Ilkka Takala

Yksinkertaisen skriptikielen suunnittelu ja toteutus pelimoottorien käytön tueksi

```
var currentNode = Node{0, null, begin};

Array.PushUnique(closedList, currentNode.state);

while (currentNode != null) {
    if (Visit(currentNode.state)) {
        for (child in getNextLegalStates(currentNode)) {

            if (child == null) continue;

            calculateCost(child);

            if (find(closedList, child.state, equal) != null) {
                continue;
            }

            var s = Map.Get(openList, child.state);
            if (s != null) {
                if (s.G > child.G) {
                    Map.insert(openList, child.state, child);
                }
            }
            else Map.insert(openList, child.state, child);
        }
    }
    else {
        break;
    }

    if (Array.Size(openList.keys) == 0) {
        break;
    }
```

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2024

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Takala Ilkka

Työn nimi: Yksinkertaisen skriptikielen suunnittelu ja toteutus pelimoottorien käytön tueksi

Tutkintonimike: Tietotekniikan ja viestinnän insinööri

Asiasanat: kääntäjä, skripti, ohjelmointi, C++, EMI, parseri, jäsentäjä, LALR, LR, virtuaalikone, bittikoodi, tavukoodi, rekisteri,

Tämän opinnäytetyön tavoitteena oli tutkia tulkattujen kielten luomista ja käyttöä. Opinnäytetyön aikana tutkittiin useita erilaisia lähestymistapoja koodin kääntämiseen, jäsentämiseen ja tuottamiseen ja näistä valittiin sopivimmat menetelmät EMI-skriptikielen toteuttamiseen. Toteutettu kieli tukee lukuisia nykyaikaisten ohjelmointikielten ominaisuuksia ja sitä voidaan käyttää ohjelmien luomiseen. Kääntäjä ja suoritusympäristö luotiin puhtaina C++-kirjastoina minimaalisilla riippuvuuksilla käyttäen CMakea käännösjärjestelmänä.

Ensimmäinen askel uuden kielen luomisessa oli tutkia kielen piirteitä ja tarvittavia ominaisuuksia. Tämä vaihe käytettiin eri kielten opiskeluun ja erilaisten mahdollisten piirteiden tutkimiseen. Lopullinen kielen syntaksi vastaa C-tyyliä, ja suurin ero ominaisuuksissa on dynaaminen tyypitys ja valinnaiset tyypittömät muuttujat. Kielen jäsentäminen tehdään LALR-jäsentäjällä, joka sisältää operaattorin ensisijaisuus- ja assosiatiivisuustaulukot. Tuloksena oleva konkreettinen syntaksipuu ohitetaan ja käsitellään suoraan abstraktiksi syntaksipuuksi jo jäsentämisen aikana tarpeettomien puusolmujen välttämiseksi. LALR-jäsentäjän käyttämät tokenit noudetaan lähdekoodista käyttämällä lekseriä, joka etsii syötevirrasta ennalta määritettyjä merkkisarjoja.

EMI-virtuaaliympäristö on rekisteripohjainen virtuaalikone, joka lukee ja suorittaa kääntäjän tuottamaa tavukoodia. Kääntäjä kävelee AST:n ja luo ennalta määritetyn käskysekvenssin kullekin solmutyypille. Rekisterivaraus käyttää yksinkertaistettua lineaarista varausmallia. Käännöstulos on yksinkertainen tavukoodi, joka sisältää rekistereitä ja käskyjä 32-bittisissä arvoissa, joita tarvitaan virtuaalikoneen ohjaamiseen. Luodun koodin suorituksessa käytetään epäsuoraa suoritusta linkittämään käskyt todelliseen konekieliseen koodiin.

Luotua kieltä ja ympäristöä testattiin kirjoittamalla yksinkertainen reitinhaku A*-algoritmilla. Ympäristön suoritusteho oli odotettua parempi ja se kykeni suorittamaan reitinhaun hyväksyttävissä ajoissa. EMIScriptiä, projektin lopputuotetta, voidaan käyttää tukemaan pelimoottoreiden käyttöä asynkronisilla skripteillä ja toiminnallisella koodilla.

KAMK • University of Applied Sciences

Abstract

Author(s): Takala Ilkka

Title of the Publication: Design and Creation of Simple Scripting Language for Supporting the Use of Game Engines

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: compiler, script, language, programming, C++, EMI, parser, LALR, LR, virtual machine, bytecode, register

This thesis aimed to study the creation and use of interpreted languages. During the thesis multiple different approaches to compilation, parsing, and generating of code were studied, and most suitable methods were selected for the implementation of EMI scripting language. The implemented language supports many features present in modern programming languages and can be used to create programs. The compiler and execution environment were created as pure C++-libraries with minimal dependencies using CMake as the build system.

The first step of creating a new language was to study the needs and required features of the language. This phase was mostly done by studying languages and conducting research on different possible features. The final language syntax matches C style and the largest difference in features is dynamic typing and optional untyped variables. The language parsing is done with LALR-parser which includes operator precedence and associativity tables. The resulting concrete syntax tree is partially skipped and directly parsed into abstract syntax tree during parsing to avoid unnecessary tree nodes. Tokens for the LALR parser are retrieved from source code using a lexer that matches predefined character sequences in the input stream.

The EMI virtual environment is a register-based machine that reads and executes bytecode generated by the compiler. The compiler walks the AST and generates the predefined sequence of opcodes for each node type. Register allocation uses simplified linear allocation model. The compilation result is simple bytecode that contains registers and opcodes needed to direct the virtual machine in 32-bit values. Evaluation of the generated code uses indirect dispatch to map opcodes to actual machine code.

The resulting language and environment were tested by creating a simple pathfinder using A*-algorithm. The execution performance of the environment was better than expected and could execute the pathfinder in suitable times. EMIScript, the final product of the project, can be used to easily augment game engines with asynchronous scripts and functional code.

Sisällys

Symboliluettelo

GPL	General purpose language, eli yleiskäyttöinen ohjelmointikieli
DSL	Domain specific language, eli täsmäkieli, joka on suunniteltu yhteen käyttökohteeseen
AOT	Ahead of Time compiling, kääntämisstrategia, jossa lähdekoodi käännetään konekieleen ennen ohjelman suoritusta.
JIT	Just in time compiling, kääntämisstrategia, jossa lähdekoodi käännetään konekieleen suorituksen aikana
JVM	Java virtual machine, Java-kielen suoritusympäristö
CPU	Central processing unit, tietokoneen suoritin
RAM	Random access memory, tietokoneen välimuisti
Tokeni	Merkkisarjaa kuvaava arvo
Tuotanto	Sarja tokeneita, jotka muodostavat kieliopin säännön
AST	Abstract syntax tree, eli abstrakti syntaksipuu, joka sisältää lähdekoodin puumuodossa
LL(1)	Left to right, leftmost first, single token lookahead. Algoritmi jäsentäjälle, jossa lähdekoodia käsitellään vasemmalta oikealla ja ensimmäinen toteutuva tuotanto lisätään AST:iin
LR(1)	Left to right, rightmost first, single token lookahead. Algoritmi jäsentäjälle, jossa lähdekoodia käsitellään vasemmalta oikealla ja uusin toteutuva tuotanto lisätään AST:iin
LALR(1)	look-ahead, left-to-right, rightmost first, single token lookahead. Sama kuin LR, mutta mahdollisten tuotantojen tiloja on optimoitu
LR-taulu	LR(x) tyyppisten parserien sääntökirja
IR	Intermediate representation, välimuoto suoritettavalle koodille kääntämisen aikana
NaN	Not-a-Number, epänumero
LLd	Last Level cache, viimeisen tason CPU-välimuisti Cachegrind-ohjelmassa
D1	First level cache, ensimmäisen tason CPU-välimuisti Cachegrind-ohjelmassa
A*	A-star reitinhakualgoritmi

1 Johdanto

Pelejä ja pelimoottoreita kehitettäessä törmätään usein ongelmaan, että pienet muutokset C++-koodiin testauksia varten vaativat usein ohjelman kääntämisen uusiksi. Useasti kehittämisen nopeuttamiseksi ja helpottamiseksi pelimoottoreissa on käytössä vaihtoehtoinen skriptauskieli, jolla käyttäjät voivat lisätä peliin toiminnallisuutta osaamatta C++- tai vastaavaa kieltä. Esimerkkinä tästä on Unreal Enginen Blueprintit.

Tässä projektissa kehitetään Eril-pelimoottorin tueksi skriptikieli Eril Manipulator Interface Script, lyhennettynä EMIScript. EMI-kieltä käytetään pelimoottorien rinnalla luomaan ja tukemaan helppoa ja nopeaa kehitysprosessia sekä ohjelmoijille että artisteille. Konkreettisesti tavoitteena on kehittää kielen syntaksi, syntaksille kääntäjä ja lopulta virtuaalikone, joka suorittaa käännettyä koodia. Tärkeimpänä tavoitteena on tehdä uusien funktioiden lisääminen kieleen hyvin helpoksi C++-kielen kautta. Käyttäjien tulisi lisätä funktioita pelimoottorista, pelistä sekä mahdollisesti DLL-linkityksen kautta. Näiden lisäksi etsitään myös mahdollisia optimointistrategioita kääntämiseen sekä kielen suoritukseen. Varsinaiset kääntäjäoptimoinnit jäävät silti vähälle, sillä aiheena se on liian laaja opinnäytetyöhön.

Aihe on varsin laaja, joten tavoitteena ei ole toteuttaa täydellistä ja tehokasta kieltä, vaan toimiva konsepti, jota olisi helppo jatkokehittää tulevaisuudessa. Tämän takia virtuaalikone suorittaa vain bittikoodia ja konekieleen kääntäminen jätetään pois suunnitelmasta. On myös tärkeää priorisoida toimivan prototyypin kasaaminen sekä funktioiden linkityksen tarvitseman systeemin luominen. Koska funktioiden linkitys on yksi tärkeimmistä syistä projektin kielen kehittämiseen, sen täytyy toimia tehokkaasti ja helppokäyttöisesti.

Työn viimeinen vaihe eli testaus tehdään luomalla pieni interaktiivinen demo, jossa käytetään mahdollisimman monia skriptikielen ominaisuuksista. Demo renderöidään Windows-konsoliin ja pelaaja pystyy antamaan syötettä ja ohjaamaan sillä demoa. Demon tulee myös käyttää funktioiden sitomista C++-puolelta, mutta suurin osa koodista on skriptin puolella. Todellisuudessa skriptin puolella tulisi olla vain minimaalinen määrä koodia, mutta demossa täytyy olla enemmän, jotta kielen toimivuus tulee kunnolla testatuksi.

2 Skriptikielien historiaa

Nykyään tunnettuja ja paljon käytettyjä ohjelmointikieliä on kymmenittäin, toisin kuin tietokoneiden alkuaikoina. Nämä kielet ovat usein saaneet alkunsa yksittäisistä tarpeista, joita on ilmestynyt eteen työskenneltäessä ja joista on lähdetty luomaan tapaa korjata puute sen hetkisessä kielessä. Esimerkiksi C-kieli sai alkunsa, kun B-kieli ei ollut enää tarpeeksi joustava ja nopea kehittäjien kasvaviin tarpeisiin. Kuten Dennis M. Ritchie [1] kertoo artikkelissaan ”The Development of the C Language”, B-kielen rajoitteet tulivat nopeasti vastaan tehokkuudessa tietyillä tietokonemalleilla, joten hän alkoi kehittää seuraajaa, joka pystyisi kilpailemaan tehokkuudessa Assembly-kielen kanssa.

Näiden yksittäisien tarpeiden takia useimmat kielet ovat hyviä erityisissä tehtävissä, ja vain suurimmat ja eniten käytetyt kielet on kehitetty toimimaan universaalisti kaikessa, jolloin niistä syntyy niin kutsuttuja yleiskäyttöisiä kieliä. Näiden kielien kehitys on kuitenkin raskasta, ja usein ne joutuvat uhraamaan joustavuutta ja ominaisuuksia ollakseen suorituskykyisiä ja hyödyllisiä useissa tilanteissa. Tästä syystä on kehitetty useita täsmäkieliä, mukaan lukien skriptikieliä, joita käytetään yksinkertaistamaan tarvittavia tehtäviä.

