Roosa Laukkanen

WebAssembly ja sen käyttäminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

15.5.2025

Tiivistelmä

Tekijä:	Roosa Laukkanen
Otsikko:	WebAssembly ja sen käyttäminen
Sivumäärä:	31 sivua
Aika:	15.5.2025
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaaja:	Lehtori Vesa Ollikainen

Opinnäytetyön tarkoituksena oli perehtyä WebAssemblyyn ja sen käyttämiseen web-sovelluskehityksessä. Työssä käsiteltiin WebAssemblya edeltäviä teknologioita ja niiden vaikutusta sen kehittämiseen. WebAssemblyn ollessa käännöskohde työssä selvitettiin C/C++-kielisen lähdekoodin WebAssemblyksi kääntämisen prosessi ja kääntämiseen tarvittavat työkalut. Lisäksi opinnäytetyössä selvitettiin WebAssembly-koodin rakennetta ja sen ominaisuuksia.

Työn tavoitteena oli saada kattava yleiskuva WebAssemblysta ja sen käyttämisestä, sekä auttaa arvioimaan WebAssemblyn soveltuvuutta omiin projekteihin. Opinnäytetyön tuloksena toteutettiin erilaisia koodiesimerkkejä C-kielisestä lähdekoodista käännetyn WebAssembly-koodin käyttämiseen. Esimerkeissä käytettiin sekä Emscripten-käännöstyökalun tarjoamia ccall()- ja cwrap()-funktioita että WebAssembly-moduulin striimausta C-funktioiden kutsumiseen. Näiden lisäksi tehtiin vielä erillinen esimerkki merkkijonojen välittämisestä WebAssemblyn avulla.

Tämän opinnäytetyön tekemisessä on käytetty apuvälineenä OpenAI:n ChatGPT:n versiota GPT-4. Tekoälyä on käytetty aiheen sisällön ideoinnissa, lähdemateriaalin suomentamisessa, tiivistelmän englanninkielisessä käännöksessä ja koodiesimerkeissä. Tekoälyä ei ole käytetty tekstin tai kuvien tuottamisessa. Opinnäytetyön tekijänä olen vastuussa kaikesta opinnäytetyöni sisällöstä.

Avainsanat: WebAssembly, Emscripten, C/C++, web-ohjelmointi

Abstract

Author:	Roosa Laukkanen
Title:	WebAssembly and Its Use
Number of Pages:	31 pages
Date:	15 May 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisor:	Vesa Ollikainen, Senior Lecturer

The purpose of the study was to explore WebAssembly and its use in web application development. The thesis addresses the technologies that preceded WebAssembly and their influence on its development. With WebAssembly as a compilation target for programming languages, the study examined the process of compiling C/C++ source code into WebAssembly and the tools required for this compilation. Additionally, the structure of WebAssembly code and its features were investigated.

The goal was to provide a comprehensive overview of WebAssembly and its use, as well as to assist in evaluating the suitability of WebAssembly for personal projects. Various code examples were created to demonstrate the use of WebAssembly code compiled from C source code. These examples utilized both the ccall() and cwrap() functions offered by the Emscripten toolchain, as well as WebAssembly module streaming for calling C functions. In addition, a separate example was created for passing strings using WebAssembly.

This study utilized OpenAI's ChatGPT version GPT-4 as a support tool. AI was used for brainstorming content related to the topic, translating source material into Finnish and the original abstract into English, and creating code examples. However, AI was not used to produce text or images for the thesis.

Keywords: WebAssembly, Emscripten, C/C++, web development

Sisällys

Lyhenteet

API: *Application Programming Interface.* Ohjelmointirajapinta, joka mahdollistaa eri ohjelmistojen välisen kommunikaation.

AST: *Abstract Syntax Tree*. Abstrakti syntaksipuu. Käytetään edustamaan ohjelman tai koodin rakennetta.

CSS: *Cascading Style Sheets*. Tyylikieli, jolla määritetään verkkosivujen ulkoasua.

DOM: *Document Object Model*. Käytetään esim. HTML-dokumentin esittämiseen loogisena puumaisena rakenteena.

HTML: *Hypertext Markup Language*. Hypertekstin merkintäkieli, jota käytetään verkkosivujen rakenteen ja sisällön määrittämiseen.

JIT: *Just-In-Time*. Nykyaikaisten JavaScript-moottoreiden käyttämä käännöstekniikka, jossa ohjelmakoodi käännetään konekieleksi suorituksen aikana.

NaCl: *Native Client.* Hiekkalaatikkoteknologia, jolla C/C++-koodia voidaan ajaa turvallisesti verkkoselaimessa.

PNaCl: *Portable Native Client.* Siirrettävyyden osalta paranneltu versio Native Clientista (NaCl).

Wasm: *WebAssembly*. Lyhenne sanasta WebAssembly.

WAT: *WebAssembly Text Format*. WebAssembly-koodin tekstimuotoinen esitys.

1 Johdanto

Web-sovellukset ovat kehittyneet valtavasti viime vuosikymmenten aikana. Alun perin yksinkertaisista HTML-pohjaisista sivustoista on tullut monimutkaisia sovelluksia, jotka tarjoavat lähes työpöytäsovellusten kaltaisia käyttökokemuksia. Vaikka web-sovelluskehityksen perusta eli HTML, CSS ja erityisesti JavaScript ovat mahdollistaneet tämän kehityksen, ne eivät aina riitä erityisen suurta suorituskykyä vaativien sovellusten, kuten 3D-pelien tai kuva- ja videoeditorien tarpeisiin.

Vuonna 2017 julkaistu ja vuonna 2019 neljänneksi kieleksi HTML:n, CSS:n ja JavaScriptin ohelle julistettu WebAssembly tarjoaa ratkaisun suorituskykykriittisille web-sovelluksille mahdollistamalla koodin suorittamisen selaimessa lähes natiivilla nopeudella. Tämän opinnäytetyön tarkoituksena on käsitellä WebAssemblyn ominaisuuksia ja sen historiaa sekä sitä, mikä WebAssemblyn rooli on web-sovelluskehityksessä ja miten se vertautuu JavaScriptiin. Lisäksi WebAssemblyn käyttämiseen perehdytään yksinkertaisten esimerkkien avulla.

WebAssembly mahdollistaa useiden web-sovelluskehitykselle epätyypillisten kielten käyttämisen, ja tämä opinnäytetyö rajautuukin koskemaan WebAssemblyn käyttämistä C/C++-ohjelmointikielten osalta. Opinnäytetyön tavoitteena on toimia eräänlaisena perehdytyksenä WebAssemblyyn ja sen käyttämiseen ja samalla auttaa työn lukeneita arvioimaan WebAssemblyn soveltuvuutta mahdollisiin omiin projekteihin. Opinnäytetyö toimii myös uuden oppimisena minulle itselleni, sillä ensimmäisen kerran tutustuin WebAssemblyyn vasta tämän opinnäytetyön myötä.

2 Lähtökohdat WebAssemblylle

Tässä luvussa käsitellään WebAssemblya edeltäviä teknologioita ja niiden vaikutusta WebAssemblyn kehittämiseen. Lisäksi luvussa esitellään tiivistetysti web-sovelluskehityksen perusteet painottuen erityisesti JavaScriptiin.

2.1 WebAssemblyn kehittäminen ja sitä edeltävät teknologiat

WebAssembly, josta käytetään myös lyhennystä Wasm, on web-sovellusten kehittämisessä käytettävä matalan tason binäärimuotoinen käskyformaatti (engl. *instruction format*). Toisin kuin yleisesti web-sovelluskehityksessä käytettävä JavaScript, WebAssembly ei itsessään ole ohjelmointikieli, vaan pikemminkin käännöskohde (engl. *compilation target*), joksi lähdekoodi käännetään. WebAssemblyn tarkoituksena on mahdollistaa web-sovellusympäristössä epätyypillisillä korkean tason ohjelmointikielillä, kuten C:llä, C++:lla ja Rustilla, kirjoitetun koodin ajamisen web-selaimissa. [WebAssembly Concepts 2025.]

WebAssembly kehitettiin ratkaisuksi JavaScript-ohjelmointikielen haasteisiin suurta laskentatehoa vaativien sovellusten suhteen. Ennen WebAssemblya käytössä oli kuitenkin useita eri teknologioita, jotka pyrkivät nopeuttamaan JavaScriptiä ja tuomaan sen suorituskykyä lähemmäs natiivin konekoodin nopeutta. Ensimmäiset askeleet kohti nopeampaa JavaScriptiä otettiin, kun aiemmin hitaaksi koettu koodirivi kerrallaan tulkattu JavaScript sai vuoteen 2008 mennessä kolme sen toimintaa nopeuttavaa JIT (Just-In-Time) -kääntäjän sisältävää JavaScript-moottoria. Nämä olivat vuonna 2008 julkaistun Chrome-selaimen mukana tullut V8-JavaScript-moottori sekä Firefoxin TraceMonkey ja Safarin SquirrelFish Extreme. [Zakai 2020.] Tehokkaammat JavaScript-moottorit toivat pelkkään tulkkaamiseen perustuvan suorituksen lisäksi myös mahdollisuuden koodin kääntämiseen nopeammaksi konekoodiksi sen suorittamisen aikana [O. 2022].

Näistä edistyksistä huolimatta heräsi epäilyksiä siitä, yltäisikö JavaScript yksinään natiivin konekielen suoritusnopeuteen tai riittäisikö se raskaampaa laskentatehoa vaativien sovellusten, kuten pelien tai kuvankäsittelyohjelmien toteutukseen. Seuraava askel lähemmäksi WebAssemblyn kehittämistä otettiin Googlen vuonna 2008 julkaiseman Native Client (NaCl) -teknologian myötä. [Zakai 2020.] NaCl-teknologia mahdollistaa käännetyn C- ja C++-koodin ajamisen web-selaimessa turvallisesti erillisessä hiekkalaatikossa (engl. *sandbox*) [Introduction to Portable… n.d.; Native Client 2024]. NaCl-teknologian hiekkalaatikko

koostuu kahdesta suojaavasta kerroksesta, joiden avulla epäluotettavaa konekoodia voidaan ajaa selaimessa turvallisesti ilman riskiä käyttäjän laitteen saastumisesta. Hiekkalaatikon ensimmäinen kerros käsittää sovelluksen koodin ja toinen itse NaCl-ajoympäristön. Koodi pystyy kommunikoimaan vain NaCl-ajoympäristön kanssa, eikä se siten pääse suoraan käsiksi laitteen muihin resursseihin. NaCl:n ongelmaksi muodostui kuitenkin sen siirrettävyys. Teknologian toimiminen perustuu koodin käännökseen erikseen aina tietylle suoritinarkkitehtuurille, esim. x86 tai ARM, sopivaksi, eivätkä sen toteutukset siten toimi sellaisenaan kaikilla laitteilla. [Zakai 2020.]

