ljudi počeli uviđati da se može koristiti i za druge svrhe. Ljudi su stoga počeli marljivo raditi kako bi ga oblikovali i učinili boljim te primjenjivim za sve ljude. Naravno, za to je bio potreban razvoj računalne industrije kako bi cijena bila pristupačna prosječnim građanima. Kada je web postao pristupačan gotovo svim ljudima na svijetu želja je bila uvesti druge sektore kroz web jer je jednostavniji za koristiti i postaviti te neovisniji o sklopovlju, odnosno hardveru. Web je bio ograničen jezicima te se sve vrtilo oko HTML-a, CSS-a i JavaScripta. Nastupom WASM-a dogodilo se ono o čemu se razmišlja još od pretprošlog desetljeća, a to je omogućiti drugim programskim jezicima pristup webu poput C-u, C++, C#, Rustu, itd. Ovi jezici su bili značajno efikasniji od skriptnih jezika. Osnovna je razlika od početka bila u izvođenju i obrađivanju koda kako bi bio shvatljiv računalu. Nakon nekoliko godina od njegove službene objave postaje službeni jezik weba uz HTML, CSS i JavaScript. Kako su C, C++, Rust i ostali jezici visoke razine dobili pristup webu moguće ga je koristiti i za druge grane u kojima je bilo potrebno provoditi više računalne snage i brzina računanja operacija potrebnih za izvođenje koda. Neke od tih grana su video igre, strojno učenje, obrada slike i mnoge druge. WASM funkcionira tako što pretvara kod drugih jezika prvo u tekstualni format te sprema u datoteku *.wasm* datoteke koji se može iščitati i izvoditi u webu. Velika je prednost što će ga moći koristiti svatko tko gradi karijeru u IT industriji. Posebno će ljudi koji se bave programiranjem dobiti veći značaj jer će se ljudi koji prethodno nisu imali znanje o webu moći lako priključiti. JavaScript je dovoljno jak jezik da preživi, štoviše ova dva jezika su već u velikoj suradnji, a i on još uvijek nekim dijelom ovisi o JavaScriptu, ali je već na dobrom putu za „osamostaljenje“. Budući da se u modernom svijetu tehnologija brzo razvija i puno se radi na njoj zaključuje se kako je ovo vrlo perspektivan programski jezik koji još ima vremena za napredovanje, omogućit će bolju budućnost weba i olakšati ljudima pristup raznim funkcijama koje do sada nisu imali.

LITERATURA

[1] CERN, „A short history of the Web“: „Where the Web was born“, CERN, CERN, dostupno na: https://home.cern/science/computing/birth-web/short-history-web (12.6.2021.)

[2] C.B., Bogh, „The Evolution of Web Technologies“: eploy, 2012., dostupno na: https://www.eploy.co.uk/about-eploy/theblog/may-2012/the-evolution-of-web-technologies/ (8.6.2021.)

[3] W.F., Fanguy, „A guide to Different Types of Website Structure“: „Types of website structures“, Adobe, 2020., dostupno na: https://xd.adobe.com/ideas/process/information-architecture/different-types-of-website-structures/ (12.6.2021.)

[4] Bitesize, „Structures and links (website)“: „Webiste structure“, BBC, Ujedinjeno Kraljevstvo, dostupno na: https://www.bbc.co.uk/bitesize/guides/z96psbk/revision/2 (12.6.2021.)

[5] CARNet, „Uvod u HTML“: „Što je HTML?“, CARNet, Republika Hrvatska, dostupno na: https://tesla.carnet.hr/mod/book/view.php?id=5430&chapterid=885 (12.6.2021.)

[6] R.M i V.B., Morin i Brown, „Scripting Languages“: „A Cross-OS Perspective“, Mactech, dostupno na: http://preserve.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html (12.6.2021.)

[7] Techopedia, „Interpreter“: „Techopedia Explains Interpreter“, Techopedia, dostupno na: https://www.techopedia.com/definition/7793/interpreter (12.6.2021.)

[8] Steemit, „WHAT IS THE DIFFERNCE BETWEEN COMPILER AND INTERPRETER ?“: „What is the difference? Which one should we use?“, Steemit, 2018., dostupno na: https://steemit.com/busy/@koroglu00071/what-is-the-difference-between-compiler-and-interpeter (12.6.2021.)

[9] A.R., Rossberg, „WebAssembly Specification“: „Introduction“, WebAssembly, 2021., dostupno na: https://webassembly.org/ (19.6.2021.)

[10] E.E., Elliott, „What is WebAssembly“: „What Exactly is WebAssembly“, Medium, 2015. dostupno na: https://medium.com/javascript-scene/what-is-webassembly-the-dawn-of-a-new-era-61256ec5a8f6 (19.6.2021.)

[11] MDN Contributors, MDN Web Docs, „WebAssembly, „In a Nutshell“, MDN Web Docs, 2017., dostupno na: https://developer.mozilla.org/en-US/docs/WebAssembly (19.6.2021.)

[12] L.W., Wagner, Mozilla Hacks, „A WebAssembly Milestone: Experimental Support in Multiple Browsers“: „Experimenting with WebAssembly“, Mozilla Hacks, 2016., dostupno na: https://hacks.mozilla.org/2016/03/a-webassembly-milestone/ (19.6.2021.)

[13] Techopedia, „Runtime Envorinment (RTE)“: „What Does Runtime Environment (RTE) Mean?“, Techopedia, dostupno na https://www.techopedia.com/definition/5466/runtime-environment-rte (19.6.2021.)