Ohjelmointikielet ovat siis tarkoitettu ratkaisemaan ongelmia itsenäisinä yksikköinä, joten usein niillä on mahdollisista luoda kokonaisia ohjelmia, jotka on mahdollista suorittaa tietokoneissa ilman erillisiä ohjelmia. Sen sijaan skriptikielien tarkoitus on usein laajentaa toisen ohjelman toimintaa helposti muokattavilla koodipalasilla, ja kielen toiminta riippuu vahvasti sen integraatiosta kyseisen ohjelman kanssa. Tästä hyvä esimerkki on kieli JavaScript, joka alun perin kehitettiin edistämään ja helpottamaan verkkosivujen kehitystä yksinkertaisten skriptien avulla selaimien sisällä. Myöhemmin kielestä kuitenkin kehittyi yleiskäyttöinen. [2.]

Skriptikielillä on erityinen asema pelikehityksessä. Kieliä, kuten UnrealScript ja Lua, on käytetty vuosia tukemaan pelien kehitystä ja mahdollistamaan nopea kehitystahti. Niiden avulla pelimoottorien sisäistä toimintaa avataan ulkopuolisille kehittäjille, kuten taiteilijoille ja kenttäsuunnittelijoille. UnrealScriptin tapauksessa kieli on myös suunniteltu tarkasti pelinkehitys mielessä ja sisältää siksi kaikki tärkeimmät ominaisuudet sisäänrakennettuna [3]. Jotta skriptikieli olisi hyödyllinen, sen tulee sekä täyttää tarkka tehtävä että olla helppokäyttöinen ja yksinkertainen laajentaa.

3 Ohjelmointikielien luokituksia

3.1 Yleiskäyttöiset kielet verrattuna täsmäkieliin

Yleiskäyttöiset kielet (General-Purpose Languages, GPL) ovat lähtökohtaisesti kykeneviä ratkaisemaan minkä tahansa ongelman ja ovat siksi niin kutsuttuja Turing-complete-kieliä. Nämä kielet tarjoavat työkaluja moneen eri työtehtävään, mutta eivät välttämättä sisällä kaikkea tarpeellista, vaan käyttäjän tulee itse yhdistää matalamman tason koodia luodakseen tarvittavan toiminnallisuuden. [4.]

Täsmäkielet (Domain-Specific Languages, DSL) on sen sijaan kehitetty hoitamaan yksittäisiä ja tarkkoja ongelmia, joihin yleiskäyttöset kielet eivät aina ole optimaalisia. Usein DSL-tyyppisten kielien rakenne kuvaa ratkaistavaa ongelmaa ja kielellä ei siksi voi tehdä muuta kuin tarvittavat asiat. Yksinkertaisuus ja tarkkuus auttavat pitämään kieltä varmatoimintaisena ja ehkäisemään usein myös ylimääräiset käyttäjävirheet. Yleiskäyttöisillä kielillä on helppo kirjoittaa virheellistä koodia, joka helposti rikkoo ohjelman toiminnan, mutta täsmäkielillä pystyy rajoittamaan käyttäjän mahdollisuuksia ja ennakoimaan mahdollisia virheitä. Tästä on eritysesti hyötyä pelimoottoreissa, joissa artistit tai kenttäsuunnittelijat voivat muokata pelin skriptejä ilman ohjelmoijan valvontaa. [4.]

Esimerkkejä täsmäkielistä ovat GML-skriptikieli GameMaker-pelimoottorissa tai UnrealScript vanhoissa Unreal Enginen versioissa. Myös grafiikan tuottamiseen tarkoitetut kielet, kuten GLSL ja HLSL, voidaan laskea täsmäkieliksi, sillä niitä voidaan käyttää vain GPU:n ohjaamiseen.

3.2 Korkea- ja matalatasoiset kielet

Matalatasoisilla kielillä tarkoitetaan ohjelmointikieliä, jotka tarjoavat hyvin vähän, jos ollenkaan, abstraktiota puhtaan konekielen päälle. Esimerkki tästä ovat Assembly-kielet, jotka ovat ihmisen luettavia versiota esittää konekieltä, mutta vaativat kääntämisen laitteen lukemaan binäärimuotoon. Korkean tason kielet ovat sen sijaan yksinkertaisia käyttää ja hyvin monipuolisia. Ne tarjoavat lisää abstraktiota laitteiden päälle, jotta kehittäjien ei tarvitse huolehtia itse laitteesta, jolle ohjelmaa kehitetään. [5.]

Korkeatasoisten kielien voi käsittää olevan vain syntaksi, jolla tietokoneen toimintaa kuvataan. Kirjoitetun koodin toiminta syntyy vasta, kun kyseisen kielen kääntäjä tai tulkitsija tulkitsee koodin ja suorittaa sen. Tarkka toiminta voi vaihdella eri ympäristöjen ja laitteiden välillä, mutta yleisesti toiminta on tarkasti kuvattu kielen standardissa. Jokaiselle eri ympäristölle, esimerkiksi x86_64- ja ARM-prosessoreille, täytyy luoda omat kääntäjänsä, jotta prosessori voi suorittaa annetut käskyt. Tämä johtuu siitä, että molemmilla ympäristöarkkitehtuureilla on hiukan erilaisia CPU-käskyjä, joilla voi optimoida ja suorittaa ohjelmoijan kirjoittamaa koodia [6].

Tästä syntyy myös ongelma skriptikielien kehittämisen kanssa. Jos kielestä haluaa mahdollisimman tehokkaan, se on käännettävä konekieleen, mutta tämä käännös on ohjelmoitava erikseen jokaiselle mahdolliselle CPU-arkkitehtuurille. Tähän löytyy useita ratkaisuja, ja monesti pienemmissä kielissä, joissa suorituskyky ei ole kriittistä, kääntäminen vältetään kokonaan ja koodi tulkitaan ja suoritetaan toisen korkeatasoisen kielen avulla ohjelmoidun ohjelman kautta.

3.3 Skriptikielet

Skriptikielet ovat edellä mainittujen luokitusten mukaan erittäin korkeatasoisia ja useasti myös täsmäkieliä, joiden yleisenä tehtävänä on laajentaa tai ohjata olemassa olevien ohjelmien toimintaa yhteisen rajapinnan kautta. Kielellä voi esimerkiksi yhdistää useita eri kielillä kirjoitettuja sovelluksia saman rajapinnan alle tai käyttää samoja skriptejä useille erillisille sovelluksille. [7.]

Jotta skriptikieli olisi hyödyllinen, sen tulee täyttää useita vaatimuksia, joista tärkein on itse kielen käyttötarkoitus. Jotta kielen valitsisi toisen sijaan, sen tulee olla hyödyllinen tietyssä käyttötarkoituksessa. Esimerkiksi skaalautuvan verkkopalvelimen rakentamiseen voisi käyttää Elixir-kieltä, sillä se on suunniteltu olemaan yksinkertaisesti skaalautuva ja tarjoamaan monta ominaisuutta verkkopalvelimille [8], joista molemmat ovat tärkeitä ominaisuuksia tarvittavaan rooliin.

Syntaksin täytyy olla yksinkertainen, mutta samalla mahdollistettava tehokas ohjelmointi. Monimutkaisesta syntaksista ei ole hyötyä, jos se ei täytä tiettyä roolia. Tämän takia monet kielet käyttävät pohjimmiltaan samaa syntaksia, vaikkakin pienillä eroilla. Seurauksena ohjelmoijien on helppo oppia uusi kieli ja siirtyä nopeasti oppimaan sen erityisiä piirteitä. Jos alussa joutuisi oppimaan perusteista asti kaiken, tärkeiden ominaisuuksien rooli voisi helposti jäädä pienemmäksi tehden kielen valitsemisesta lähes turhaa.

Viimeiseksi kielen on oltava myös riittävän tehokas suorittaa. Usein ei ole tarpeellista päästä matalatasoisien kielien suorituskykyyn asti, mutta koodin tulisi silti olla riittävän tehokasta kyseiseen ongelmaan. Mitä nopeammaksi suorituksen saa, sitä enemmän ja useammissa kohteissa kieltä voi käyttää.

Kielen käytössä auttaa myös, jos se on helposti kasvatettavissa. Moni kieli antaa mahdollisuuden lisätä toiminnallisuutta toisen, usein matalatasoisen kielen kautta. Esimerkiksi Pythoniin pystyy lisäämään uusia funktioita ja kirjastoja C:n kautta. Koska skriptikielet eivät yleisesti pääse samaan suoritustehoon kuin matalatasoiset kielet, laskennallisesti vaativat kokonaisuudet voidaan siirtää ennalta käännettyyn muotoon, joka sitten suoritetaan skriptikielen virtuaaliympäristön ulkopuolella.

4 Kääntäjät ja tulkit

Ohjelmointikielet jaetaan karkeasti kahteen luokkaan sen mukaan, miten ne yleisesti suoritetaan tietokoneilla. Käännettävät kielet kirjaimellisesti käännetään konekieleen ennen suoritusta ja vain konekielinen data, Windows-järjestelmissä EXE-tiedosto, jaetaan käyttäjille. Kääntäjä huolehtii kielen toiminnasta ja varoittaa virheistä etukäteen, ja valmiin tiedoston voi suorittaa kääntäjän määräämässä ympäristössä. Tätä kutsutaan Ahead-of-Time-kääntämiseksi (AOT), koska kääntämisprosessi tapahtuu vain kerran ennen ohjelman suoritusta. Kielet, kuten C ja C++, käyttävät tätä kääntämisstrategiaa [9]. Toinen vaihtoehto on ohjelman tulkitseminen, jossa ohjelman koodi suoritetaan välittömästi kääntämisen jälkeen virtuaaliympäristön avulla ja ohjelman koodi jaetaan suoraan käyttäjän kanssa. Toiminnallisesti molemmat lähestymistavat ovat hyvin samankaltaisia, mutta tärkein ero tulee dynaamisen ja staattisen linkityksen välillä.

4.1 Staattinen ja dynaaminen linkitys

Käännetyt kielet, kuten C++, käyttävät enimmäkseen staattista linkitystä. Siinä kääntäjä päättää symbolien muistiosoitteet kääntämisen aikana, eikä niitä usein voi muuttaa ajonaikaisesti. Tämä parantaa suorituskykyä, sillä suoritettaessa ei tarvitse käyttää aikaa oikean osoitteen etsimiseen symbolille. Kääntäjä voi myös optimoida muuttujien käyttöä ja jättää turhaa koodia kääntämättä. Heikkoutena staattisessa linkityksessä on se, että valmiin käännetyn ohjelman täytyy sisältää kopio kaikista tarvittavista funktioista, joten sen tiedostokoko ja muistivaatimukset voivat olla suuremmat kuin dynaamisen linkityksen kanssa. [10.]

Dynaamisessa linkityksessä symbolien osoite tai tyyppi ei ole tiedossa kääntämisen aikana, vaan se ratkaistaan suorittaessa. Tämä antaa enemmän vapautta käyttäjälle valita ja muuttaa muuttujien ja funktioiden arvoja ajonaikaisesti. Se mahdollistaa myös koodin kääntämisen, vaikka kaikkia kutsuttuja funktioita ei olisi vielä määritetty. Seurauksena koodin suoritus on kuitenkin hitaampaa, sillä kääntäjä ei voi optimoida tuntemattomien muuttujien ja funktioiden käyttöä, vaan joutuu olettamaan, että ohjelmoija on kirjoittanut tehokasta koodia. Dynaaminen linkitys on yleensä mahdollista myös käännetyissä kielissä. [10.]

Staattisella ja dynaamisella linkityksellä voidaan tarkoittaa myös tulkittujen kielien symbolien etsintäkeinoja. Esimerkiksi skriptikielissä voi olla mahdollista kutsua tuntematonta funktiota tai muuttujaa, jolloin virtuaaliympäristö yrittää dynaamisesti löytää sen osoitteen.