Vuonna 2010 ratkaisuksi NaCl-teknologian siirrettävyysongelmiin Google julkaisi Portable Native Client (PNaCl) -teknologian, joka on paranneltu versio Native Clientista [Zakai 2020]. PNaCl tukee NaCl-teknologian tapaan mm. C- ja C++-kielten kääntämistä, mutta teknologiat eroavat siinä, missä vaiheessa käännös natiiviksi konekoodiksi tapahtuu. Kun NaCl:a käytettäessä vastuu ohjelman rakentamisesta, testaamisesta ja käyttöönotosta kaikille tuetuille laitealustoille jää kehittäjälle, PNaCl jättää vastuun käännöksestä asiakaspuolelle. [Donovan ym. 2010.] NaCl:a käytettäessä lähdekoodi on siis käännettävä etukäteen suoritinarkkitehtuurikohtaisesti, mutta PNaCl kääntää koodin LLVM IR -välikoodiksi (Low-Level Virtual Machine Intermediate Representation) ja lopullinen käännös tapahtuu vasta asiakaspuolella selaimessa. LLVM-tyylinen tavukoodi voidaan säilöä palvelimelle siirrettävänä suoritettavana tiedostona tavallisen verkkosivuston resurssin tapaan. Käyttäjän vieraillessa sivustolla tiedosto ladataan ja lopullinen käännös suoritetaan asiakaspuolella selaimessa suoraan käyttäjän laitteen suoritinarkkitehtuurin mukaisesti. Asiakaspuolella kääntäminen ja koodin ajaminen tapahtuvat samalla tavalla kuin NaCl-teknologiassa käyttäjän laitetta suojaavana hiekkalaatikkototeutuksena. [Sehr 2013; Zakai 2020.]

NaCl-teknologia oli käytössä alun perin vain selainlaajennuksena, mutta myöhemmin se integroitiin osaksi Googlen Chrome-selainta. NaCl-teknologia kohtasi kuitenkin vastustusta muilta selainkehittäjiltä, erityisesti Mozillalta ja Operalta, eikä se koskaan saanut laajaa kannatusta. Vuodesta 2011 eteenpäin

NaCl-teknologian käyttö rajoittui pelkästään Chrome-selaimeen ja sen sovelluksiin. [Zakai 2020.]

Mozillan vaihtoehtoinen kehityslinja kohti konekielen nopeutta sai pohjaa vuonna 2010, kun Emscripten-käännöstyökalun kehittäminen aloitettiin. Emscriptenin alkuperäinen toimintamalli oli kääntää C++-lähdekoodia JavaScriptiksi LLVM-välikoodia hyödyntäen. Käännöstyökaluja JavaScriptille oli tehty aiemminkin esimerkiksi Java-ohjelmointikielen kääntäjänä, mutta ei niinkään C++-kielen. Emscripten ei itsessään nopeuttanut JavaScriptin toimintaa, mutta mahdollisti C++-kielisten ohjelmien kääntämisen JavaScriptiksi. Kehitys nojasi pitkään siihen, että JavaScript-koodin suoritusnopeutta saataisiin kehitettyä lopulta lähelle konekoodin nopeutta. Emscriptenin lisäksi kehityksen alla oli vuodesta 2013 alkaen myös asm.js. [Zakai 2020.]

Asm.js on Mozillan kehittämä muodollisesti määritelty, täsmällinen JavaScriptin osajoukko, jota voidaan käyttää matalan tason tehokkaana käännöskohteena muille ohjelmointikielille. Se on myös JavaScriptin alikieli ja juuri siksi sen etuna onkin sen suorittamisen mahdollisuus kaikilla selaimilla. Vaikka asm.js-koodi muistuttaa C-koodia monin tavoin, on se silti validia JavaScriptiä. Suorituskyvyltään asm.js on parhaimmillaan kuitenkin lähempänä natiivia koodia kuin tavallista JavaScriptiä. Asm.js oli ensimmäinen askel kohti suoritinarkkitehtuuririippumatonta assembly-koodia ja se kuvaa hiekkalaatikossa toimivaa virtuaalikonetta ohjelmointikielille, kuten C:lle ja C++:lle. Lähdekoodin kääntämiseen asm.js-koodiksi käytettiin Emscripteniä, joka nykyään tukee WebAssemblya. Yksi asm.js:n merkittävä etu on sen ennenaikaisen kääntämisen (engl. *ahead-of-time compilation*, AOT) mahdollisuus. AOT-kääntäminen tarkoittaa koodin kääntämistä tehokkaaksi konekieleksi ennen sen suorittamista. Kuitenkaan kaikki selaimet eivät tukeneet AOT-kääntämistä ja kyseisissä selaimissa asm.js suoritettiin tavallisen JavaScriptin tavoin. [Asm.js 2024; From asm.js to… 2024; Steiner 2023.]

Googlen NaCl sekä Mozillan asm.js ja Emscripten olivat aluksi kilpailevia teknologioita. Google ja Mozilla päätyivät kuitenkin tekemään yhteistyötä ja tuloksena

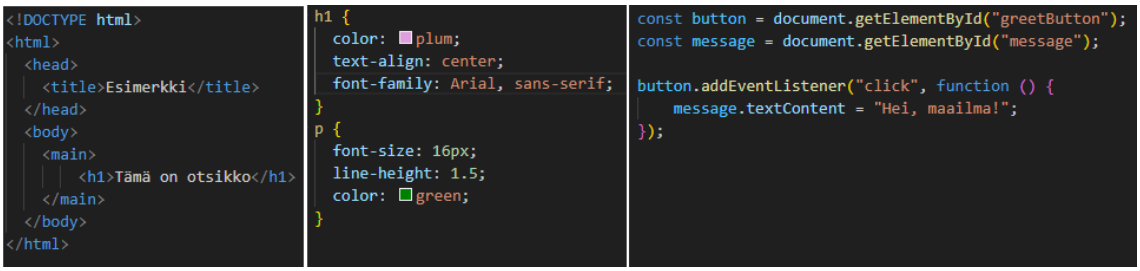
syntyi WebAssembly, joka uusien teknologioiden lisäksi yhdisteli aiempien toteutusten hyväksi havaittuja teknologioita. NaCl-teknologian tavoin WebAssembly ei perustu JavaScriptiin ja sen toteutus on binäärimuotoista ja asm.js-teknologian tavoin WebAssembly suoritetaan samassa prosessissa JavaScriptin kanssa. WebAssemblyn kääntäjätyökalun kehityksessä on käytetty pohjana Emscripteniä. [Zakai 2020.]

WebAssembly julkistettiin ensimmäisen kerran vuonna 2015, ja tällöin käytettävissä oli vielä kehitysvaiheen versio. Ensimmäinen pienin julkaistavissa oleva tuoteversio eli MVP-versio (Minimum Viable Product) julkaistiin vuonna 2017. Samalla kaikki tunnetuimmat selaimet (Chrome, Firefox, Safari ja Edge) alkoivat tukea sen käyttämistä. WebAssemblyn yleistymisen myötä aiemmin käytössä olleiden NaCl-, PNaCl- ja asm.js-teknologioiden tuki lakkautettiin. WebAssemblyn kehityksestä vastasivat aluksi pääosin Google ja Mozilla, mutta myöhemmin mukaan kehitykseen liittyivät myös Apple ja Microsoft. [Zakai 2020.] Vuonna 2019 web-standardeja ja suosituksia ylläpitävä ja kehittävä World Wide Web Consortium (W3C) julisti WebAssemblyn virallisesti suosittelemiensa web-standardien joukkoon. Samalla siitä tuli JavaScriptin, HTML:n ja CSS:n ohella neljäs web-selaimessa koodin ajamisen mahdollistava kieli. [W3C 2019.]

2.2 JavaScript ja web-sovelluskehityksen perusteet

Jotta voisi täysin ymmärtää WebAssemblyn käyttötarkoituksen ja syyt sen kehittämiselle, on hyvä perehtyä hieman web-sovelluskehityksen perusteisiin erityisesti JavaScriptin osalta. Alun perin vuonna 1995 julkaistu JavaScript on yksi nykypäivän käytetyimmistä ohjelmointikielistä, ja sen rooli web-sovelluskehityksessä on merkittävä. Yhdessä HTML:n ja CSS:n kanssa JavaScript muodostaa web-kehityksen perustan. [JavaScript – Overview n.d.; Introduction 2025.] Lyhyesti esiteltynä HTML (HyperText Markup Language) on merkintäkieli, jota käytetään verkkosivujen ja muun verkkosisällön rakenteen määrittämiseen eli sillä voidaan määrittää ja järjestää esimerkiksi otsikoita, kappaleita, kuvia ja videoita. CSS (Cascading Style Sheets) on puolestaan tyylikieli, jolla määritetään verkkosivujen ulkoasu esimerkiksi taustavärien ja fonttien muodossa. JavaScriptin

osuus tässä kaikessa on tuoda verkkosivuille dynaamisuutta ja toiminnallisuutta, ja ilman sitä verkkosivut olisivatkin täysin staattisia. [Introduction 2025.] Kuvassa 1 on esitetty esimerkkikoodia HTML:stä, CSS:stä ja JavaScriptistä kielten rakenteiden eroavaisuuksien havainnollistamiseksi.