[14] H.A, Ahn, „Bulk Memory Operations and Conditional Segment Initialization“: „Bulk Memory Operations“, 2021., GitHub, dostupno na: https://github.com/WebAssembly/bulk-memory-operations/blob/master/proposals/bulk-memory-operations/Overview.md (26.6.2021.)

[15] B.S., Smith, „Import/Export mutable globals proposals“: „Proposed Solution: Import and Export Mutable Globals“, 2018., Github, dostupno na: https://github.com/WebAssembly/mutable-global/blob/master/proposals/mutable-global/Overview.md (26.6.2021.)

[16] A.R., S.R., I.S., H.A., Rossberg, Rubanov, Szmozsánszky, Ahn, „Reference Types for WebAssembly“: „Introduction“, Github, 2020., dostupno na: https://github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md (26.6.2021.)

[17] D.G., H.A., Gohman, Heejin, „Non-trapping Float-to-int Conversion“: „ Introduction“, Github, 2020., dostupno na: https://github.com/WebAssembly/nontrapping-float-to-int-conversions/blob/master/proposals/nontrapping-float-to-int-conversion/Overview.md (26.6.2021.)

[18] V8 Dev, „Fast, paralel applications with WebAssembly SIMD“: „WebAssembly SIMD proposal“, Github, 2020., dostupno na: https://v8.dev/features/simd (26.6.2021.)

[19] W.C., Chen, „Performance Testing Web Assembly vs JavaScript“: „The Experiment“, Medium, 2018., dostupno na: https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875 (3.7.2021.)

[20] L.L., R.A., D.B., D.N.V, Nhu Vy, N.T.M., S.T., T.P., Petersen i A.B.B, Latham, Anderson, Berry, Mullen, Addie i Bhargav: „Introduction to APS.NET Core Blazor“: „Blazor

WebAssembly“, Microsoft, 2020., dostupno na: https://docs.microsoft.com/en-gb/aspnet/core/blazor/?view=aspnetcore-5.0 (4.7.2021.)

[21] The AssemblyScript Project, „Getting Started“: „Introduction“, The AssemblyScript Project, 2021., dostupno na: https://www.assemblyscript.org/introduction.html (4.7.2021.)

[22] B.C., Couriol, „Iconic Doom3 Game Now in Brosers with WebAssembly: Q&A with Gabriel Cuvillier“: InfoQ, 2019., dostupno na: https://www.infoq.com/news/2019/07/doom3-web-assembly-port/ (4.7.2021.)

[23] M.W., D.P., S.K., Wagner, Pulusani, Khalkhali, „A client-side DSP library utilizing the power of WebAssembly (.wasm)“: Github, 2017. dostupno na: https://github.com/shamadee/web-dsp (4.7.2021.)

[24] L.C., Clark, „Standardizing WASI: A system interface to run WebAssembly utside the web“: „What's a system interface?“, Mozzila Hacks, 2019., dostupno na: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (4.7.2021.)

SAŽETAK

U diplomskom radu obrađen je WebAssembly standard, odnosno jezik koji omogućava pristup drugim programskim jezicima i korištenje weba na totalno drugoj razini. Istražene su razne grane i sektori na koje će uspješno djelovati i na koje je već djelovalo. Za konkretan primjer ovog rada izvršen je eksperiment usporedbe brzine izvođenja računskih operacija za kod u WebAssembly-u i JavaScriptu. Budući da je tehnologija još uvijek vrlo mlada, ali ima izvanredne rezultate, mnogi IT stručnjaci vjeruju da će se vrlo brzo razvijati i da će se početi koristiti na mreži više u neke druge svrhe, osim u komercijalne i društvene.

Ključne riječi: JavaScript, JS, WASI, WASM, Web, WebAssembly

ABSTRACT

Future of the Web: WebAssembly

In this thesis the WebAssembly standard, i.e. programming lanuage was expolred, and it allows access to other programming languages and the use of the web at a completely different level. Various branches and sectors were explored in which WebAssembly will leave it's mark and have influence and on which it has already influenced. Concrete example for this paper an experiment is performed to compare the speed of performing computational operations for code in WebAssembly and JavaScript. Since technology is still very young but has remarkable results many IT professionals believe that it will evolve very quickly and it will begin to be used online more for some purposes besides commercial and social ones.

Key words: JavaScript, JS, WASI, WASM, Web, WebAssembly

ŽIVOTOPIS

Antonio Bajivić je rođen 11. srpnja 1997. godine u Virovitici. Od 2004. do 2012. godine pohađa Osnovnu školu Vladimira Nazora u Virovitici. Za vrijeme osnovnoškolskog obrazovanja bio je sudionik natjecanja iz geografije i matematike. Godine 2012. upisuje Opću gimnaziju Petra Preradovića u Virovitici koju završava u 2016. godine polaganjem ispita državne mature. Tijekom svojeg srednjoškolskog obrazovanja sudjeluje na natjecanjima iz geografije, matematike, latinskog jezika. Godine 2016. upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija na Sveučilištu Josipa Jurja Strossmayera u Osijeku, preddiplomski studij Računarstvo. Godine 2019. upisuje diplomski studij, smjer Računalno Inženjerstvo.

_____

Potpis autora

PRILOZI

Prilog 1: Završni rad u docx i pdf formatu

Prilog 2: Programske datoteke iz eksperimentalne usporedbe između WebAssembly-a i JavaScripta