4.2 Ahead-Of-Time-kääntäminen

AOT-tekniikkaa käyttävä kääntäjä koostuu usein kolmesta vaiheesta: etuosa, keskiosa ja jälkiosa [11]. Etuosa vaihtelee kielen mukaan, mutta usein sen ensimmäinen vaihe on esisuoritin, joka järjestää koodin loogiseen muotoon poistaen ylimääräiset rivinvaihdot ja välilyönnit sekä ajaa mahdolliset käyttäjän antamat komennot. Seuraava vaihe on tokenisoija eli lekseri, joka lukee jokaisen merkin koodissa ja muuttaa ne kielen asettamiksi tokeneiksi. Samalla löydetyistä symboleista tallennetaan taulukkoon perustiedot, jotta kääntäjä voi suorittaa staattisen linkityksen myöhemmässä vaiheessa. Seuraavaksi nämä tokenit järjestetään kielen syntaksia vastaavaan puurakenteeseen, josta tunnistetaan mahdolliset ongelmat tietotyyppien tai syntaksin kanssa. Keskiosassa tämä puu analysoidaan ja optimoidaan mahdollisimman nopeaksi käyttäen useita strategioita. Joissain tapauksissa puu käännetään vielä välikieliseksi ohjelmaksi, mikä helpottaa lopullista generointia. Staattinen linkitys tehdään puuta luotaessa. Viimeiseksi jälkiosassa lopullinen puu tai välikieli käännetään konekielisiin komentoihin, joista luodaan lopullinen ajettava ohjelma. Lisäksi kääntäjä laskee ja allokoi rekisterit, pyrkien optimoimaan tarvittavan rekisterimäärän jokaiselle koodiketjulle, jotta rekistereitä voidaan säästää suorituksen tarvitsemille staattisille arvoille nopeuttamaan vakioarvojen hakua.

4.3 Tulkit

Toinen yläluokka kielille on tulkitut kielet. Tulkituissa kielissä itse koodin mukana jaetaan ympäristöä vastaava suoritusympäristö, joka huolehtii koodin suorittamisesta. Kun koodi suoritetaan suoritusympäristössä, se usein käännetään ensin välimuotoon, joka sitten suoritetaan ympäristössä. Komentojen tulkinta vaihtelee käytänteeltään, mutta monesti prosessi ei eroa paljon kääntämisestä. Tulkitun kielen kääntäminen välimuotoon seuraa kääntämisen ensimmäisiä vaiheita, mutta välikoodia ei käännetä konekieleen ja monesti mitään ei tallenneta levylle. Kääntämiseen verrattuna tulkitsemisen täytyy olla erittäin nopeaa, koska ohjelma suoritetaan välittömästi käännöksen jälkeen. Tämän takia optimointia ei voi tehdä niin tehokkaasti kuin käännettäessä [12].

4.4 Just-In-Time-kääntäminen

Just-in-Time-kääntäjät (JIT) ovat kääntäjän ja tulkitsijan välimuoto, jossa koodi yleisesti tulkitaan, mutta usein käytetyt koodipolut käännetään välikielestä konekieleen mahdollistamaan hyvä suorituskyky. Koska kääntäminen tehdään ajonaikaisesti, kääntäjällä on tieto ohjelman aikaisemmasta suorituksesta ja tilasta, joten se pystyy optimoimaan tulevia komentoja paremmin kuin perinteiset tulkit [13]. Se pystyy esimerkiksi huomaamaan turhia koodipolkuja, ylimääräisiä silmukoita ja yksinkertaistusmahdollisuuksia if-lauseissa. Esimerkkejä tätä teknologiaa hyödyntävistä kielistä ja ympäristöistä ovat C# ja Java Virtual Machine (JVM).

Käytännössä JIT-kääntäminen toimii siten, että bittikoodin kääntämisen jälkeen sen kääntämistä jatketaan ajon aikana konekieleen asti [14]. Bittikoodin rekisterit vaihdetaan CPU-rekistereihin ja käskyt vaihdetaan vastaamaan kyseisen alustan käskysettiä. Kääntämisen jälkeen virtuaalikone voi bittikoodin suorituksen sijaan antaa suorittaa puhdasta konekielistä koodia CPU:lla. Koska konekielinen koodi on huomattavasti nopeampaa kuin bittikoodi, kääntämisessä käytetty ylimääräinen aika saadaan hyvin nopeasti kiinni.

4.5 Jäsentäjän toiminta kääntämisessä

Jäsentäjän, tai parserin, tehtävä on järjestää alkuperäismuotoinen lähdekoodi välimuotoon valmiiksi suoritettavan koodin generointia varten. Yleinen välimuoto tässä vaiheessa on abstrakti syntaksipuu (AST, Abstract Syntax Tree), joka sitten suoritetaan tai käännetään uuteen välimuotoon lähemmäksi konekieltä. Jotkin parserit kääntävät lähdekoodin suoraan bittikoodia vastaavaan välimuotoon, mutta ne ovat vaikeampia toteuttaa ja optimoida.

Parserityyppejä on monenlaisia. Useimmat suurikokoiset kielet käyttävät tarkasti muokattuja parsereita juuri niiden tarkoituksiin, mutta pienet kielet ovat usein riippuvaisia valmiista parserigeneraattorista, joista esimerkejä ovat ANTLR, Bison ja YACC [15] [16]. Näille generaattoreille annetaan tarkasti määritelty kielioppi, joiden pohjalta ne generoivat ohjelman, joka voi muuttaa kielioppia käyttäen kirjoitetun tekstin ohjelmien ymmärtämään ja helposti käännettävään muotoon. Jokaisella voi olla hieman erilainen tapa ilmaista kielioppia ja erilaiset käännösmenetelmät, mutta lopputuloksena kaikilla on solmupohjainen puurakenne.

Näiden parserigeneraattorien käyttäminen on vahvasti suositeltua, koska ne nopeuttavat kehitystä ja antavat paljon luotettavamman ja tehokkaamman pohjan kuin käsin kirjoitettu generaattori tai parseri. Tässä työssä kuitenkin käsitellään myös generaattorin kirjoittaminen ja toiminta oppimismielessä.

Lähes kaikki parserit menetelmästä riippumatta luodaan käyttämällä kielioppia. Kieliopissa määritellään kaikki tarvittavat säännöt, tai tuotannot, kielen lukemiseen sekä roolit jokaiselle sen sisältämälle tokenille. Tokenit jaetaan kahteen kategoriaan: terminaaleihin ja epäterminaaleihin. Terminaalit ovat lopullisia merkkejä, joita lekseri tuottaa ja jokainen niistä vastaa merkkijonoa lähdekoodissa. Epäterminaalit sen sijaan ovat kieliopin määrittämiä tokeneita, jotka koostuvat yhdestä tai useammasta terminaalista ja epäterminaalista. Epäterminaaleja käytetään luomaan abstrakteja solmuja puurakenteen kasaamiseen ja lopulta kääntäjän toiminnan ohjaamiseen.

Parserit ja kieliopit voi luokitella niiden käyttämän ennakoinnin perusteella. Ennakoinnilla tässä kontekstissa tarkoitetaan merkkimäärää, jonka parseri katsoo eteenpäin lekserin antamassa tokenijonossa. Kieliopin käyttämä ennakointi ilmaistaan rakenteen mukaan suluissa, esimerkiksi LL(x), jossa x on käytetty ennakoinnin määrä. Yksinkertaisimmat kieliopit voi käsitellä käyttämättä yhtään ennakointia, mutta monet rakenteet vaativat vähintään yhden tokenin ennakointia. Toinen ääripää löytyy esimerkiksi C++:sta, joka vaatii rajattoman ennakoinnin tiettyihin rakenteisiin [17]. Kieliopit, jotka voidaan ilmaista rajallisella ennakoinnilla, voidaan yleensä esittää muodossa, joka käyttää yhden merkin ennakointia.

Epäselvällä kieliopilla tarkoitetaan sääntöjä, joita ei voi ilmaista valitulla rakenteella tai ennakoinnilla yksiselitteisesti, vaan sille generoitaisiin kaksi tai useampi rakenne. Monet kielien säännöistä ovat epäselviä, mutta ne ratkaistaan generointivaiheessa käyttämällä sääntöjen edeltävyyttä tai prioriteettia.

LL(x)-parserit ovat niin kutsuttuja “Left to right, Leftmost first” -parsereita (Vasemmalta oikealle, Vasen ensin). Ne lukevat tokenivirtaa vasemmalta oikealle ja luovat solmuja aina, kun kohtaavat uuden tuotannon alun. Tätä kutsutaan myös top down -jäsennykseksi, sillä parseri aloittaa korkeimmasta tuotannosta ja lisää uusia solmuja aina vanhempiin.

LR(x)-parserit ovat sen sijaan “Left to right, Rightmost first (Vasemmalta oikealle, Oikea ensin)”. Yleisin on LR(1). Ne lukevat tokenivirtaa ja soveltavat jokaiseen tokeniin sääntöä erityisestä LR-taulusta. Tämä taulu sisältää mahdollisia tiloja, kontekseja, joissa parseri voi olla kyseisellä hetkellä. Jokainen solu sisältää joko accept-, error-, shift-, tai reduce-toiminnon. Shift-toiminto siirtää nykyisen merkin merkkipinoon ja lisää uuden tilan tilapinoon. Tilapinon päällimmäinen merkki on

aina nykyinen tila, jota käytetään lukemaan LR-taulua. Kun parseri törmää reduce-toimintoon, se poistaa reduce-säännön määrittämän määrän merkkejä merkki- sekä tilapinosta ja lisää merkkipinoon kyseisen tuotannon tuottaman epäterminaalin. Parseri lopettaa jäsentämisen, jos se törmää accept-sääntöön LR-taulua luettaessa. Tässä tapauksessa merkkipinon tulisi sisältää vain korkeimman säännön epäterminaali. Tämä epäterminaali on toisin sanoen AST-rakenteen juuri. LR-parserit ovat bottom up, koska ne luovat uusia solmuja pohjalta käsin ja lisäävät uusiin solmuihin vanhoja. Valmiisiin solmuihin ei siis enää vierailla. [18.]

Molemmista parseritekniikoista on myös muunnelmia, jotka parantavat toimintaa tietyillä osa-alueilla. Esimerkiksi vaikka LR-parserit ovat nopeita, LR-taulut voivat nopeasti kasvaa erittäin suuriksi. Tätä varten on kehitetty LALR, joka toimii muuten identtisesti, mutta sen taulu on huomattavasti pienempiä [19]. Tämä projekti käyttää muunneltua LALR(1)-parserigeneraattoria assosioituvuus-, prioriteetti- ja optimointisääntöjen kanssa.

4.6 Tyypitys

Tyypitys on yksi suurimmista kääntämiseen liittyvistä aihealueista, joten sen käsittely jätetään vähälle tässä työssä. Tyypityksellä tarkoitetaan muuttujien ja muiden koodin osien datatyyppien selvitystä ja sovittamista. Tyypitetyt kielet ovat tarkkoja siitä, millaisia tyyppejä eri konteksteissa käytetään. Esimerkiksi C++ ei anna käyttäjän asettaa merkkijonoa float-tyyppiseen muuttujaan ilman erityistä tyyppimuunnosta välissä [20]. Kielen tyypitys generoinnin aikana pitää huolen siitä, että tällaisia tilanteita ei pääse käännöksestä läpi.

Tyypitys on siis tärkeä osa kääntämistä turvaamaan koodin oikeus ja varmistamaan ympäristön jatkuva toiminta. Vaikka kieli olisi käyttäjän puolelle tyypitön tai dynaamisesti tyypitetty, kääntäjä kuitenkin tekee tyypitystä tietyissä määrin varmistamaan optimaalinen suorituskyky lisäämällä tyyppimuunnoksia tarvittuihin kohtiin koodia.

Todellisuudessa tyypityksen aikana koodi käydään läpi ja kääntäjä varmistaa, että se tietää jokaisen käskyn käyttämät ja tuottamat tyypit ja että se voi tehdä tarvittavat muutokset niiden välillä. Kieleen voidaan lisätä tyypityssääntöjä, joita kääntäjä käyttää tarkistamaan mahdollisia muunnoksia tai yhteensopivia tyyppejä. Tämä tyypitys voidaan toteuttaa joko kääntämisen aikana mahdollisimman hyvän suorituskyvyn mahdollistamiseksi tai ajonaikaisesti, jos suorituskyky ei ole kriittinen tai kielen ominaisuudet vaativat sitä. [21.]

4.7 Välikielen ja konekielen erot

Välikielet ja konekielet ovat kaksi hyvin eri asiaa, vaikka ne ajavatkin samaa asiaa ohjelmien suorituksessa. Välikielet ovat eri kielien määrittämiä tapoja ilmaista ohjelman rakenne tiiviissä muodossa mahdollisimman lähellä konekieliä. Ne ovat lähes aina bitti- tai puupohjaisia rakenteita, joita käytetään joko konekielen generoimiseen tai ohjelman suoritukseen virtuaaliympäristössä [22]. Ympäristö määrittää jokaiselle bittiarvolle käskyn, joka suoritetaan, kun arvo tulee vastaan. Käskybittiä seuraa mahdollisia arvoja, joita käsky tarvitsee suoritukseen. Esimerkiksi pinopohjaisessa virtuaaliympäristössä komento PUSH voi työntää käskyä seuraavan bittisarjan pinoon, jotta seuraava komento ADD voi ottaa kaksi edellistä arvoa pinosta, lisätä ne yhteen ja työntää tuloksen pinoon.