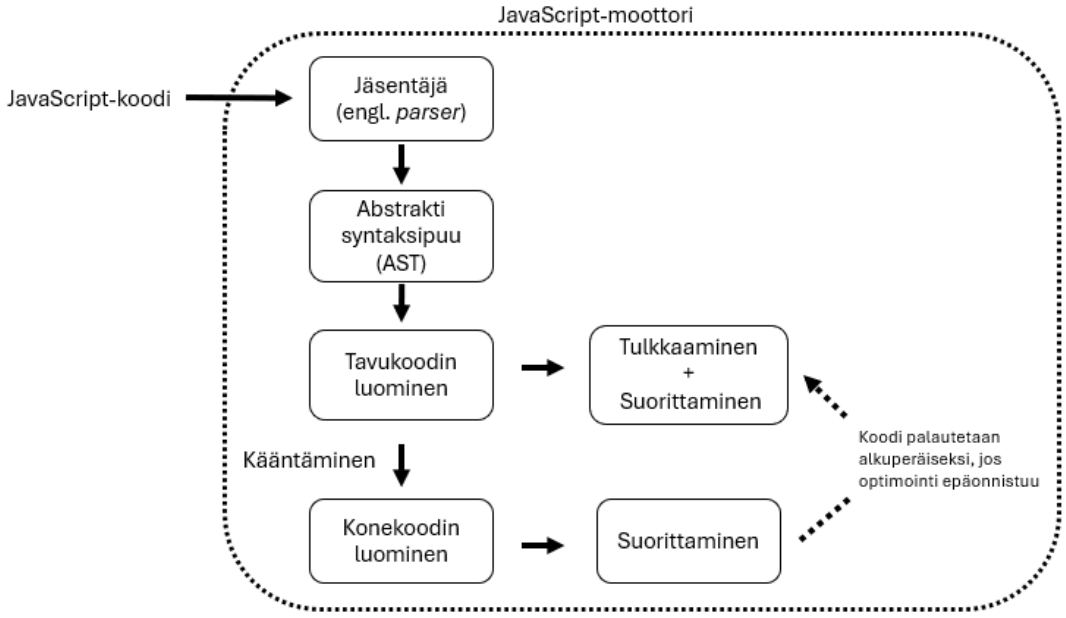


Kuva 1. HTML:n, CSS:n ja JavaScriptin eroavaisuudet koodin rakenteissa.

JavaScript on tulkattava olio-ohjelmointia tukeva kieli, jota voidaan käyttää sekä asiakaspuolella web-selaimessa (frontend) että palvelinpuolella Node.js-ajoympäristön avulla (backend). JavaScriptissä ei käytetä vahvaa tyypitystä eli muuttujia ei luoda esimerkiksi Integer- tai String-tyyppisinä, kuten vaikkapa Java-ohjelmointikielessä. [Introduction 2025.] Dynaaminen tyypitys mahdollistaa sen, ettei muuttujan tietotyyppiä tarvitse määrittää sitä luotaessa ja tietotyyppi muunnetaan automaattisesti tarpeen mukaan ohjelman suorituksen aikana. Saman muuttujan arvo voi siis ohjelman suoritusvaiheen mukaan olla esimerkiksi joko numeraalinen tai tekstimuotoinen. [Grammar and Types 2025.]

JavaScript-koodi on muodoltaan helppolukuista ihmisen kirjoitettavissa olevaa koodia. Se on tulkattava kieli eli web-selaimen JavaScript-moottori pystyy suorittamaan JavaScript-koodia ilman, että sitä tarvitsee kääntää ensin konekoodiksi. Selain lataa JavaScript-koodin joko HTML-tiedoston <script>-elementistä tai erillisestä JavaScript-tiedostosta, minkä jälkeen moottori lukee ja jäsentää (engl. *parse*) koodin muuntaen sen abstraktiksi syntaksipuuksi (Abstract Syntax Tree, AST). Abstraktiin syntaksipuuhun kootaan koodin merkitykselliset osat, kuten funktiot ja muuttujat. [O. 2022.] Nykyaikaiset JavaScript-moottorit eivät suorita koodia suoraan abstraktista syntaksipuusta, vaan muuntavat sen tavukoodiksi (engl. *bytecode*) [Padmanabhan 2024].

Nykyaikaiset web-selaimet tukevat tulkkaamisen ja kääntämisen yhdistävää JIT-kääntämistä. JIT-kääntämisellä tarkoitetaan sitä, että JavaScript-moottori kääntää tarvittaessa JavaScript-koodista luotua tavukoodia konekoodiksi, jota tietokoneen suoritin pystyy käsittelemään huomattavasti selainta nopeammin. JIT-kääntäjä analysoi jatkuvasti koodia sen suorittamisen aikana ja pyrkii tunnistamaan koodista usein suoritettavia osia (engl. *hot spots*). Mikäli kääntämisestä aiheutuva laskennallinen kustannus on pienempi kuin suorituskyvyn parannus, koodi käännetään konekoodiksi. Koodin kääntäminen konekoodiksi tapahtuu samaan aikaan koodin suorittamisen kanssa ja tästä tuleekin ajonaikaisen kääntämisen nimi Just-In-Time. [Just-In-Time Compilation 2024]. Mikäli koodin optimointi tavukoodista konekoodiksi ei onnistu, kääntäjä palauttaa koodin takaisin normaalisti suoritettavaksi tavukoodiksi [Pattanayak 2024]. JIT-kääntäjällä toimivan JavaScript-moottorin toiminta on havainnollistettu kuvassa 2.

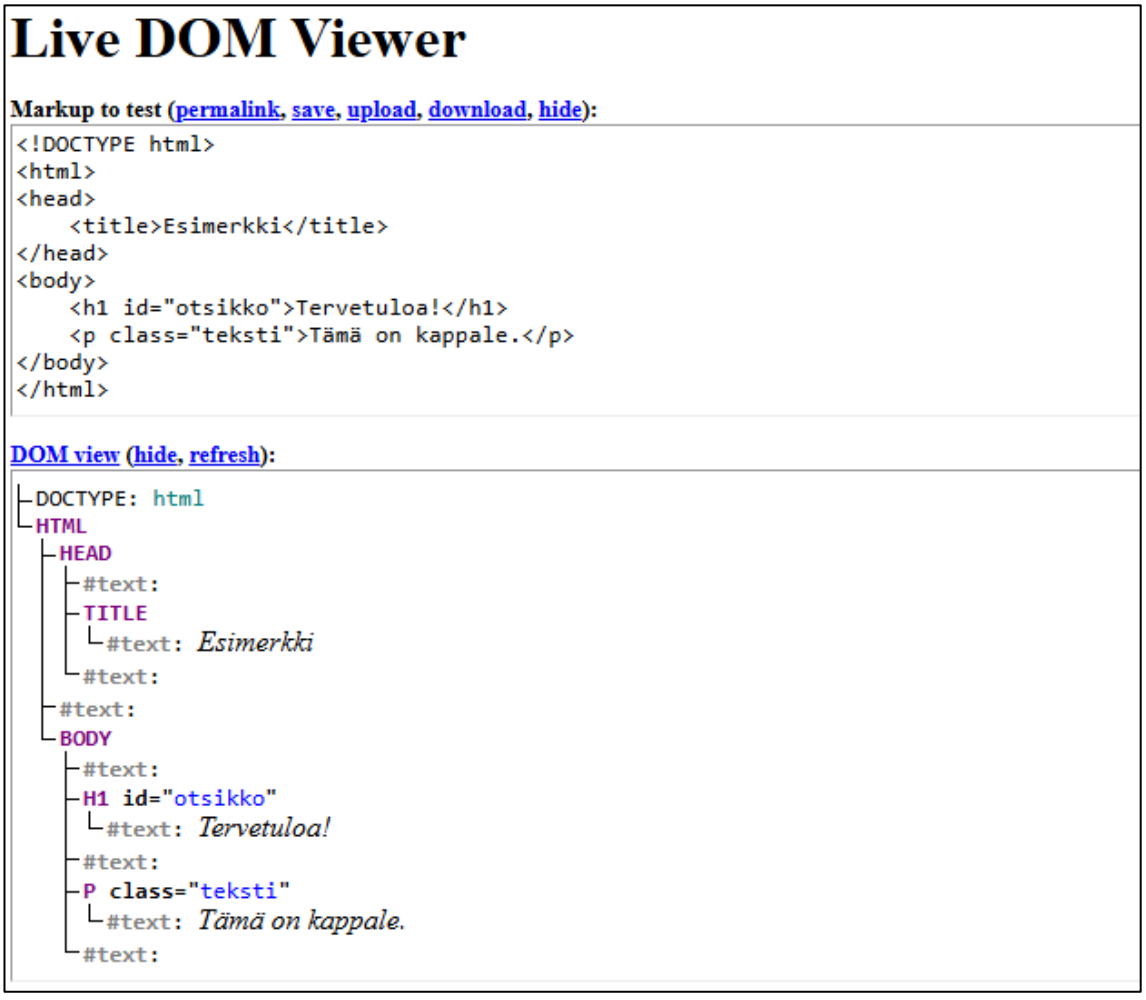


Kuva 2. JIT-kääntämistä tukevan JavaScript-moottorin toiminta [Pattanayak 2024].

JavaScriptin yksi tärkeimmistä ominaisuuksista dynaamisten verkkosivujen luomisessa on sen kyky käyttää ohjelmointirajapintoja (Application Programming Interface, API). Ohjelmointirajapinnat eivät sisälly JavaScript-moottoriin, vaan

ovat osa web-selaimen tarjoamaa ajonaikaista ympäristöä (engl. *runtime environment*). [Croad 2020.] Ohjelmointirajapintoja löytyy sekä web-selaimista sisäänrakennettuina että kolmansien osapuolten tarjoamina. Selainten tarjoamista rajapinnoista yksi merkittävimmistä on DOM API. [What is JavaScript? 2025.]

Document Object Model eli DOM esittää verkkosivun rakenteen hierarkkisena mallina. Kun web-selain lataa HTML-sivun, se muuntaa HTML-tiedoston koodin DOM-puuksi (engl. *DOM tree*), joka esittää tiedoston loogisena puuna (kuva 3). Jokainen puun haara päättyy solmuun (engl. *node*) ja jokainen solmu sisältää objekteja. [Introduction to the DOM 2025.]



Kuva 3. HTML-dokumentti esitettynä DOM:in avulla [Live Dom Viewer n.d.].

DOM mahdollistaa verkkosivun rakenteen, tyylin ja sisällön muokkaamisen JavaScriptin avulla. Tyypillisiä DOM:in avulla tehtäviä toimintoja ovat esimerkiksi erilaisten tapahtumakuuntelijoiden ja -käsittelijöiden lisääminen elementteihin (engl. *event listener* ja *event handler*) ja niiden avulla käyttäjän tekemiin toimiin reagoiminen. Lisäksi DOM:in avulla voidaan hakea verkkosivun elementeistä käyttäjän syöttämää dataa ja vastavuoroisesti lisätä haluttuja elementtejä käyttäjän toimien mukaan. DOM ja muut ohjelmointirajapinnat ovat siis merkittävässä roolissa verkkosivujen dynaamisuuden ja toiminnallisuuden rakentamisessa. [DOM Scripting Introduction 2025.]

3 WebAssembly

Vuonna 2017 julkaistu WebAssembly on web-ympäristöjen kehittämisessä käytettävä matalan tason binäärimuotoinen käskyformaatti, jota voidaan suorittaa lähes natiivilla konekielen nopeudella. Toisin kuin yleisesti web-sovelluskehityksessä käytettävä JavaScript, WebAssembly ei itsessään ole ohjelmointikieli, vaan käännöskohde, joksi lähdekoodi käännetään. Binäärimuotonsa vuoksi WebAssemblya ei ole tarkoitus kirjoittaa JavaScriptin tavoin itse, vaan lähdekoodin kielestä riippuen käytetään eri käännösohjelmia koodin kääntämiseksi WebAssemblyksi. [WebAssembly Concepts 2025; Zakai 2020.]

3.1 WebAssembly-koodin rakenne

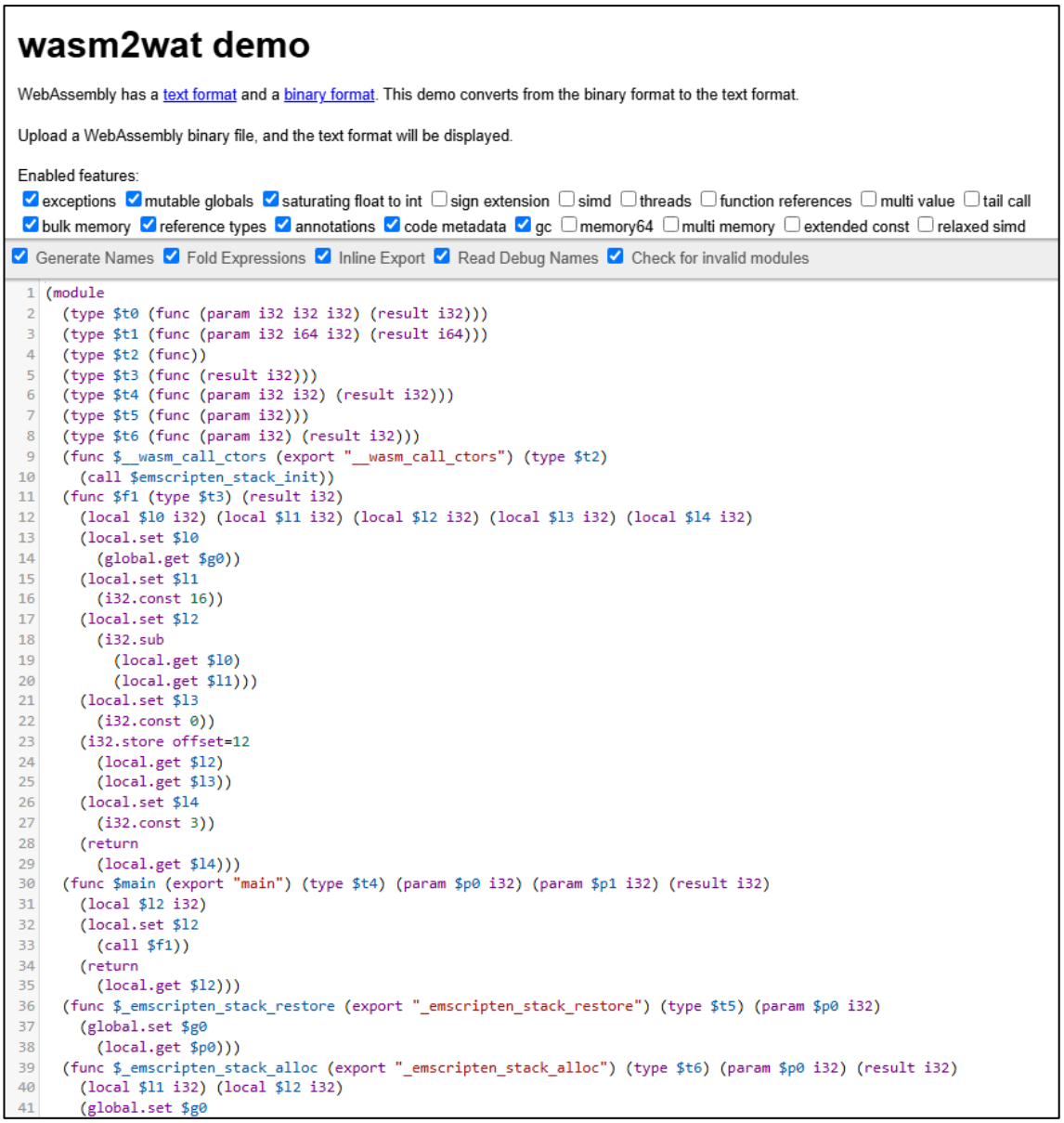
WebAssemblyksi voidaan kääntää useita eri ohjelmointikieliä, mutta kattavin tuki on C-, C++- ja Rust-ohjelmointikielille. Ohjelmointikielestä riippuen lähdekoodin kääntämiseen vaadittava käännöstyökalu vaihtelee ja esimerkiksi C- ja C++-koodin kääntämiseen käytetään Emscripten-käännöstyökalua. [Developer’s Guide n.d.] WebAssemblyn tukiessa web-sovelluskehityksessä useita epätyypillisiä kieliä, se mahdollistaa uudenlaisten sovellusten ajamisen verkossa binäärimuotonsa vuoksi lähes natiivilla nopeudella [WebAssembly Concepts 2025]. Toisin kuin JavaScript-koodi, joka on helposti ihmisen kirjoitettavissa ja luettavissa, WebAssembly-koodia ei ole sellaisenaan tarkoitettu kirjoitettavaksi tai luettavaksi [Zakai 2020]. WebAssemblyn binäärimuodosta

huolimatta, sitä on mahdollista tarkastella erillistä työkalua käyttäen. Havainnollistetaan WebAssembly-koodia esimerkin avulla: Kuvassa 4 on yksinkertainen C-ohjelmointikielellä tehty ohjelma, joka palauttaa lukujen 1 ja 2 yhteenlasketun summan.

```
1   int main()
2   {
3       return 1+2;
4   }
```

Kuva 4. Yksinkertainen C-kielellä kirjoitettu ohjelma, joka laskee yhteen kokonaisluvut 1 ja 2 ja palauttaa tuloksen.

Kuvan 4 ohjelma voidaan kääntää C- ja C++-kielten kääntämiseen tarkoitetulla Emscripten-käännöstyökalulla WebAssembly-koodiksi. Tulokseksi saadaan .wasm-päätteinen tiedosto, jota ei ole mahdollista lukea sellaisenaan binäärimuotonsa takia. WebAssemblysta on olemassa kuitenkin tekstimuotoinen formaatti WAT (WebAssembly Text Format), jota käytetään pääsääntöisesti koodin tarkasteluun mahdollisissa ongelmatilanteissa [Understanding WebAssembly Text… 2025]. Kuvassa 5 on kuvan 4 lähdekoodista käännetty Wasm-tiedosto WAT-formaattiin muunnettuna. Muunnokseen on käytetty erillistä wasm2wat demo -sovellusta.



Kuva 5. C-kielestä WebAssemblyksi käännetty koodi WAT-formaatissa [Wasm2wat Demo].

Kuvassa 5 näkyvässä WAT-formaattisessa WebAssembly-koodissa on lähdekoodin neljän rivin sijaan yhteensä 231 riviä. WAT-muotoisesta koodiesimerkistä on nähtävissä WebAssembly-koodin keskeisiä rakenteita. Sekä binääri- että tekstimuodossa WebAssembly-koodin perusyksikkö on kuvassa rivillä 1 näkyvä moduuli. Moduuli edustaa käännettyä ja suoritettavaa WebAssembly-binääritiedostoa, ja se esitetään yhtenä suurena S-lausekkeena (engl. *S-expression*), joka on vanha ja yksinkertainen tapa esittää dataa tekstimuodossa. Toisin

kuin JavaScriptin käyttämä abstrakti syntaksipuu WebAssemblyn puu on varsin litteä ja koostuu pääasiassa käskysarjoista. Moduulia voidaan pitää puuna, jonka solmut kuvaavat moduulin rakennetta ja sen koodia. [Understanding WebAssembly Text… 2025; WebAssembly Concepts 2025.]

Kuvan 4 (s. 10) main()-funktio näkyy kuvassa 5 (s.11) riveillä 30-35. Sekä funktion parametrit että paluuarvo ovat 32-bittisiä kokonaislukuja. WebAssemblyn funktiot tukevat vain yksinkertaisia matalan tason tietotyyppejä, joita ovat 32- ja 64-bittiset kokonais- ja liukuluvut sekä vektori- ja viittaustyypit. Rivillä 30 main()-funktioon liitetty export mahdollistaa funktion kutsumisen moduulin ulkopuolelta. Koodissa on näkyvillä sekä paikallisia että globaaleja muuttujia. Paikalliset muuttujat ovat kuin JavaScriptin var-muuttujat, mutta muuttujien tyypit ovat täsmällisiä. WebAssemblylla on mahdollista tehdä globaaleista muuttujista instansseja, joita voidaan käyttää sekä JavaScriptistä että toisesta WebAssemblyn moduulista. [Understanding WebAssembly Text… 2025.]

WebAssemblyn keskeisiin käsitteisiin kuuluvat myös taulu (engl. *table*) ja muisti. Kuvan 5 (s. 11) WebAssembly-koodin viimeiset rivit näkyvät kuvassa 6, jossa taulu ja muisti on määritelty riveillä 226 ja 227. Taululla tarkoitetaan laajennettavissa olevaa tyypitettyä viitteiden taulukkoa (engl. *array*), joka voi sisältää esim. viitteitä funktioihin. Taulua käytetään viitteille, joita ei turvallisuus- ja siirrettävyyssyistä voi tallentaa raakatavuina muistiin. WebAssemblyn muisti mahdollistaa monimutkaisempien tietotyyppien, kuten merkkijonojen, käsittelyn. Muisti toimii lineaarisesti ja sen kokoa voidaan kasvattaa dynaamisesti. Lineaarista muistia voidaan lukea ja sinne voi kirjoittaa arvoja mille tahansa tavuosoitteelle. [Overview 2025; Understanding WebAssembly Text… 2025; WebAssembly Concepts 2025.]

```
226   (table $__indirect_function_table (export "__indirect_function_table") 1 1 funcref)
227   (memory $memory (export "memory") 258 258)
228   (global $g0 (mut i32) (i32.const 65536))
229   (global $g1 (mut i32) (i32.const 0))
230   (global $g2 (mut i32) (i32.const 0)))
231
```

Kuva 6. Taulukko ja muisti WebAssemblyn WAT-muotoisessa koodissa [Wasm2wat Demo].