Konekieli sen sijaan sisältää CPU-arkkitehtuurin määrittämiä käskyjä, jotka suoritetaan suoraan CPU:lla. Tarkat käskyt riippuvat suoraan CPU:n arkkitehtuurista ja ohjelma täytyy kääntää tarkasti oikealle ympäristölle, jotta sen voi suorittaa. Konekieliset käskyt käsittelevät suoraan rekistereitä ja muistiosoitteita, toisin kuin välikielet.

4.8 Virtuaaliympäristöt

Virtuaaliympäristöjä (virtuaalikone, VM) käytetään tulkittujen kielien suorituksessa. Ne simuloivat CPU:n toimintaa ja toteuttavat sen omalla käskysetillään, jossa jokainen käsky vastaa jotain konekielisen koodin osoitetta. Ne myös pitävät tiedon käännetyistä symboleista ja muuttujien ja funktioiden osoitteista. Yksinkertaistetusti virtuaalikone suorittaa silmukkaa, jossa bittikoodia luetaan bitti kerrallaan ja bitin vastaama käsky suoritetaan. Virtuaalikone pitää huolen, että tarvittavat funktiot ja muuttujat löytyvät symbolitaulukosta ja lataa ne tarvittaessa, mikäli niiden sijainti levyllä on tiedossa. [23.]

Muistinhallinta on tärkeä syy virtuaalikoneen käyttöön. Kone pitää sisällään tiedot allokoiduista objekteista ja käyttää omia allokaattoreitaan uusien luomiseen välttäen lähes kokonaan käyttöjärjestelmätason muistinvarauksen. Muistin pyytäminen käyttöjärjestelmältä on hitaampaa kuin jo valmiiksi varatun muistin käyttö, joten sen välttäminen on usein kannattavaa. Erityisesti pienet, usein käytetyt allokoinnit hyötyvät tästä.

Suorituksen säikeistys on mahdollista, kun virtuaalikone suorittaa bittikoodia useammalla säikeellä. Jokaiselle säikeelle varataan niiden oma pinomuisti tai rekisterialue, joten usean funktion

suorittaminen ei itsessään aiheuta vaikeuksia synkronisoinnin kanssa. Mikäli funktiot käyttävät suorituksessaan julkisia ja globaaleja muuttujia, virtuaalikoneen tulee pitää huolta mahdollisten kilpatilanteiden ratkaisusta.

4.8.1 Pinopohjaiset virtuaalikoneet

Helpoin virtuaalikoneiden rakenne on pinopohjainen. Siinä jokainen virtuaalikoneen käsky muokkaa suoraan pinon päällimmäisiä arvoja eikä aikaisempiin arvoihin juuri kosketa. Esimerkiksi jos käskynä olisi lisätä kaksi numeroa yhteen, käskyjono ensin puskee kaksi arvoa pinoon ja yhteenlaskun käsky poistaa kaksi päällimmäistä arvoa, laskee ne yhteen ja puskee tuloksen pinoon [24]. Tämä kuitenkin aina optimoidaan niin, että vain yksi arvo poistetaan pinosta ja tulos ylikirjoittaa toisen. Esimerkki tämän kaltaisen operaation käskysarjasta avataan Kuva 1.

Puhtaat pinokoneet eivät koskaan lue arvoja pinon keskeltä, mutta ne eivät ole erityisen käytännöllisiä kieliin, jotka vaativat paikallisia muuttujia. Sen sijaan useimmat pinokoneet voivat lukea arvoja käyttämällä etäisyyttä pinon pohjalta, kuten tapahtuu käskyssä LOAD 0, Kuva 1. Paikallisista muuttujista tulee siten indeksejä pinossa, josta ne voi sitten kopioida pinon päälle operaatioita varten.

```
var x = 10 + 5 * 2;
4 + 5;
var y = x + 10;
```

Käsky	Pinon ylin arvo
PUSH	10
PUSH	5
PUSH	2
MUL	10
ADD	20
STORE 0	
PUSH	4
PUSH	5
ADD	9
POP	
LOAD 0	20
PUSH	10
ADD	30
STORE 1	

Kuva 1 Koodipala ja sen tuottama bittikoodi

Pinokoneiden vaatima bittikoodi on yleensä suurempikokoista kuin rekisterikoneiden. Vaikka jokainen käsky ei vaadi parametrejä, käskyjen määrä on useasti moninkertainen, varsinkin peräkkäisten operaatioiden kohdalla.

Vaikka pinokoneet ovat yksinkertaisia, ne ovat myös hitaita. Jokainen operaatio vaatii pinon muokkaamista ja uusien arvojen kopiointia. Esimerkiksi jokainen muuttuja täytyisi kopioida pinon päälle, jotta sitä voi käyttää operaatioissa. Jatkuva kopiointi ja käskyjen määrä tekee pinokoneista hitaita verrattuna muihin virtuaalikoneiden rakenteisiin. [24.]

4.8.2 Rekisteripohjaiset virtuaalikoneet

Rekisterikoneet mallintavat läheisesti oikeita prosessoreita [24]. Ne käsittelevät arvoja koneen rekistereissä, ja vaikka rekisterit käytännössä toimivat kuin pino, käskyt käsittelevät arvoja vapaasti. Rekisterikoneiden käyttämä bittikoodi on tiivistä ja vaatii vähemmän käskyjä kuin vastaava pinokoodi. Rekisteripohjainen koodi on myös paljon helpompaa muuttaa konekieliseen muotoon JIT-kääntämisellä ajonaikaisesti.

Kuva 2 esittää rekisterikoneen käyttämää bittikoodia. Usein koodia optimoidaan lisää eikä sen tarvitsisi käyttää kolmea osoitetta, vaan operaatiot käyttävät erityistä varattua ”accumulator”-rekisteriä toisena parametrinä ja kohteena. Kuvassa käskyt käyttävät kahta tai kolmea parametria, käskystä riippuen.

```
var x = 10 + 5 * 2;
4 + 5;
var y = x + 10;
```

Käsky	Parametrit		
LOAD INT	0	2	
LOAD INT	1	5	
MUL	0	0	1
LOAD INT	2	10	
ADD	0	0	2
LOAD INT	3	4	
ADD	1	1	3
ADD	1	0	2

Kuva 2 Rekisterikoneen bittikoodi kolmiosoitejärjestelmässä

Koska rekisterikoneet välttävät turhia kopiointeja, ne voivat olla huomattavasti nopeampia kuin pinopohjaiset virtuaalikoneet. Paikalliset muuttujat ovat myös nopeampia ja helpommin käytettäviä. Rekisterit kuitenkin lisäävät monimutkaisuutta, sillä jo käännösvaiheessa täytyy varata rekisterit operaatioille ja pitää huoli, että ne riittävät koko ohjelman suoritukselle. Virtuaalikoneissa on lähes aina käytössä melkein rajaton määrä rekistereitä, mutta todellisissa prosessoreissa on vain hyvin rajallinen määrä. Jos bittikoodia myöhemmin käännetään konekieleen, rekisterien allokointi on erittäin tärkeä aihe.

Allokoinnissa on hyvin monia eri strategioita ja algoritmeja. Yleisesti kääntäjät ja tulkitsijat käyttävät eri strategioita eri vaatimuksien takia, ja näistä yleisimmät ovat lineaarinen ja graafivärjäys. Lineaarinen algoritmi skannaa koodialueen ja kerää datan muuttujien eliniästä [25], minkä jälkeen se varaa rekistereitä ahneesti jokaiselle muuttujien eliniälle. Tätä käytetään toteutuksissa, missä rekisterivaraukseen halutaan käyttää mahdollisimman vähän aikaa, muun muassa monissa JIT-kääntäjissä [26]. Graafivärjäys on taas yleisin rekisterivarausalgoritmi, mutta sen heikkoutena on ylimääräinen varaukseen käytetty aika. Värjäys perustuu puurakenteen rakentamiseen käyttämällä muuttujien elinaikoja, ja tämän puun läpikäynnistä niin, ettei yhdelläkään yhdistyneellä solmulla ole samaa ”väriä” [27]. Lopussa solmujen väri merkitsee niille varattua rekisteriä. Graafivärjäys on kompleksi algoritmi ja se ei ole oleellinen työn kannalta, joten sen toteutusta ei avata tässä. Työssä rakennettu kääntäjä käyttää muunnelmaa lineaarisesta rekisterivarauksesta.

Allokoinnissa voidaan myös käyttää useita optimointeja. Esimerkiksi jokainen muuttujan asetus käytännössä luo uuden muuttujan ja vanha rekisteri voidaan vapauttaa toiseen käyttöön viimeisen lukukerran jälkeen [28]. Tätä kutsutaan SSA-muodoksi. Jokaisella paikallisella muuttujalla on tietty elinikä, jonka aikana sitä käytetään. Elinikä alkaa muuttujan määrittelemisestä ja päättyy viimeiseen lukuun ennen uuden arvon asetusta siihen. Edellisen luvun ja seuraavan asetuksen välillä voi kulua pitkä aika, joten se kannattaa optimoida pois mahdollisuuksien mukaan.

5 Vaatimusmäärittely

Vaatimusmäärittelyssä määritellään projektin tavoitteet ja EMI-kieleen halutut ominaisuudet. Monesti sen tavoitteena on rajata projektin käsittelemää aluetta, mutta tässä projektissa määritellään alussa vain pakolliset ja toivotut ominaisuudet kieleen. Vaaditut ominaisuudet sisältävät kaiken, mitä on toteutettava projektissa, jotta lopputulos on hyväksyttävä ja käyttökelpoinen. Lisäominaisuuksissa listataan sen sijaan valinnaisia ja vähemmän kriittisiä ominaisuuksia, jotka kuitenkin parantavat kielen käytettävyyttä ja hyödyllisyyttä merkittävästi.

5.1 Vaaditut ominaisuudet

EMI-kieli on tyypitykseltään vapaa, mutta tyypitys on myös mahdollista. Sisäisesti virtuaalikone tekee tyypityksiä muuttujille ja funktioille, jotta suoritettavan bittikoodin voisi optimoida mahdollisimman hyvin. Kieli ei erottele kokonais- ja liukulukujen välillä, vaan kaikki käsitellään numeroina. Tämän takia datatyyppejä ovat vain numero, totuusarvo, teksti, funktio ja objektireferenssi. Käytännössä teksti, funktiot, taulukot ja muut vastaavat käsitellään objekteina.

Kieli tarvitsee kaikki perinteiset suorituksen kontrollirakenteet, joihin kuuluvat if-, for- ja while-silmukat. Erilaiset for-silmukan muodot, kuten rajattu alue, olisi myös hyvä lisätä syntaksiin, mutta perinteinen C++-versio riittää aluksi. Silmukoiden mukana kielen täytyy tukea eri avainsanoja, kuten C++:n käyttämät break, continue, ja return. Break pakenee sisimmästä silmukasta, continue palaa silmukan alkuun ja return palaa alkuperäiseen kutsuvaan funktioon. For-silmukka tukee myös else-lohkoa, joka suoritetaan aina kun for-silmukasta poistutaan käyttämällä break-avainsanaa.

Perinteinen if-lause suorittaa annetun lausekkeen ja tuloksen totuuden mukaan päättää seuraavien lohkojen suorituksesta. Sen tulee myös tukea else-lohkoa, joka toteutetaan, mikäli annettu ehto ei toteudu. Vaihtoehtoinen muoto if-lauseesta on niin kutsuttu “ternary”, joka toimii muuten täysin samalla tavalla, mutta käyttää syntaksia:

$$ehto\ ?\ (tosi)\ :\ (else - lohko)$$

Aritmetiikka on seuraava tärkeä elementti kielen käytettävyyden kannalta. Kielellä tulee olla yksinkertaista tehdä aritmetiikkaa vähintään kokonaisluvuilla ja liukuluvuilla. Mikäli mahdollista

ajan kanssa, myös vektorityypit olisi hyvä lisätä natiivilla tuella, sillä kaksi- ja kolmiulotteiset vektorit ovat erittäin tärkeitä peleissä käytettäville kielille. Varsinkin näiden suorituskyky on tärkeää. Aritmetiikan lisäksi totuusarvot ovat pakollisia. Perinteiset totuusarvo-operaattorit on lisättävä kieleen. Näihin kuuluvat esimerkiksi and-, or-, ja not-operaattorit.