3.2 WebAssemblyn ominaisuudet

WebAssemblya ei ole tarkoitettu korvaamaan JavaScriptiä, vaan se kehitettiin JavaScriptin rinnalle täydentämään sitä suurta laskentatehoa vaativien sovellusten toteuttamisessa. JavaScript riittää edelleen tarpeeksi hyvin useimpien websovellusten toteuttamiseen, mutta mitä vaativammaksi sovellukset kehittyvät, sitä todennäköisemmin toteutus pelkällä JavaScriptillä voi johtaa ongelmiin suorituskyvyn kanssa. Esimerkiksi erilaiset kuva- ja videoeditorit, 3D-mallinnusta käyttävät pelit sekä virtuaalisen ja lisätyn todellisuuden (VR ja AR) sovellukset hyötyvät WebAssemblyn käyttämisestä. [WebAssembly Concepts 2025.]

WebAssembly-koodi on suunniteltu nopeaksi ja tiedostokooltaan pieneksi ja sitä voidaan JavaScriptin tavoin suorittaa nykyaikaisissa web-selaimissa. Siinä missä JavaScript-koodi täytyy lataamisen jälkeen jäsentää, tulkata ja kääntää, WebAssembly on jo valmiiksi käännettyä binäärimuotoista koodia ja valmista suoritettavaksi. JavaScriptin suoritusaikaa hidastaa myös sen dynaaminen tyypitys, sillä muuttujat luodaan ja määritetään ajonaikaisesti. WebAssemblyssa muuttujat on määritetty jo ennen ohjelman suorittamista ja täten koodin suorittaminen on nopeampaa. WebAssembly-koodi on kuitenkin käännettävä lähdekoodista etukäteen erillisen käännöstyökalun avulla, mitä ei JavaScriptissä tarvitse tehdä. Tällä ei kuitenkaan ole vaikutusta suoritusaikaan, sillä käännös tehdään jo kehitysvaiheessa eli ennen kuin koodi päätyy selaimelle. [WebAssembly – Quick Guide n.d.]

Selaimen verkkoalusta (engl. *web platform*) voidaan jakaa kahteen osaan: virtuaalikoneeseen (engl. *virtual machine*, VM) ja web-ohjelmointirajapintoihin.

Virtuaalikone suorittaa esimerkiksi web-sovelluksen JavaScript-koodia ja web-ohjelmointirajapinnoilla voidaan puolestaan ohjata web-selaimen tai laitteen toimintoja esimerkiksi DOM-rajapinnan kautta. Perinteisesti virtuaalikone on kyennyt lataamaan vain JavaScript-koodia, mutta WebAssemblyn kehittämisen myötä se kykenee lataamaan ja suorittamaan myös WebAssemblya. [WebAssembly Concepts 2025.]

WebAssemblyn heikkoutena on kuitenkin se, ettei se voi tällä hetkellä suoraan käyttää web-ohjelmointirajapintoja. Siispä mikäli WebAssembly-koodista halutaan kutsua jotakin ohjelmointirajapintaa, on kutsuttava ensin JavaScriptiä, joka sen jälkeen hoitaa varsinaisen kutsun esimerkiksi DOM:iin. WebAssembly ei siis suoraan pysty muokkaamaan esimerkiksi verkkosivun tekstielementtejä. Tämä rajoite on kuitenkin tarkoitus ratkaista tulevaisuudessa. WebAssemblyn ollessa matalan tason koodia se tukee rajatusti eri tietotyyppien välittämistä funktioissa. WebAssembly tukee ainoastaan kokonais- ja liukulukuarvoja (engl. *integer* ja *float*) funktioiden parametreina ja paluuarvoina. Monimutkaisempien tietotyyppien, kuten merkkijonojen (engl. *string*) ja taulukoiden, välittämiseen käytetään muistia ja osoittimia (engl. *pointer*). [WebAssembly Concepts 2025.]

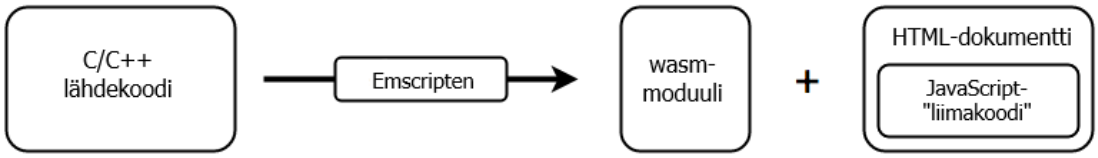
Turvallisuuden suhteen WebAssembly on kehitetty kahta tavoitetta silmällä pitäen: käyttäjien laitteiden tulee olla suojassa virheellisiltä ja haitallisilta moduuleilta ja kehittäjille on tarjottava hyödyllisiä perustoimintoja ja ratkaisuja turvallisten sovellusten kehittämiseen. Käyttäjille turvallisuus näkyy esimerkiksi siten, ettei lähdekoodista käännettyä WebAssembly-moduulia suoriteta suoraan selaimessa vaan eristetyssä hiekkalaatikkoympäristössä. WebAssembly-moduuli pystyy kommunikoimaan selaimen kanssa vain WebAssembly JavaScript API:n avulla eli toisin sanoen WebAssembly-koodin ulkopuoliset kutsut molempiin suuntiin onnistuvat vain JavaScriptin välityksellä. WebAssembly-moduulilla ei myöskään ole pääsyä laitteen resursseihin. Kehittäjille turvallisuustekijät näkyvät muun muassa siten, että moduulien on määriteltävä kaikki käytettävissä olevat funktiot ja niihin liittyvät tyypit lataushetkellä. Lisäksi käännetty koodi on muuttumatonta eikä ole havaittavissa ohjelman ajon aikana, mikä suojaa

WebAssembly-ohjelmia ohjausvirran kaappausyrityksiltä. [Security n.d.; WebAssembly – Quick Guide n.d.]

3.3 C- ja C++-ohjelmointikielten kääntäminen WebAssemblyksi

Koska WebAssembly-koodi syntyy käännöskohteena, vaatii se aina lähdekoodin kääntämisen etukäteen web-sovelluskehittäjän toimesta. Emscripten on C- ja C++- ohjelmointikielille tarkoitettu avoimen lähdekoodin käännösohjelma, joka käyttää kääntämisessä LLVM-kääntäjäinfrastruktuuria. Emscriptenin avulla on mahdollista kääntää lähes mitä tahansa C- tai C++ lähdekoodia WebAssemblyksi. [About Emscripten n.d.] Koska WebAssembly ei yksinään kykene käyttämään web-selaimen ohjelmointirajapintoja, kuten DOM:ia, Emscripten luo lisäksi automaattisesti niin sanotun JavaScript-liimakoodin. WebAssembly voi käyttää ohjelmointirajapintoja kutsumalla JavaScriptiä, mutta kykenee välittämään vain primitiivisiä kokonais- ja liukulukutyyppisiä arvoja. JavaScript-liimakoodin tarpeellisuus saattaa kuitenkin muuttua tulevaisuudessa, sillä suunnitteilla on mahdollistaa ohjelmointirajapintojen käyttäminen suoraan WebAssemblysta. [WebAssembly Concepts 2025.]

Kuvassa 7 esitetään käännösprosessi, jossa C- tai C++-kielinen lähdekoodi käännetään Emscriptenin avulla WebAssembly-moduuliksi. Lisäksi Emscripten luo pakollisen JavaScript-liimakoodin ja tarvittaessa HTML-tiedoston. HTML-tiedosto on pakollinen WebAssemblyn ja JavaScriptin toiminnan kannalta, mutta sen voi halutessaan luoda myös itse [WebAssembly Concepts 2025].



Kuva 7. C/C++-lähdekoodin käännösprosessi Emscripten-käännöstyökalulla [WebAssembly Concepts 2025].

WebAssemblyksi käännettävä lähdekoodi voi olla koko sovelluksen koodipohja tai vain osa sitä. WebAssemblyn avulla on mahdollista tuoda JavaScriptin ulkopuolista olemassa olevaa koodia osaksi web-sovelluksia ilman tarvetta erilliselle JavaScript-käännökselle. [Use Cases n.d.]

4 WebAssemblyn käyttäminen esimerkkisovelluksissa

4.1 Alkuvalmistelut

Tässä luvussa havainnollistetaan WebAssemblyn käyttämistä yksinkertaisten esimerkkien avulla. Kaikissa esimerkeissä WebAssemblyksi käännettävä lähdekoodi on kirjoitettu C-kielellä ja niissä havainnollistetaan C-lähdekoodista Wasm-koodiksi käännettyjen funktioiden käyttämistä JavaScriptin kautta eri tavoin. Emscripten-käännöstyökalulla on mahdollista luoda automaattisesti myös yksinkertainen HTML-tiedosto, mutta esimerkeissä ei tätä toimintoa käytetä. Emscripteniä käytetään siis ainoastaan C-lähdekoodin kääntämiseen Wasm-koodiksi ja JavaScript-liimakoodiksi. Esimerkkien toteuttamisessa on käytetty seuraavia ohjelmia:

- Windows 10 -käyttöjärjestelmä
- Git-versionhallintajärjestelmä
- Python-ohjelmointikielen versio 3.13.1
- Emscripten SDK versio 4.0.1
- Visual Studio Code -editori
  Lisäksi seuraavat Visual Studio Coden laajennukset:
  - C/C++ for Visual Studio Code
  - WebAssembly Toolkit for VSCode
  - Live Server
- Google Chrome -selain.

Koska esimerkeissä käytettävä lähdekoodi on kirjoitettu C-kielellä, käännöstyökaluna käytetään C- ja C++-ohjelmointikieliä tukevaa Emscripteniä. Emscripteninkäyttäminen eroaa hieman käyttöjärjestelmien välillä ja tässä opinnäytetyössä kuvatut esimerkit koskevat toimintaa Windows-ympäristössä. Emscripten

ladataan ja asennetaan Emscriptenin verkkosivujen asennusohjeiden mukaisesti. Emscripten SDK -ohjelmistokehityspaketin lataaminen onnistuu joko Git-versionhallintajärjestelmän avulla kloonaamalla repositorio tai GitHubin kautta ZIP-tiedostona. Emscripten SDK:n ajuri on Python-skripti, joten tietokoneessa tulee olla asennettuna lisäksi Pythonin versio 3.6 tai uudempi. [Emscripten Tutorial n.d.]