Pelit tarvitsevat usein myös merkkijonoja ja tekstiä. Vaikka tekstinkäsittely ei ole niin tärkeää kuin aritmetiikka, se lisää huomattavasti käytettävyyttä kielelle. Tekstiä pitää tukea virtuaalikoneen tasolla, mutta tekstinkäsittelyyn käytettävät funktiot voidaan toteuttaa standardikirjaston tyylisesti myöhemmässä kehityksen vaiheessa.

Funktiokutsut ovat tärkein osa skriptikieliä ja siksi niiden toiminta on myös yksi tärkeimmistä kehityskohteista tässä projektissa. Funktioita täytyy pystyä lisäämään sekä itse skriptikielessä että dynaamisesti sen ulkopuolelta. Kielen funktioita pitäisi pystyä myös kutsumaan ulkopuolelta käsin. Funktiot voivat ottaa teoriassa rajattoman määrän argumentteja, mutta käytännössä tämä määrä tulee rajoittumaan toteutuksen myötä. Argumenttien tyypillä ei pitäisi olla merkitystä, mutta funktiota ei kutsuta, jos tyyppi ei sovi yhteen käyttäjän määrittämän argumenttityypin kanssa. C++-puolelta lisätyt funktiot voivat käyttää vain perinteisiä datatyyppejä.

Funktiot voi myös käsitellä arvoina skriptien sisällä. Koska kyseessä on pitkälti funktionaalinen kieli, käyttäjän täytyy kyetä käsittelemään funktioita arvoina ja esimerkiksi asettamaan niitä muuttujiin tai käyttämään niitä funktioiden argumentteina.

Viimeiset tärkeät lisäykset kielen käyttöön ovat taulukot ja mahdollisesti listat ja hajautustaulukot. Erilaiset säiliöt ovat tärkeitä, mutta aluksi riittää, että kieli tukee yksinkertaisia taulukoita. Hajautustaulut, kartat ja listat olisivat hyödyllisiä, mutta niiden lisääminen kieleen voidaan toteuttaa myöhemmin standardikirjaston kautta. Kaikki säiliöt voivat sisältää mitä tahansa arvoja ja tyyppejä.

5.2 Lisäominaisuudet

Lisäominaisuudet ovat ominaisuuksia, jotka hyödyttäisivät kieltä merkittävästi, mutta joiden kehitykseen kuluu paljon aikaa. Nämä ominaisuudet olisi hyvä lisätä kieleen, mikäli kehitykseen jää aikaa:

- Putki-operaattori (|>). Putki eli pipe ottaa vasemmanpuoleisen lausekkeen ja asettaa sen oikeanpuoleisen funktiokutsun ensimmäiseksi argumentiksi. Oikeanpuoleisen osuuden täytyy olla funktiokutsu. Putki-operaattori on lainattu suoraan Elixir-kielestä [29].

- Helppo asynkroninen suoritus. Funktiot voi suorittaa asynkronisesti käyttämällä async-avainsanaa. Tulevaisuudet ja lupaukset toimivat muuttujina ja suoritus pysäytetään vain, jos niiden arvoa kysytään ennen asynkronisen funktion lopetusta. Asynkroninen suoritus on kuitenkin vaikea toteuttaa ja se jätetään pois tämänhetkisestä kielen toteutuksesta.

- Delay, eli viivefunktio, joka pysäyttää funktion suorituksen annetuksi ajaksi. Viive on erittäin hyödyllinen pelien ohjelmoinnissa, mutta vaatii osittaista asynkronista tukea kielelle. Sen lisääminen kieleen riippuu suorituksen toteutuksesta.

- Objektit. Käyttäjät pystyvät määrittämään objekteja JavaScript-tyylisesti, vaikkakin staattisesti, eli niihin ei voi lisätä uusia arvoja ajonaikaisesti. Objektit tässä kontekstissa vastaavat lähinnä C-kielen tietueita, joihin käyttäjä voi määritellä useita muuttujia ja käyttää luotua objektia datatyyppinä muuttujille. Objektit luodaan erillisien muistiallokaattorien kautta ja käyttäjä voi säilyttää vain kahvoja niihin.

- Tagi-systeemi. Ohjelman suorituksen aikana voi asettaa tageja syntaksilla set :tag_name, poistaa tageja käyttämällä unset :tag_name ja tarkistaa onko tagi asetettu käyttämällä isSet :tag_name. Tageja voi myös asettaa muihin muuttujiin ja niitä pystyy vertailemaan yhtäsuuruudella. Näiden tagien arvo on niiden nimi.

5.3 Kielen käyttö

C++-puolella voi lisätä kieleen funktioita joko makrolla tai vastaavalla helpolla tavalla. VM-kahvalla voi myös pyytää funktion kahvaa, jolla voi sitten kutsua pyydettyä funktiota tarvittaessa. Koska funktion etsiminen voi olla hidasta, kahva on hyvä pyytää, jos funktiota kutsutaan useammin.

Vain perinteisiä C-datatyyppejä, kuten bool, int, float ja const char* tuetaan funktioiden argumenteiksi. Käyttäjän ei itse tarvitse listata niitä rekisteröinnin yhteydessä, vaan se jätetään kääntäjän tehtäväksi. Tämä toteutetaan alustavasti C++-mallien ja konseptien avulla.

Jokainen tiedosto on oletuksena oma yksityinen yksikkönsä, mutta käyttämällä avainsanaa extend käyttäjä voi määrittää nimiavaruuden tuleville funktioille. Esimerkiksi jos tiedosto alkaa extend String ja se sisältää julkisen split-funktion, funktioita voi kutsua muista tiedostoista kirjoittamalla String.split(). Yksi tiedosto voi kasvattaa useampaa nimiavaruutta.

Ympäristöön ladataan skriptejä lähdekooditiedostoista, ja käännetyt skriptit voi tallentaa välimuodossa tiedostoihin, joita voi myöhemmin ladata uudestaan. Välimuodon lataaminen virtuaaliympäristöön on huomattavasti nopeampaa kuin kääntäminen, sekä sitä on vaikeampi muokata. Tämä sopii erityisesti peleille, joissa käynnistysnopeus on tärkeää. Välimuotoinen kieli tarvitsee silti käännösrajapinnan, joka muuttaa tiedostoon tallennetut ID-arvot virtuaaliympäristön sisältämiin arvoihin.

Kääntämisen tueksi voi myös kehittää erillisen tiedostomuodon, joka kertoo tarvittavat tiedot skriptikokoelmasta myös silloin kun skriptit ladataan tekstimuodosta. Se sisältäisi vain polut kaikkiin kokoelman tiedostoihin sekä mahdollisesti muuta tärkeää tietoa kääntämisen tueksi. Suurin hyöty tästä tulee, kun peleissä halutaan kerralla ladata tai poistaa useampi skriptitiedosto.

5.4 Kielen syntaksi

Syntaksin kannattaa pitkälti vastata C++- ja C#-kieliä, sillä toinen tai molemmat ovat tuttuja useimmille pelinkehittäjille. Yksi tärkeimmistä skriptikielien eduista on tietysti niiden vapaus syntaksin kanssa, mutta yksinkertainen ja tuttu kieli on yksinkertaisempi toteuttaa projektin aikana. EMI-kielellä ei myöskään ole tarvetta erityiselle syntaksille, vaan se voi noudattaa C- ja C++-kielien syntaksia. Esimerkki syntaksista näkyy kuvassaKuva 3, jossa on funktio taulukoiden luomista varten. Kuvan funktiossa käytetään useita kielen vaatimusmäärittelyssä listatuista ominaisuuksista.

```
def makeArray(size : number, value) : array {
    var out : array;
    Array.Resize(out, size);
    for (out) {
        out[_index_] = Copy(value);
    }
    return out;
}
```

Kuva 3 EMI-kielellä kirjoitettu funktio

Muutoksena C-kielen syntaksiin, EMI käyttää avainsanaa “var” muuttujien luomiseen. Sen lisäksi muuttujan perään voi merkitä halutun datatyypin kaksoispisteen kanssa. Muuttujat alustetaan aina oletusarvoihin, jos käyttäjä ei aseta muuta arvoa alustuksen aikana.

Toinen suuri ero on for-silmukat. EMI tukee C++:n mukaista for-silmukkaa, jossa on alustus, vertailu ja lopetusosat erikseen kirjoitettuina. Sen lisäksi kieli tukee myös kahta muuta muotoa, eli C++ range for -silmukkaa ja numeron yli iterointia. Range for -silmukassa määritetään sääntö muodossa:

$$for\ (muuttuja\ in\ säiliö)\ \{\}$$

Muuttuja on uusi nimi, joka sisältää joka silmukan suorituksella seuraavan säiliön alkioista. Säiliö voi olla taulukko tai numero, mutta niiden toiminta on hieman erilaista. Taulukon kanssa silmukka ajetaan jokaiselle taulukon elementille ja muuttuja on aina kyseisen elementin arvo, kun taas numeron kanssa silmukka ajetaan numeron kertoman määrän ja muuttuja on ajokerran indeksi.

Numeron yli iteroinnissa silmukalle annetaan vain numero joko vakiona tai muuttujan kautta. Numero voidaan myös etsiä erityyppisistä muuttujista. Parhaiten haku toimii taulukoiden kanssa, missä tapauksessa numerona käytetään taulukon kokoa.

$$for\ (numero)\ \{\}$$

Silmukka suoritetaan numeron osoittaman määrän. Käytännössä sama tulos saadaan kirjoittamalla:

$$var\ \_index\_ :\ number\ = 0;\ while\ (\_index\_ < numero)\ \{\ koodi;\ \_index\_ + +;\ \}$$

Molemmissa erikoistetuissa silmukoissa on myös piilotettu automaattinen muuttuja _index_, joka sisältää ajokerran indeksin numerona. Käyttäjä voi käyttää tätä muuttujaa kertomaan nykyisen silmukan suorituksen indeksi ilman manuaalista työtä, kuten Kuva 3.

Totuusarvotestaukset tehdään käyttämällä operaattoreita &&, || ja ==, tai vastaavasti avainsanoja and, or ja is. Avainsanojen käyttäminen merkkien sijaan on yleistä skriptikielissä, mutta ne voivat tietyissä tilanteissa aiheuttaa hämmennystä käyttäjälle. Esimerkiksi käyttäjä voi luulla että ”is not” olisi sama operaatio kuin ”!=”, kun todellisuudessa se ei vertaa epäsuuruutta vaan molempien operandien totuuden eroavuutta. Tästä syystä merkkioperaattoreiden käyttäminen on suositeltua.

Muita huomioita kielen syntaksissa on puolipisteiden ja sulkeiden käyttö. Puolipiste rivin lopussa on pakollinen, mutta sitä ei tarvitse aaltosulkeiden jälkeen. Myöskään lausekkeen kontekstilla ei ole merkitystä puolipisteen käytölle. Sulkeita sen sijaan voi käyttää missä tahansa lausekkeen osassa tai ympärillä, mutta sulkeet nimen tai tiettyjen operaatioiden jälkeen käsitellään funktiokutsuna. Aaltosulkeita voi käyttää paikallisia näkyvyysalueiden luomiseen samoin kuin C++-kielessä.

Objektien alustus tehdään käyttämällä objektin täyttä nimeä ja aaltosulkeita. Aaltosulkeiden sisään voi määrittää pilkulla erotettuja lausekkeita, joiden tulokset asetetaan objektin kenttiin esiintymisjärjestyksessä.

6 Toteutus

Tässä kappaleessa käydään läpi kaikki EMI-kielen toteutuksen vaiheet. Todellinen toteutus ei ole yhtä suoraviivainen, mutta se seuraa samaa järjestystä. Projekti toteutettiin puhtaalla C++-kielellä käyttäen C++20-standardia. Kehitysympäristönä toimi Visual Studio 2022 Windows-käyttöjärjestelmällä ja projektin hallintaan käytettiin CMake-ohjelmaa. Projektin loppupuolella siirryttiin myös Linux-pohjaiselle käyttöjärjestelmälle ja G++-kääntäjälle, jotta kieltä pystyisi käyttämään useammassa ympäristössä.

6.1 Arkkitehtuuri

Arkkitehtuurin suunnittelun aikana suunnitellaan kaikki projektin aikana syntyvät systeemit sekä rakenteellisesti että toiminnallisesti. Tarkka kuva arkkitehtuurista on tärkeä, jotta projekti pysyy yhtenäisenä koko toteutuksen ajan ja kaikki osat toimivat yhdessä kokonaisuudessa.