Emscripteniä käytetään Emscripten Compiler Frontendin (emcc) avulla joko Windowsin komentokehotteen (engl. *command prompt*) tai Emscripten Windows Command Promptin kautta. Emscripten Windows Command Prompt avataan paikallistamalla ja avaamalla Emscripten SDK:n asennussijainnista tiedosto *emcmdprompt.bat*. [Emscripten Tutorial n.d.]

4.2 Ccall()- ja cwrap()-funktiot

Seuraavissa esimerkeissä havainnollistetaan käännetyn C-koodin main()-funktion ulkopuolisten funktioiden kutsumista Emscriptenin ccall()- ja cwrap()-funktioiden avulla. Ccall()-funktio kutsuu käännettyä C-funktiota määritetyillä parametreilla ja palauttaa funktion paluuarvon. Cwrap() puolestaan ”käärii” käännetyn C-funktion ja palauttaa JavaScript-funktion, jota voi kutsua sen jälkeen normaalisti. Cwrap() toimii erityisesti tilanteissa, joissa samaa funktiota on tarve kutsua useasti. [Interacting With Code n.d.]

Oletusarvoisesti Emscriptenillä luotu koodi kutsuu aina main()-funktiota ja muut funktiot poistetaan kuolleena koodina käännetyn tiedoston koon minimoimiseksi. Kuvan 8 C-koodiesimerkissä funktio addNumbers() saa parametreinaan kaksi kokonaislukua ja palauttaa näiden summan. Jotta funktiota ei poisteta ja sitä voi kutsua, on sen yhteyteen laitettava kuvan 8 rivillä 3 näkyvä makro EMSCRIPTEN_KEEPALIVE. Makro vaatii toimiakseen emscripten.h-kirjaston, joka otetaan käyttöön rivillä 1 #include <emscripten/emscripten.h>. [Compiling a New… 2024; FAQ n.d.]

```c
#include <emscripten/emscripten.h>

EMSCRIPTEN_KEEPALIVE
int addNumbers(int a, int b)
{
    return a + b;
}
```

Kuva 8. C-kielinen funktio addNumbers() ja Emscriptenin vaatimat lisäykset koodiin.

Kuvan 8 C-kielinen koodi käännetään Wasm-koodiksi käyttämällä Emscripteniä komentokehotteen kautta. Komentokehotteen sijainnin tulee olla aktiivisena samassa hakemistossa käännettävän tiedoston kanssa ja komentokehotteeseen syötetään esimerkkikoodin 1 komento. Mikäli C-koodiin tehdään WebAssembly-käännöksen jälkeen muutoksia, on komento syötettävä uudestaan. Rivinvaihdot eivät kuulu komentoon, vaan ne on lisätty esimerkkiin sen selkeyttämiseksi.

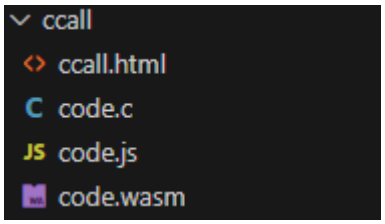
```
emcc code.c -o code.js
-sNO_EXIT_RUNTIME=1
-sEXPORTED_RUNTIME_METHODS=ccall,cwrap
```

Esimerkkikoodi 1. Komentokehotteeseen syötettävä emcc-komento ccall()- ja cwrap()-funktioiden käyttämisen mahdollistamiseksi.

Esimerkkikoodin 1 komento emcc viittaa Emscripten-kääntäjään ja code.c kuvan 8 C-kieliseen lähdekoodiin. Optio -o määrittää kohteen, joka esimerkissä on nimetty code.js:ksi. Jos kohdetta ei ole erikseen määritelty, Emscripten luo sekä JavaScript- että Wasm-tiedostot nimillä a.out.js ja a.out.wasm. Optio -sNO_EXIT_RUNTIME=1 estää main()-funktion suorittamisen jälkeen suoritusympäristön sulkeutumisen, jotta addNumbers()-funktiota voidaan käyttää. Optio -sEXPORTED_RUNTIME_METHODS=ccall,cwrap kertoo kääntäjälle, että Emscriptenin ccall()- ja cwrap()-funktioita halutaan käyttää. [Compiling a New… 2024; Emscripten Compiler… n.d.; FAQ n.d.]

Emscripten-kääntäjä luo esimerkkikoodin 1 komennon syöttämisen jälkeen code.c lähdekoodista kuvassa 9 näkyvät tiedostot code.js ja code.wasm. Näistä

code.wasm on käännetty WebAssembly-moduuli code.c-tiedostosta ja code.js on liimakoodi, joka toimii siltana JavaScriptin ja WebAssemblyn välillä. [WebAssembly Concepts 2025.] Kuvassa näkyvä ccall.html-tiedosto on erikseen luotu eikä liity Emscripten-kääntäjän toimintaan.



Kuva 9. Emscripten-kääntäjän C-tiedostosta luomat JavaScript- ja WebAssembly-tiedostot.

WebAssemblyksi käännetty C-koodi saadaan käyttöön lisäämällä se HTML-tiedoston <script>-tagiin lähteeksi kuvan 10 osoittamalla tavalla. Web-selaimen avatessa HTML-sivun, code.js-tiedosto lataa WebAssembly-moduulin. [WebAssembly Concepts 2025.] Tähänastiset toimenpiteet pätevät sekä ccall()- että cwrap()-funktioiden käyttöönottoon.

```
<head>
    <title>CCALL JA CWRAP</title>
    <script src="code.js"></script>
</head>
```

Kuva 10. Code.js-tiedosto <script>-tagin lähteenä.

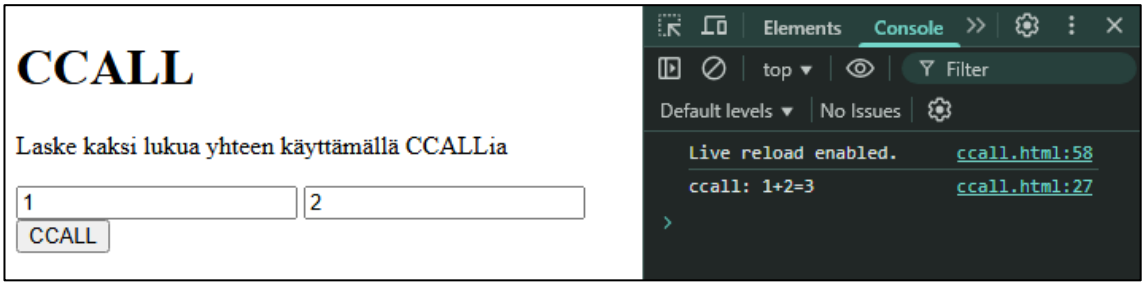
Ccall()-funktiota käytetään käännetyn C-funktion kertaluontoiseen kutsumiseen. Kuvan 11 esimerkissä on funktio useCCall(), jossa sijoitetaan ensin id-attribuuteilla a ja b olevien elementtien arvot muuttujiin a ja b (rivit 19 ja 20). Tämän jälkeen code.wasm-moduulista kutsutaan ccall()-funktion avulla käännettyä C-funktiota addNumbers() ja funktion paluuarvo sijoitetaan muuttujaan result (rivit 21-26). Rivin 21 Module viittaa code.wasm-moduuliin. Ccall()-funktion ensimmäinen parametri rivillä 22 on kutsuttavan C-funktion nimi ja sen täytyy täsmätä C-tiedostossa olevan funktion nimen kanssa (kuva 8, s. 18). Toinen parametri

määrittää paluuarvon tyypin ja kolmas addNumbers()-funktion argumenttien tyypit. Viimeinen parametri määrittää taas argumentit, jotka tässä tapauksessa ovat muuttujat a ja b. [Interacting With Code n.d.]

```
17      <script>
18          function useCCall() {
19              let a = document.querySelector("#a").value;
20              let b = document.querySelector("#b").value;
21              let result = Module.ccall(
22                  "addNumbers",    // C-funktion nimi
23                  "number",    //paluuarvon tyyppi
24                  ["number", "number"],    //argumenttien tyypit
25                  [a, b]  //argumentit
26              );
27              console.log(`ccall: ${a}+${b}=${result}`);
28          }
29      </script>
```

Kuva 11. Koodiesimerkki ccall()-funktion toiminnasta.

Kuvassa 12 näkyy ladattu HTML-sivu, jossa lomake-elementeistä saadut arvot lasketaan painiketta klikkaamalla yhteen addNumbers()-funktion avulla. Kuvan 11 rivillä 27 oleva konsolitulostus näkyy kuvan 12 konsolissa ja laskutoimituksen tuloksesta voidaan päätellä C-funktion kutsun onnistuneen.



Kuva 12. Ccall()-esimerkkisovelluksen toiminta web-selaimessa.

Cwrap()-funktion toteutuksessa on jonkin verran yhtäläisyyksiä ccall()-funktion kanssa. Kuvassa 13 esitetään riveillä 18-23 cwrap()-funktion toteutus. Ccall()-funktion tavoin Wasm-moduulin kautta kutsutaan kuvan 11 kanssa samoilla

parametreilla samaa C-kielestä käännettyä addNumbers()-funktiota. Erona ccall()-funktioon cwrap()-funktio palauttaa käännettyyn C-funktioon ”kääritty” JavaScript-funktion, joka sijoitetaan addNums-muuttujaan. Tätä käärittyä funktiota kutsutaan koodissa rivillä 28, jolloin result-muuttujan arvoksi sijoitetaan addNums()-funktiolla kutsutun addNumbers()-funktion paluuarvo. [Interacting With Code n.d.]

```
17      <script>
18          let addNums = Module.cwrap(
19              "addNumbers",    // C-funktion nimi
20              "number",    //paluuarvon tyyppi
21              ["number", "number"],    //argumenttien tyypit
22              [a, b]  //argumentit
23          );
24
25          function useCWrap() {
26              let a = document.querySelector("#a").value;
27              let b = document.querySelector("#b").value;
28              let result = addNums(a, b);  //Tässä kutsutaan "käärittyä" C-funktiota
29              console.log(`cwrap: ${a}+${b}=${result}`);
30          }
31      </script>
```

Kuva 13. Koodiesimerkki cwrap()-funktion toiminnasta.

Kuvan 13 cwrap()-koodiesimerkki johtaa muuten samaan lopputulokseen kuvan 12 ccall()-esimerkin kanssa, mutta maininnat ccallista on korvattu cwrapilla.