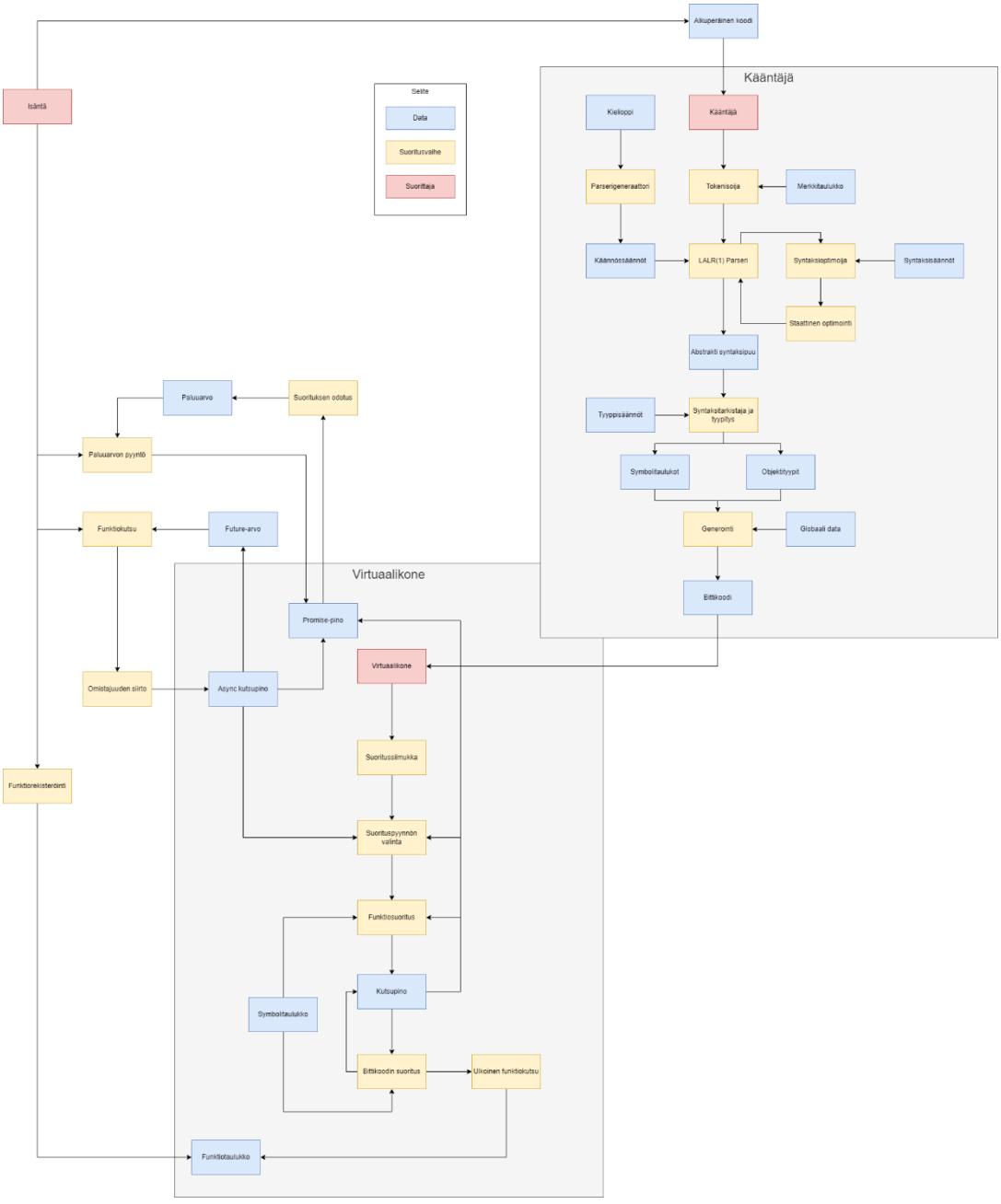
6.1.1 Yleiskuva

Työn tavoitteena on toteuttaa ajonaikaisesti käännetty ja dynaamisesti linkitetty skriptikieli, jolla on yksinkertaista lisätä toiminnallisuutta peleihin. Kielen virtuaalikone toteutetaan rekisteripohjaisena, ja sen kääntäjä tulee käyttämään LALR(1)-kääntämistä.

EMI-virtuaalikone on vahvasti säikeistetty, eli toisin sanoen kääntämiselle ja suoritukselle on varattu useita säikeitä, joista ensimmäinen vapaa suorittaa pyydettyä komentoa. Käyttäjä voi antaa virtuaalikoneelle komentoja C++-käyttöliittymän kautta. Käyttöliittymä toteutetaan alustavasti käyttäen vain C++20-standardia, sillä se tarjoaa parhaat työkalut määritettyihin vaatimuksiin. Tulevaisuudessa toteutus ja rajapinta tulisi muokata sopiviksi vanhempien standardien ja C-standardin mukaiseksi.

Ohjelman kääntämiseksi käyttäjän tulee antaa polku kooditiedostoon. Virtuaalikone antaa tämän polun sitten eteenpäin kääntäjäsäikeelle, joka aloittaa kääntämisen. Kääntämisen jälkeen säie yhdistää tuloksen virtuaalikoneen omaan muistiin synkronisesti. Käyttäjä saa kahvan kääntöprosessiin, jonka kautta on mahdollista odottaa käännöksen valmistumista ja lukea sen onnistuminen. Kääntäjän rakenne esitetään graafisesti kuvassaKuva 4.

Käyttäjä voi myös pyytää funktion suoritusta virtuaalikoneelta, missä tapauksessa komento lisätään kutsupinoon, josta ensimmäinen vapaa suoritussäie ottaa sen suoritettavaksi. Koska suoritus tapahtuu asynkronisesti, komennon lisääminen palauttaa käyttäjälle kahvan tallennettuun std::futureen, jonka kautta palautettu arvo voidaan lukea myöhemmässä kohtaa.



Kuva 4 Projektin toimintarakenne

6.1.2 Käyttöliittymä

C++-käyttöliittymän täytyy käyttää vain C-standardia DLL-tason kommunikoinnissa. Koska C++-kääntäjiä on monia, myös standardikirjaston toteutuksia on useita. Tästä johtuen ei voi luottaa siihen, että kahdella kääntäjällä käännetyt ohjelmat noudattavat samaa muistiasettelua luokille. Kahden käännösyksikön välillä, tässä tapauksessa käyttäjän ohjelman ja virtuaalikoneen DLL:n, tulee käyttää vain yksinkertaisia “plain-old-data”-datatyyppejä. [30.]

EMI-käyttöliittymä tarjoaa siksi abstraktioita ongelmien välttämiseksi. Käyttäjä ei koskaan ole suoraan tekemisissä virtuaalikoneen kanssa, vaan kaikki palautetut arvot ovat vain kahvoja virtuaalikoneen resursseihin. Kaikki DLL-funktiot ottavat vastaan vain näitä kokonaislukuarvoja tai käyttöliittymässä määriteltyjä luokkia, joiden muistirakenne on sama kääntäjästä riippumatta. [31.]

Käyttäjä luo uuden virtuaalikoneen ja saa VMHandle-tyyppisen objektin, jonka kautta virtuaalikoneen funktioita voi kutsua. Käyttöliittymä tarjoaa funktiot skriptien lisäämiseen ja poistamiseen, funktio- ja muuttujakahvojen hakemiseen ja virtuaalikoneen suorituksen hallintaan. Kahvat on toteutettu omina luokkinaan, joita käyttäjä voi käyttää suoraan ilman VMHandle-objektia. Ne kuitenkin viittaavat aina alkuperäiseen virtuaalikoneeseen, eikä niiden osoittamaa kohdetta voi muuttaa.

6.1.3 Virtuaaliympäristöt

Käyttäjä luo uuden virtuaalikoneen tarvittaessa ja lisää sinne skriptejä, joko yhdistetyllä tiedostolla, suoralla lähdekoodilla tai valmiiksi käännetyllä bittikoodilla. Virtuaalikoneet täytyy vapauttaa erikseen, kun niitä ei enää käytetä.

Virtuaalikone pitää tiedon tiedostoista ja niiden sisältämistä funktioista. Jokaisella koneella on oma kekomuisti. Jokaisella koneen säikeellä on oma pinopohjainen rekisterimuisti. Rekisterit on toteutettu pinona, jota kasvatetaan aina, kun funktiot tarvitsevat enemmän rekistereitä kuin on varattu. Tarvitun määrän tietää aina funktiota kutsuttaessa, joten ensimmäiset kutsut kasvattavat rekisterien määrän tarvittuun kokoon.

Ympäristö pitää tiedon, mistä tiedostosta funktiot ovat ja mitkä tiedostot ovat ladattuina sisään. Tätä tietoa voi käyttää myöhemmin tiedostojen poistamiseen virtuaalikoneesta muistin vapauttamiseksi. Tätä tietoa käytetään hyväksi myös myöhemmässä virheenjäljityksessä, jotta käyttäjän olisi helpompi löytää virhetiloja virtuaalikoneen antamien tietojen perusteella.

Kun tiedosto käännetään ja valmistetaan, funktiokutsuille etsitään lähin vastaava funktio-osoite. Jos osoitetta ei löydy, linkkitauluun lisätään tyhjä indeksi paikalliselle kutsulle. Jos tiedosto exportataan, linkkitaulu lisätään sen mukaan. Kun tällainen export ladataan sisään, täytyy käyttää välikerrosta linkittämään paikalliset funktiokutsut virtuaalikoneen funktio-osoitteisiin.

Ympäristöllä on oma säieallas parserille ja virtuaalikoneelle. Molemmille alustetaan säikeitä koneen laitteiston suosiman määrän. Näitä säikeitä voi sitten käyttää tarvittuihin suorituksiin ilman ylimääräistä hidastusta uuden säikeen luomisesta. Jokainen säie odottaa suorituskäskyä, joten ne eivät käytä paljon CPU:n tehosta ollessaan odotustilassa.

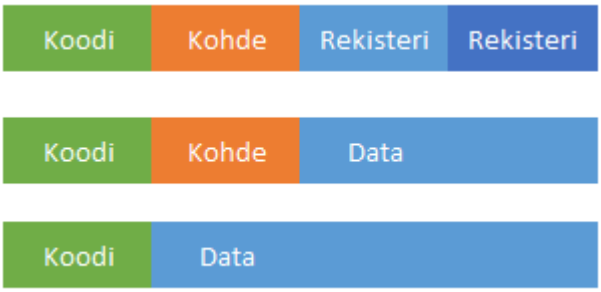
6.1.4 Syntaksi

Kielen syntaksin sääntöjen kirjoittaminen on haastavaa, ja ne kehittyvät jatkuvasti kielen rinnalla. Alkuperäinen syntaksi (katso 5.4) luotiin käyttämällä verkossa toimivaa interaktiivista ANTRL-suunnittelijaa, johon ensin kirjoitetaan haluttu esimerkkikoodi ja sen jälkeen suunnitellaan kielioppi, joka voi kääntää sen halutuksi puuksi. Tällä sai alussa hyvän kuvan kieliopista, mutta lopussa se ei taipunut haluttuun muotoon. Tämän takia kielelle kehitettiin oma kieliopin kääntäjä, joka tulee perinteisen terminaalipohjaisen kieliopin lisäksi ylimääräisiä optimointi-, prioriteetti- ja assosioituvuussääntöjä. Näiden sääntöjen pohjalta pystyy rakentamaan käännöstaulukon, joka avulla on mahdollista rakentaa yksinkertainen ja helposti käsiteltävä syntaksipuu.

Yksi tärkeimmistä toteutuspäätöksistä kehityksen aikana oli kielen tyypityksen määrittäminen. Lopullinen päätös oli heikosti tyypitetty kieli, jota voi käyttää tietyissä määrin vahvasti tyypitettynä. Useasti nopeassa kehityksessä kielen käyttäjä ei suuresti välitä muuttujien tyypeistä, vaan niiden sisältämästä datasta. Tämän takia virtuaalikoneen sallitaan tehdä tiettyjä tyyppimuunnoksia automaattisesti, kuten numerosta tekstiksi ja päinvastoin. Kaikki kehittäjät eivät kuitenkaan pidä tästä, ja se myös heikentää suorituskykyä, joten tulevaisuudessa vahva tyypitys tai vaihtoehto sille olisi suositeltavaa.