4.3 WebAssembly-moduulin striimaus

Vaihtoehtoinen tapa WebAssemblyksi käännettyjen funktioiden kutsumiseen on WebAssembly-moduulin striimaus. Tämä tapa eroaa ccall()- ja cwrap()-funktioista sillä, että yhden tietyn funktion kutsumisen sijaan voidaan hakea koko moduuli ja tallettaa sen sisältämät viedyt funktiot yhteen muuttujaan. Lisäksi erona on myös se, ettei striimauksessa käytetä Emscriptenin tarjoamia funktioita. [Using the WebAssembly… 2025.] Aiempien koodiesimerkkien tapaan tässäkin esimerkissä käytetään käännettävänä C-kielen lähdekoodina kuvan 8 (s.18) esimerkkiä.

Ensimmäinen eroavaisuus aiempiin tapoihin ilmenee Emscripteniin syötettävässä komennossa. C-kielen käännettävä funktio on edelleen merkittävä EMSCRIPTEN_KEEPALIVE-makrolla, jotta sitä voidaan kutsua moduulin ulkopuolelta. Itse komennoksi riittää esimerkkikoodin 2 komento, jolla ainoastaan käännetään C-kielinen lähdekoodi sekä JavaScript- että WebAssembly-koodiksi.

```
emcc code.c -o code.js
```

Esimerkkikoodi 2. Komentokehotteeseen syötettävä emcc-komento WebAssembly-moduulia striimatessa.

HTML-tiedoston osalta (kuva 14) Wasm-moduulin striimaus eroaa ccall()- ja cwrap()-funktioiden käytöstä siten, ettei JavaScript-liimakoodia tarvitse määrittää lähteeksi <script>-tagiin kuvan 10 (s. 19) tavoin.

```
4   <head>
5       <title>Streaming</title>
6   </head>
```

Kuva 14. HTML-tiedosto ilman määritettyä JavaScript-tiedostoa.

Kuvassa 15 esitetään WebAssembly-moduulin striimaus ja sen kautta C-funktion addNumbers() kutsuminen. Riveillä 19-21 käytetään WebAssembly.instantiateStreaming()-funktiota, joka hakee moduulin, kääntää sen ja luo siitä instanssin yhdellä kertaa raa'asta tavukoodista. InstantiateStreaming()-funktion parametreina ovat pakollinen striimauksen lähde ja valinnainen tuontiolio (engl. *import object*), jota käytetään esimerkiksi muistin hallinnassa. Tässä esimerkissä valinnainen parametri jätetään tyhjäksi. Code.wasm-tiedoston hakemiseen käytetään Fetch API:a. Tuloksena saadun moduuli-instanssin viedyt funktiot sijoitetaan erikseen luotuun exports-muuttujaan rivillä 21. [Loading and Running… 2025; Using the WebAssembly… 2025; WebAssembly.instantiateStreaming() 2025.]

```
15      <script>
16          let exports;
17          //sijoitetaan code.wasm-tiedostosta striimattu
18          //wasm-moduuli exports-muuttujaan
19          WebAssembly.instantiateStreaming(
20              fetch("code.wasm"), {}
21          ).then(results => exports = results.instance.exports);
22
23          function addNums() {
24              let a = document.getElementById("a").value;
25              let b = document.getElementById("b").value;
26              //Kutsutaan C-funktiota. Funktion nimen (addNumbers)
27              //täsmättävä C-tiedoston funktion kanssa!
28              let result = exports.addNumbers(a,b);
29              console.log(`Striimattuna: ${a}+${b}=${result}`);
30          };
31      </script>
```

Kuva 15. Koodiesimerkki C-funktion kutsumisesta WebAssembly-moduulin striimauksen avulla.

C-kielestä käännetyn funktion kutsuminen tapahtuu kuvan 15 rivillä 28. Funktioon addNumbers() päästään käsiksi aiemmin luodun exports-muuttujan avulla, johon on talletettu code.wasm-moduulista kaikki sen ulkopuolelta kutsuttavissa olevat funktiot. Kuten aiemmissakin esimerkeissä, myös tässä tapauksessa rivin 28 addNumbers() on täsmättävä C-funktion nimen kanssa. [Loading and Running… 2025.] Kuvassa 15 oleva koodi toimii ulkoisesti katsottuna samalla tavalla kuin aiemmat esimerkit ja vastaa lähes täysin kuvan 12 (s. 20) tulosta.

4.4 Merkkijonon välittäminen

Aikaisemmissa esimerkeissä käännettyjen funktioiden parametrit ja paluuarvot ovat olleet numeraalisia. Myös merkkijonojen välittäminen on mahdollista, vaikka WebAssembly ei sitä suoraan tuekaan. Seuraavassa esimerkissä käydään läpi yksi mahdollinen tapa merkkijonojen välittämiseen molempiin suuntiin. Kuvassa 16 esitetään C-lähdekoodi, jossa on funktiot sekä merkkijonojen viemiseen että tuomiseen.

```c
#include <stdio.h>
#include <emscripten/emscripten.h>

EMSCRIPTEN_KEEPALIVE
char *getString()
{
    return "Tämä on WebAssemblyn palauttama merkkijono.";
}

EMSCRIPTEN_KEEPALIVE
int addString(char *str)
{
    printf("Vastaanotettu merkkijono: %s\n", str);
    return 0;
}
```

Kuva 16. C-kieliset funktiot merkkijonojen välittämiseen.

Kuvan 16 funktio *getString() (rivit 5-9) palauttaa ennalta määrätyn merkkijonon, joka on tallennettu vakiona ohjelman muistialueelle, eikä se siten ole muokattavissa. Funktio addString() (rivit 11-15) saa parametrinaan merkkijonon osoittimen ja tulostaa merkkijonon konsoliin käyttämällä printf()-funktiota. Printf()-funktion käyttämiseksi rivillä 1 on otettu käyttöön kirjasto stdio.h.

Kuvan 16 koodi käännetään WebAssemblyksi esimerkkikoodin 3 komennolla. Kuten aiemmissakin esimerkeissä, rivinvaihdot eivät kuulu komentoon, vaan ne on lisätty sen selkeyttämiseksi. Komennon kahdella ensimmäisellä rivillä käännetään lähdekoodi JavaScript-liimakoodiksi ja Wasm-koodiksi ja estetään suoritusympäristön sulkeutuminen main()-funktion suorittamisen jälkeen. -sEXPORTED_RUNTIME_METHODS sisältää esimerkissä käytettävän Esmcriptenin ccall()-funktion lisäksi nyt myös funktiot lengthBytesUTF8() ja stringToUTF8(). -sEXPORTED_FUNCTIONS toimii samalla tavalla kuin EMSCRIPTEN_KEEPALIVE-makro, eli se varmistaa, että funktio on kutsuttavissa moduulin ulkopuolelta. Tässä yhteydessä vietäviksi funktioiksi määritellään kuvan 16 funktioiden lisäksi malloc() ja free(). [FAQ n.d.; Interacting With Code n.d.]

```
emcc code.c -o code.js
-sNO_EXIT_RUNTIME=1
-sEXPORTED_RUNTIME_METHODS=ccall,lengthBytesUTF8,stringToUTF8
-sEXPORTED_FUNCTIONS=_malloc,_free
```

Esimerkkikoodi 3. Komentokehotteeseen syötettävä emcc-komento merkkijonojen välittämisen mahdollistamiseksi.

Kuvassa 17 esitetään JavaScript-toteutus käyttäjän syöttämän merkkijonon viemiseen C-funktioon (rivit 19-34) ja ennalta määritetyn merkkijonon hakemiseen (rivit 36-40). Tässä tapauksessa merkkijonon hakeminen on helpompaa, sillä merkkijono on jo määritetty C-tiedostossa, minkä lisäksi funktion getString() rivillä 37 oleva ccall()-funktio käyttää C-pinoa (engl. *stack*) tilapäisten arvojen luomiseen. Tällä tavoin välitetty merkkijono on kuitenkin elossa vain kutsun ajan eli jos merkkijonon osoite tallennetaan myöhempää käyttöä varten, voi se osoittaa virheelliseen dataan. Funktion getString()-toteutus on kuitenkin riittävä tähän yksinkertaiseen esimerkkiin. Rivin 37 ccall()-funktion toiseksi parametriksi eli paluuarvon tyypiksi on määritetty ”string”, jota käytetään char*-tyyppisille merkkijonoille. [Interacting With Code n.d.; Preamble.js n.d.] Haettu merkkijono tulostetaan konsoliin ja näytetään HTML-sivulla (kuva 18, s. 26).

```
18    <script>
19        function addString() {
20            let inputString = document.getElementById("a").value
21            // Muunnetaan merkkijono UTF-8:ksi ja kirjoitetaan se WebAssemblyn muistiin
22            let length = Module.lengthBytesUTF8(inputString) + 1; // +1 lopetusmerkkiä varten
23            let memoryAddress = Module._malloc(length); //Varataan muistia Wasmin muistista
24            Module.stringToUTF8(inputString, memoryAddress, length); //Kirjoitetaan muistiin
25
26            Module.ccall(
27                "addString",    //C-funktio
28                null,   // Ei paluuarvoa
29                ["number"], //Parametrin tyyppi
30                [memoryAddress] //Muistin osoite
31            );
32
33            Module._free(memoryAddress); // Vapautetaan varattu muisti
34        }
35
36        function getString() {
37            let result = Module.ccall("getString", "string", []);
38            console.log(result);
39            document.getElementById("return").innerHTML = "Haettu: " + result;
40        }
41    </script>
```

Kuva 17. Koodiesimerkki merkkijonojen viemiseksi ja hakemiseksi C-funktiolta.

Funktion addString() toteutuksessa on käytetty WebAssemblyn muistia merkkijonon välittämisessä. Käyttäjän syöttämän merkkijonon pituus lasketaan rivillä 22 funktiolla Module.lengthBytesUTF8() ja tähän lisätään +1 lopetusmerkkiä varten. Rivillä 23 Module._malloc() varaa merkkijonolle muistipaikan WebAssemblyn muistista edellisellä rivillä lasketun pituuden mukaan. Module.stringToUTF8() kirjoittaa merkkijonon muistiin käyttäen syötettyä merkkijonoa, sen pituutta ja muistipaikkaa. Merkkijono säilyy tällä tavoin WebAssemblyn muistissa, kunnes se erikseen vapautetaan. Riveillä 26-31 kutsutaan ccall()-funktion avulla C-funktiota addString(), joka tulostaa parametrina välitetyssä muistisijainnissa olevan merkkijonon. Rivillä 33 kutsutaan Module._free()-funktiota, joka vapauttaa aiemmin varatun WebAssemblyn muistin. [Interacting With Code n.d.; Preamble.js n.d.] Kuvan 17 koodiesimerkin toiminta esitetään kuvassa 18.