6.1.5 Bittikoodin rakenne

Virtuaalikone on rekisteripohjainen, ja bittikoodi seuraa kolmiosoitejärjestelmää [32]. Se tarkoittaa, että jokainen koodin osa on 32 bittiä, joista ensimmäiset kahdeksan kertovat käytettävän käskyn. Loput biteistä käytetään argumenteiksi käskyn määräämällä käytänteellä. Kolmiosoitejärjestelmä tarkoittaa sitä, että jokaiselle operaatiolle annetaan kolme erillistä rekisteriä, joita se sitten käyttää operaation suorittamiseen. Koska operaattorit täytyy mahduttaa neljään tavuun, jokaiselle rekisterille jää vain yksi tavu. Tämä tarkoittaa käytännössä sitä, että bittikoodi tukee korkeintaan 256 erillistä rekisteriä operaatioihin. Rajoite ei kuitenkaan tule esteeksi, koska jokaisella funktiolla on niiden oma rekisteriavaruus. Kaikki mahdolliset yhdistelmät näytetään kuvassaKuva 5.



Kuva 5 Mahdolliset 32-bittiset muodot

Toinen mahdollinen rakenne ja optimointi olisi käyttää yhtä parametria kertomaan osoite, ja sen lisäksi erityistä accumulator-rekisteriä, jota käytetään aina toisena operaattorina ja operaation kohteena. Käytännössä tämä puolittaisi bittikoodin koon, kun jokainen operaatio vaatisi vain 16 bittiä. Vaihtoehtoisesti voi käyttää myös 32 bittiä operaatiolle, missä tapauksessa rekisteriraja siirtyisi 8 388 607 rekisteriin. [33.]

Tietyt käskyt, kuten funktiokutsut ja objektikenttien lataaminen, tarvitsevat enemmän dataa kuin mitä yhteen käskyyn mahtuu. Esimerkiksi funktiokutsun täytyy tietää, mitä funktiota kutsutaan, missä argumentit sijaitsevat muistissa, kuinka monta argumenttia on käytössä ja mihin palautettava arvo tallennetaan. Näitä kaikkia ei voi tallentaa neljään tavuun, joten käsky käyttää yhtä ylimääräistä käskyrakennetta ylimääräisten argumenttien tallentamiseen. Toivottavaa olisi, että tämän ylimääräisen käskyn koodi olisi no-op eli nolla, jotta se ei vaikuttaisi ohjelman toimintaan, vaikka suoritus siirtyisi siihen virhetilassa.

6.1.6 Funktiokutsut

Funktiokutsut voidaan jakaa kahteen tai kolmeen erilliseen ryhmään. Ensimmäinen taso on virtuaalikoneen kääntämät bittikoodifunktiot, ja toinen taso on käyttäjän lisäämät natiivit C++-funktiot. Mahdollinen kolmas taso on virtuaalikoneeseen sisäänrakennetut funktiot, niin kutsutut sisäiset funktiot (intrinsic functions). Jokaiselle tasolle on omat kutsuoperaatiot, jotka optimoidaan käyttämään mahdollisimman vähän funktioiden etsintää. Tarkka toteutus kuvataan kappaleessa 6.8.3.

6.2 Tokenisointi

Tokenisoija (Lexer) on toteutettu erillisenä luokkana. Jäsentäjä-funktio luo uuden Lexer-objektin jokaista skriptitiedostoa kohden. Tästä objektista voi sitten pyytää seuraavan tokenin yksi kerrallaan, kunnes koko data on käsitelty tai tokenisoija kohtaa virheen, josta se ei voi palautua. Jokainen tokeni palauttaa sekä sen tyypin että tokenin merkitsemän merkkijonon. Merkkijonoa ei kuitenkaan palauteta kopiona, vaan std::string_view -muodossa pelkällä lukuoikeudella. Tällä vältetään ylimääräiset kopiot ja muistin sirpaloituminen. Käytännössä std::string_view toimii const char* -tapaisesti, mutta sen avulla voi käyttää lähes kaikkia std::string-metodeja [34]. Mahdollinen tokenisarjatuotanto näytetään kuvassaKuva 6.

Sisään:
var x = 10 + 2 * 5;
Ulos:
VAR ID ASSIGN NUM PLUS NUM MULT NUM SEMICOLON

Kuva 6 Lekserin tuottama tulos

Isalnum ja muut sarjan funktiot ovat hitaampia kuin vastaavat ascii-tarkistukset. Tämä johtuu siitä, että monella systeemillä isalnum sisältää lokaalin tarkistuksen, joka on ohjelmointikielen kannalta tarpeeton [35]. Kirjoittamalla funktiot puhtaasti ascii-tarkistusta ajatellen on mahdollista saada jopa yli 20 %:n parannus lukunopeuteen. Keskimääräinen aika tuhannen merkin tarkistukseen on noin 25.4µs vanhalla systeemillä, mutta uudella vain 18.6µs. Käyttämällä std::string string_viewn sijaan nostaa ajan 33.5µs:iin. EMI-kääntäjän tokenisoija käyttää näiden tuloksien takia optimoituja versioita ascii-merkkien tunnistamiseen.

Käytännössä Lexer-luokka on toteutettu siten, että ensimmäinen merkki luetaan nykyisestä kohdasta alkuperäiskoodissa ja sen tyyppi tarkistetaan. Jos löydetty merkki on kelvollinen olemaan numero tai symboli, koodin lukemista jatketaan, kunnes seuraava merkki ei täytä sen hetkistä sääntöä. Jos löydetty merkkisarja on symboli, sitä verrataan hajautustaulussa oleviin avainsanoihin, joista jokaisella on oma palautettava tokeninsa. Jos merkki on sen sijaan jokin muu, sille annetaan merkin vastaama tokeni. Parserifunktio käyttää Lexer-objektia lukemaan merkkejä ja järjestää niistä oikean syntaksipuun.

Kuva 7 esittää tärkeän osan tokenisoijan kehittämistä. Kehityksen aikana tokenisoijalle syötettiin satunnaisia tekstiyhdistelmiä ja sen tuotantoa seurattiin tarkasti. Tokenisoijan tulisi pystyä erottelemaan kaikki oikeat tokenit toisistaan, vaikka ne olisivat harvinaisessa yhdistelmässä tai muuten vaikeasti havaittavissa. Datan ei tarvitse noudattaa kielen kielioppia, vaan riittää että se sisältää sekä oikeita tokeita että virheellistä tekstiä.

```
extend Default:

public def Run()
{
    var 54564TestVariable = 10;
    += - *= ?
    if(true == TestVariable)
    |> || |
    :test
    0x42f x34a
    return TestVariable + 100;
}
```

Kuva 7 Esimerkki tokenisoijan testauksessa käytetystä tekstistä

6.3 Jäsentäjä

Jäsentäjä eli parseri on vastuussa tokenisoijan tuottaman merkkivirran järjestämisestä käsiteltäväksi AST-puuksi. Tämänhetkinen parseri on toteutettu LALR(1)-teknologialla. LALR valittiin parserigeneraattoriksi, koska se tukee vasenta rekursiota, yhteisiä etuliitteitä säännöissä sekä operaattorien sidontajärjestystä, joista yksikään ei olisi mahdollinen LL(0)-parserilla. LALR on myös parempi vaihtoehto kuin LR, sillä sen tuottama LR-taulu vie vähemmän muistia. Seurauksena pienemmästä muistijäljestä kieli menettää vähän mahdollisuuksia, mutta menetetty ekspressiivisyys ei vaikuta nykyiseen kielioppiin lisäsääntöjen avulla.

6.3.1 Rakenne

Kun käyttäjä käynnistää virtuaalikoneen, parseri alustetaan automaattisesti. Debug-alustuksen aikana generoidaan myös jäsennystaulukot, joiden avulla lopullinen jäsennys suoritetaan. Näiden taulukoiden generointi kestää kääntämiseen verrattuna hyvin kauan, joten se tehdään vain kerran ohjelman alkaessa ja lopputulos jaetaan jokaisen säikeen kanssa. LR-taulu voi myös kasvaa varsin suureksi. Kehitysvaiheessa tämä kääntäminen on nopeampaa ja se sallii kieliopin uudelleenlataamisen sulkematta virtuaalikonetta, mutta todelliseen käyttöön se on liian hidas. Sen sijaan julkaisussa käännöstaulu on sisällytetty virtuaalikoneen binaariin ja generointi jätetään kokonaan pois.

6.3.2 LR-taulun rakennus

Debug-versiossa ennen LR-generointia ohjelma lukee kielen kieliopin tekstitiedostosta nopeampaa kehitystä varten. Kuva 8 näyttää osan EMI-kieleen käytetystä kieliopista. Tästä kieliopista generoidaan sääntölista käyttäen perinteistä tuotantosääntö-merkintätapaa, jota käytetään useimmissa parserigeneraattoreissa. Tämän lisäksi tiedostosta luetaan ylimääräistä AST-generaatiodataa, jolla nopeutetaan parserin toimintaa. Generaatiodataa käytetään myöhemmin lukemaan LALR(1)-mallin antamia epäterminaaleja ja muodostamaan niistä mahdollisimman yksinkertainen AST-rakenne. Tätä avataan enemmän yksinkertaistuksen kanssa. Tässä työssä käytettävä LALR-generaattorialgoritmi perustuu avoimen lähdekoodiin JavaScript-toteutuksessa JSMachines-projektissa [36]. Algoritmi muokattiin C++-muotoon ja optimoitiin tehokkaaseen suoritukseen.

```
#rules

Start Program
Program RProgram Program
Program None

RProgram ObjectDef
RProgram FunctionDef
RProgram PublicFunctionDef
RProgram NamespaceDef
RProgram VarDeclare Semi
RProgram Stmt

Typename TypeString
Typename TypeNumber
Typename Id

ObjectDef Object Id Lcurly MObjectVar Rcurly
MObjectVar ObjectVar MObjectVar
MObjectVar None
ObjectVar Id OTypename OExpr Semi
OTypename Colon Typename
OTypename None
OExpr Assign Expr
OExpr None
```

Kuva 8 Kielioppi suunnittelumuodossa, käyttäen tokenien nimiä

LR-taulun rakentaminen koostuu laajasti katsottuna kolmesta eri osa-alueesta. Ensimmäisenä täytyy käsitellä kieliopin antamat säännöt, joista jokaiselle listataan ensimmäiset mahdolliset terminaalitokenit, sekä terminaalit, jotka voivat seurata säännön epäterminaalia. Jokainen sääntö alkaa aina epäterminaalilla, jota seuraa lista mahdollisista tokeneista, esimerkiksi Expression -> Expression Add Expression. Kun näitä seuraavia epäterminaalitokeneita avataan enemmän, lopulta päädytään pelkkiin terminaaleihin. Avauksessa on lähes aina useampi vaihtoehto, esimerkiksi Expression-tokeni voi olla ID, numero tai moni muu terminaali. Kaikista näistä mahdollisuuksista kasaantuu säännön ennakointisetti.

Seuraava vaihe on sääntöjen sulkeumien ja tokenisettien löytäminen. Sulkeumalla tarkoitetaan parseritilan odottaman tokenilistan osaa, jota kyseisessä tilassa ei ole vielä käsitelty, ja tokenisetillä tilan käsiteltyä tokenisarjaa. Jokaisen säännön tokenisarjaa käydään läpi tokeni kerrallaan, ja toisiaan vastaavat tokenisetit ja niiden tilat yhdistetään ja eroavien sulkeumien mukaan setit eritellään omiksi tiloikseen. Mikäli sulkeumat haarautuvat, niiden välille luodaan goto-polku.

Koska tiloja yhdistetään paljon, lopullisia tiloja on huomattavasti vähemmän kuin LR-generoinnissa.

Sulkeumia etsitään generoinnin aikana vertaamalla tilojen mahdollisia ennakointisettejä, jotka koostuvat tokenilistoista [19]. Koska näitä ennakointisettejä voi joutua luomaan ja poistamaan useita satojatuhansia kertoja, taulukoiden allokointi niille käy kalliiksi suorituskyvyn kannalta. Tämän takia tärkeä optimointistrategia on käyttää objektipoolia niille. Suurimman mahdollisen määrän yhtäaikaisia ennakointisettejä voi useiden testien perusteella arvioida nostamalla sääntöjen määrä potenssiin 1.75. Kieliopista riippuen se ei aina päde, mutta ennustus on riittävän tarkka ollakseen hyödyllinen. Objektipoolin käyttäminen välttää lähes kaiken muistiallokoinnin, nostaen suorituskykyä huomattavasti, käyttäen monesti jopa 90 % vähemmän aikaa generointiin.