Kuva 18. Merkkijonojen välittämisen toiminta web-selaimessa.

Esimerkkisovelluksen toiminnasta nähdään, että merkkijonojen välittäminen toimii halutulla tavalla (kuva 18). Sovelluksen lomakekenttään syötetty teksti ”heippa” välittyy painiketta klikkaamalla C-kielen addString()-funktiolle ja konsoliin tulostuu ”Vastaanotettu merkkijono: heippa”. GetString()-funktion palauttama merkkijono puolestaan näkyy sekä tulosteena konsolissa että tekstielementtinä HTML-tiedostossa toista painiketta klikkaamalla.

5 Yhteenveto

Tämän opinnäytetyön tarkoituksena oli perehtyä WebAssemblyyn ja sen ominaisuuksiin sekä tarkastella WebAssemblyn roolia ja sen käyttämistä web-sovelluskehityksessä. Tavoitteena oli oman tietämykseni lisäksi lisätä

mahdollisten muiden tämän työn lukevien tietämystä aiheesta. Työssä pyrittiin tuottamaan materiaalia, joka voisi auttaa tarkastelemaan WebAssemblyn soveltuvuutta mahdollisiin omiin projekteihin.

Opinnäytetyössä käsiteltiin WebAssemblya edeltäviä teknologioita ja niiden vaikutusta sen kehittämiseen. Tarkastelun kohteena oli myös web-sovelluskehityksen lähtökohdat erityisesti JavaScriptin osalta huomioiden sen WebAssemblyyn vertautuvat ominaisuudet. Opinnäytetyössä käsiteltiin WebAssemblyn rakennetta, ominaisuuksia ja käännösprosessia Emscriptenin avulla. Lopuksi havainnollistettiin sekä Emscriptenin että WebAssemblyn käyttöä yksinkertaisten esimerkkien avulla. Esimerkit rajautuivat C-kieliseen lähdekoodiin ja funktiokutsujen suunta oli JavaScriptistä WebAssemblyyn.

Työlle asetetut tavoitteet saavutettiin erityisesti WebAssemblyyn liittyvän tietämyksen lisäämisen osalta. Opinnäytetyöprosessi opetti luonnollisesti itselleni lisää aiheesta, mutta uskon työn auttavan myös muita lukijoita WebAssemblyn perusteiden ja käyttöönoton hahmottamisessa. Haasteita opinnäytetyön tekemisessä aiheutti aiheen vahvasti englanninkielisen terminologian asiallinen suomentaminen, sillä aiheeseen liittyvää suomenkielistä materiaalia löytyi hyvin vähän. Työn toteutuksessa pidin kuitenkin tärkeänä englanninkielisten termien käytön minimoimisen, mutta sen seurauksena tietyt osat tekstistä saattavat olla haastavampia hahmottaa.

Opinnäytetyössä käsiteltiin WebAssemblyn käyttämistä web-selaimessa ja WebAssemblyn kutsumista JavaScriptin kautta. Mahdollisia jatkokehityskohteita tästä opinnäytetyöstä voisivat olla esimerkiksi funktiokutsujen toteuttaminen toiseen suuntaan eli käännettävästä lähdekoodista JavaScriptin kutsuminen tai kokonaisen valmiin C/C++-ohjelman tuominen web-selaimeen WebAssemblyn avulla. Lisäksi tämän työn ulkopuolelle rajattiin tarkoituksella WebAssembly System Interface eli WASI, joka laajentaa WebAssemblyn käytön selaimen ulkopuolelle esimerkiksi työpöytäsovelluksiin. Myös WASI:in perehtyminen voisi olla hyvä jatkumo tälle opinnäytetyölle.

Lähteet

About Emscripten. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/introducing_emscripten/about_emscripten.html> Luettu 22.1.2025.

Asm.js. 2024. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js?source=post_page> Päivitetty 23.7.2024. Luettu 11.4.2025.

Compiling a New C/C++ Module to WebAssembly. 2024. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_Wasm/> Päivitetty 19.12.2024. Luettu 22.1.2025.

Croad, Gemma. 2020. Understanding the JavaScript Runtime Environment. Verkkoaineisto. Medium. <https://medium.com/@gemma.croad/understanding-the-javascript-runtime-environment-4dd8f52f6fca> Julkaistu 3.8.2020. Luettu 28.3.2025.

Developer’s Guide. Verkkoaineisto. WebAssembly. <https://webassembly.org/getting-started/developers-guide/> Luettu 22.1.2025.

DOM Scripting Introduction. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/DOM_scripting> Päivitetty 11.4.2025. Luettu 16.4.2025.

Donovan, Alan; Muth, Robert; Chen, Brad & Sehr, David. 2010. PNaCl: Portable Native Client Executables. Massachusetts Institute of Technology <https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf> Julkaistu 22.2.2010. Luettu 20.3.2025.

Emscripten Compiler Frontend (emcc). Verkkoaineisto. Emscripten. <https://emscripten.org/docs/tools_reference/emcc.html> Luettu 12.4.2025.

Emscripten Tutorial. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/getting_started/Tutorial.html#> Luettu 22.1.2025.

FAQ. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/getting_started/FAQ.html#> Luettu 12.4.2025.

From asm.js to Wasm With Emscripten Creator Alon Zakai – WasmAssembly. 2024. Verkkoaineisto. Chrome for Developers. <https://www.youtube.com/watch?v=cv5uQ_hQVE0&ab_channel=ChromeforDevelopers> Julkaistu 26.4.2025. Katsottu 11.4.2025.

Grammar and Types. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types> Päivitetty 21.3.2025. Luettu 30.3.2025.

Interacting With Code. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html> Luettu 12.4.2025.

Introduction to Portable Native Client. Verkkoaineisto. The Cromium Projects. <https://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client/> Luettu 20.3.2025.

Introduction to the DOM. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction> Päivitetty 6.3.2025. Luettu 28.3.2025.

Introduction. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction> Päivitetty 6.3.2025. Luettu 28.3.2025.

JavaScript – Overview. Verkkoaineisto. Tutorials Point. <https://www.tutorialspoint.com/javascript/javascript_overview.htm> Luettu 10.3.2025.

Just-In-Time Compilation (JIT). 2024. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation> Päivitetty 30.12.2024. Luettu 1.4.2025.

Live DOM Viewer. Verkkoaineisto. <https://software.hixie.ch/utilities/js/live-dom-viewer/> Käytetty 28.3.2025.

Loading and Running WebAssembly Code. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Loading_and_running> Päivitetty 7.4.2025. Luettu 13.4.2025.

Native Client. 2024. Verkkoaineisto. Chrome for Developers. <https://developer.chrome.com/docs/native-client/> Päivitetty 11.10.2024. Luettu 20.3.2025.

O., Arika. 2022. How Web Browsers Work - Executing the Javascript (Part 5, With Illustrations). Verkkoaineisto. DEV. <https://dev.to/arikaturika/how-web-browsers-work-executing-the-javascript-part-5-with-illustrations-21ok/> Päivitetty 3.6.2022. Luettu 1.4.2025.

Overview. 2025. Verkkoaineisto. WebAssembly Community Group. <https://webassembly.github.io/spec/core/intro/overview.html> Päivitetty 28.1.2025. Luettu 17.4.2025.

Padmanabhan, Mukil. 2024. Inside JavaScript Engines: How Browsers Bring Your Code to Life. Verkkoaineisto. DEV. <https://dev.to/mukhilpadmanabhan/inside-javascript-engines-how-browsers-bring-your-code-to-life-h1> Julkaistu 22.11.2024. Luettu 6.4.2025.

Pattanayak, Laxmi Narayana. 2024. How JavaScript Really Works? Verkkoaineisto. DEV. <https://dev.to/laxminarayana31/how-javascript-really-works-1p6i> Päivitetty 14.4.2024. Luettu 1.4.2025.

Preamble.js. Verkkoaineisto. Emscripten. <https://emscripten.org/docs/api_reference/preamble.js.html#> Luettu 20.4.2025.

Security. Verkkoaineisto. WebAssembly. <https://webassembly.org/docs/security/> Luettu 19.4.2025.

Sehr, David. 2013. Portable Native Client: The "Pinnacle" of Speed, Security, and Portability. Verkkojulkaisu. Chromium Blog. <https://blog.chromium.org/2013/11/portable-native-client-pinnacle-of.html/> Julkaistu 12.11.2013. Luettu 25.3.2025.

Steiner, Thomas. 2023. What is WebAssembly and Where Did It Come From? Verkkoaineisto. Web.dev. <https://web.dev/articles/what-is-webassembly> Päivitetty 29.6.2023. Luettu 10.4.2025

Understanding WebAssembly Text Format. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format> Päivitetty 9.4.2025. Luettu 11.4.2025.

Use Cases. Verkkoaineisto. WebAssembly. <https://webassembly.org/docs/use-cases/> Luettu 22.4.2025.

Using the WebAssembly JavaScript API. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Using_the_JavaScript_API> Päivitetty 9.4.2025. Luettu 19.4.2025.

W3C. 2019. World Wide Web Consortium (W3C) Brings a New Language to the Web as WebAssembly Becomes a W3C Recommendation. World Wide Web Consortium. <https://www.w3.org/press-releases/2019/wasm/>. Julkaistu 5.12.2019. Luettu 7.3.2025.

Wasm2wat demo. Verkkoaineisto. <https://webassembly.github.io/wabt/demo/wasm2wat/> Käytetty 1.4.2025.

WebAssembly – Quick Guide. Verkkoaineisto. Tutorials Point. <https://www.tutorialspoint.com/webassembly/webassembly_quick_guide.htm> Luettu 7.4.2025.

WebAssembly Concepts. 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts/> Päivitetty 3.4.2025. Luettu 10.4.2025.

WebAssembly.instantiateStreaming(). 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/JavaScript_interface/instantiateStreaming_static> Päivitetty 13.3.2025. Luettu 20.4.2025.

What is JavaScript? 2025. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/What_is_JavaScript/> Päivitetty 10.3.2025. Luettu 10.3.2025.

Zakai, Alon. 2020. The History of WebAssembly. Verkkoaineisto. Chrome for Developers. <https://www.youtube.com/watch?v=6r0NKEQqkz0&ab_channel=ChromeforDevelopers>. Julkaistu 7.2.2020. Katsottu 5.3.2025.