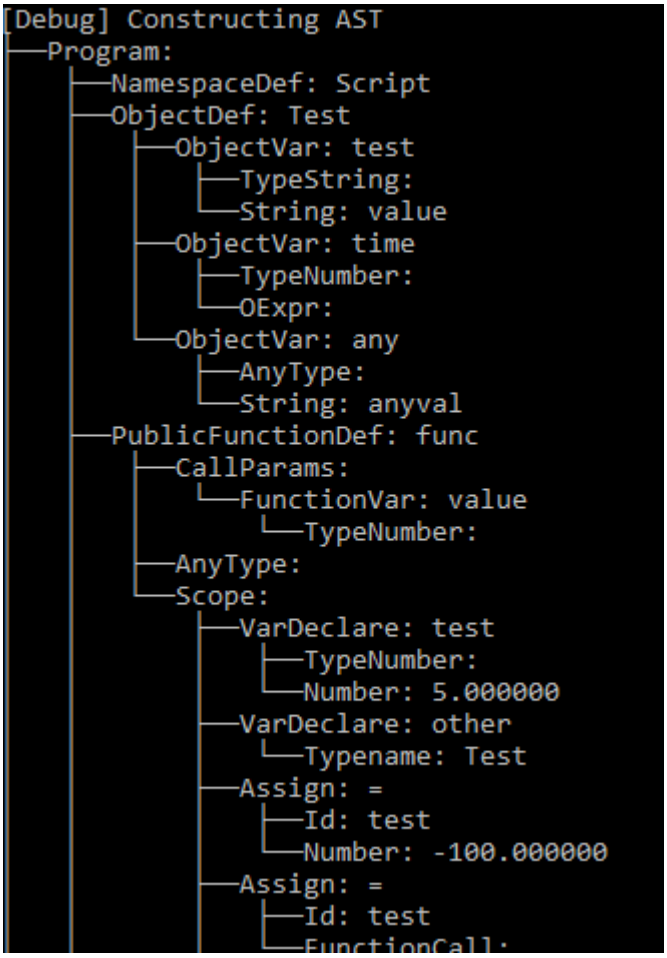
Viimeiseksi valmiiden tokenisettien mukaan generoidaan lopullinen LR-taulu. Jokainen setti on oma tilansa, ja niiden goto-setit määräävät shift-toiminnon ja ennakointisetit määräävät shift-, reduce- tai error-toiminnon. Koska settejä yhdistettiin niiden generoinnin aikana, lopullinen taulu on huomattavasti pienempi kuin LR(x)-generoinnissa, jossa sitä ei tehty. Tästä johtuen on kuitenkin mahdollista, että useampi tila haluaa muokata samaa toimintoalkiota taulukossa, mistä seuraa erilaisia virheitä. Näitä kutsutaan shift/reduce- ja reduce/reduce-virheiksi sen mukaan, mikä toiminto taulukossa oli virheen käydessä. Tavallisesti reduce/reduce -tyyppisestä virheestä ei voi palautua generoinnin aikana, mutta siihen on kehitetty sääntöjä ja kiertoteitä, joista osaa käytetään tässä projektissa.

Tavallisesti shift/reduce- ja reduce/reduce-virheet ovat ongelmallisia ja kertovat heikosti määritellystä ja epäselvästä kieliopista, mutta EMI-kääntäjässä niitä käytetään määrittämään operaattorien sidontajärjestys. Yleinen LR-taulun solu koostuu vain shift- tai reduce-toiminnosta ja numeerisesta indeksistä, mutta nyt tauluun lisätään myös mahdollinen decide-toiminto ja ylimääräinen indeksi. Toista indeksiä käytetään shift-toiminnan kanssa ja toista reducen kanssa. Jos toiminto on kuitenkin decide, jäsentäjä voi tilanteen mukaan päättää, kumpaa toimintoa se haluaa käyttää. Jäsentäjä pitää ylimääräistä listaa kaikista kontekstin aktiivisista operaattoreista, ja kun se kohtaa decide-toiminnon, se vertaa edellisen ja nykyisen operaattorin sidontaprioriteettiä. Jos uuden operaattorien sidontaprioriteetti on pienempi kuin nykyisen, käytetään reduce-sääntöä, ja jos se on suurempi, käytetään shift-sääntöä. Jos molemmilla operaattoreilla on sen sijaan sama prioriteetti, verrataan niiden assosioituvuutta. Perinteiset aritmetiikkaoperaattorit ovat vasen-assosioituvia ja niiden kanssa käytetään reduce-sääntöä. Oikea-assosioituvien kanssa käytetään shift-sääntöä. Jos taas verrattavaa operaattoria ei löydy, käytetään aina shift-sääntöä. [37]

Tämän avulla päästään eroon shift/reduce-ongelmista. Reduce/reduce-konfliktit ovat silti ongelmallisia, mutta tämän kieliopin kannalta paras ratkaisu on ottaa sidontaprioriteetti sääntöjen järjestyksestä. Toisin sanoen, konfliktitilanteessa valitaan reduce-toiminto, jonka sääntö on määritelty kieliopissa ensimmäisenä. Tämä mahdollistaa operaattoreiden ja muiden epäterminaalien sidontajärjestyksen määrittämisen myös sääntöjä kirjoitettaessa.

6.3.3 Jäsennys

Itse parseri käyttää valmista LR-taulua luodakseen AST-rakenteen. Se lukee uuden tokenin lekserin antamasta virrasta ja etsii sen LR-taulusta. Taulun solu kertoo nykyisen tilan ja tokenin perusteella tarvittavan toiminnon ja parseri suorittaa sen. Shift-sääntö lisää tokenin ja sen arvon symbolipinoon ja siirtyy LR-taulun osoittamaan tilaan. Jos toiminto taas on reduce, parseri poistaa symbolipinosta säännön kertoman määrän symboleita ja siirtyy LR-taulun mukaisesti uuteen tilaan. Samalla luodaan uusi solmu ja sille lisätään lapseksi jokaisen poistetun symbolin sisältämä solmu. Koska LR-parserit ovat bottom-up-tyyppisiä, valmiiseen solmuun ei enää tämän jälkeen kosketa. Solmun voi siis antaa yksinkertaistajalle käsiteltäväksi ilman ongelmia. Valmis AST sisältää hyvin paljon solmuja, vaikka monet niistä ovat täysin turhia lopullisen muodon kannalta. On kuitenkin mahdollista optimoida solmujen luontia ja välttyä ylimääräisiltä solmuilta tuottaen yksinkertaisen AST-rakenteen, kuten kuvassaKuva 9. AST ilman optimointeja on liian suuri esitettäväksi tekstimuodossa.



Kuva 9 Generoitu ja optimoitu AST tekstimuodossa

6.4 Yksinkertaistus

Monet ominaisuudet kielessä ovat vain syntaksitasoisia, ja käyttäytyvät samoin kuin jokin toinen ominaisuus. Tästä yksinkertaisin esimerkki on for- ja while-silmukat. For-silmukka on vain käyttäjää varten tehty yksinkertaistus, jotta kielen käyttö olisi sujuvampaa, mutta kuitenkin käännettäessä se yksinkertaistetaan while-silmukaksi. Sama toistuu monen ominaisuuden kanssa, esimerkiksi useassa operaattorissa. Myös vakioarvot ja ylimääräisen koodin voi etsiä ja poistaa mahdollisuuksien mukaan. EMI-kielessä silmukoita ei kuitenkaan yksinkertaisteta vielä tässä vaiheessa, vaan ne käsitellään erikseen bittikoodin generoinnin aikana optimointeja varten.

Tärkeä yksinkertaistuksen vaihe suoritetaan jo parserin aikana. Aikaisemmin ladatut generaatiosäännöt kuvaavat mahdollisuuksia optimoida ja yksinkertaistaa AST-solmuja jo niiden luonnin ai-

kana. Koska LALR on bottom-up parseri, sen luomat AST:t voivat kasvaa hyvin suuriksi ja sen mukana myös tarvittavien allokointien määrä kasvaa liikaa. Käyttämällä näitä lisäsääntöjä, käyttäjä voi määrittää tiettyjen sääntöjen sulauttamisesta. Esimerkiksi voi määrittää, että sääntö Stmt -> Expr ';' ottaa valmiiksi luodun solmun tyyppiä Expr ja käyttää sitä suoraan luomatta Stmt-tyyppistä solmua. Tällä voi toisin sanoen välttää allokoinnin lähes kaikille epäterminaalien solmuille ja valmis AST on huomattavasti nopealukuisempi. Lopullisessa AST:ssä solmuja on vain kaikilla terminaaleilla ja muutamalla epäterminaalilla.

Solmuoptimointi tapahtuu välittömästi solmun luonnin jälkeen. Optimointialgoritmi toimii eri tavalla jokaiselle solmutyypille, mutta esimerkiksi operaattoreille se tarkistaa, mikäli kaikki operandit ovat vakioita, ja jos ne ovat, se evaluoi solmun, asettaa tuloksen solmun arvoon ja muuttaa solmun tyypin vastaavaksi vakioksi. Toinen esimerkki on Scope-solmut, joista optimoija etsii mahdollisia Control-tyyppisiä lapsia, eli continue-, break- ja return-avainsanoja. Koska Control-solmut lopettavat aina Scope-evaluoinnin, niiden jälkeiset lapset ovat turhia ja ne voi poistaa ilman vaikutusta ohjelman toimintaan.

EMI-kielessä ainut suora AST-muokkaus jäsennyksen jälkeen tehdään putkioperaattorille. Putki ottaa kaksi parametria ja asettaa ensimmäisen parametrin seuraavan argumentiksi. Tuloksena operaattori yksinkertaistetaan funktiokutsusarjaksi ennen bittikoodin generointia.

6.5 Abstrakti syntaksipuu

Parserin jälkeen tuloksena on valmis abstrakti syntaksipuu. Sen jokainen solmu sisältää solmun tyypin, mahdollisen arvon std::variant-muodossa sekä mahdolliset osoittimet lapsisolmuihin. AST:n voisi suorittaa jo tässä kohtaa, mutta tulevien optimointien ja suorituksien nopeuttamiseksi se muutetaan vielä IR-bittikoodimuotoon (Intermediate Representation, väliaikainen ilmaisu).

Solmut sisältävät myös dataa IR-koodin generointia varten. Tähän kuuluvat rekisteri-indeksi, osoitin symboliin ja muuttujatyyppi. Nämä täytetään vasta generoinnin aikana ja ovat merkityksellisiä vain optimoinnin ajan.

6.6 Nimiavaruudet ja näkyvyysalueet

Nimiavaruudet on toteutettu siten, että jokainen avaruus säilytetään hajautustaulukossa nimensä alla, ja jokainen niistä sisältää tiedot omista funktioista, muuttujista ja objekteista.

Paikalliset näkyvyysalueet on toteutettu puurakenteella siten, että jokaisella alueella on yksi omistaja ja useampi lapsi. Bittikoodin generoinnin aikana on aina tiedossa, mikä on nykyinen alue, ja sen mukaan voi etsiä ja lisätä symboleita. Kun symboleita lisätään, ne lisätään aina nykyiseen näkyvyysalueeseen, mutta etsittäessä käydään läpi ensin nykyinen alue ja sen jälkeen rekursiivisesti nykyisen alueen omistaja, kunnes symboli löytyy tai näkyvyysalueella ei ole enää omistajaa.

Uusia nimiavaruuksia perustetaan ainoastaan, kun kääntäjä löytää extend-merkin syntaksipuusta. Kaikki sen jälkeiset funktiot, objektit ja muuttujat yksinkertaisesti lisätään sen luoman avaruuden alle. Sen sijaan näkyvyysalueita luodaan monessa erillisessä paikassa, esimerkiksi aaltosulkeilla tai koodilohkoissa. If-lohko tekee itsensä ympärille aina uuden näkyvyysalueen, jotta sen ehdossa voi alustaa paikallisen muuttujan.

6.7 Bittikoodin generointi

Bittikoodi generoidaan kävelemällä syntaksipuun solmuja läpi yksitellen. Kävelijä on suuri switch-valikko, jossa jokaisella solmutyypillä on oma generointimenetelmä. Vaihtoehtoinen toteutus olisi ollut käyttää virtuaalisia funktioita ja perittyjä lapsiluokkia, mutta niistä aiheutuva ylimääräinen kustannus ei ole hyväksi kääntäjän suoritusnopeudelle. Kävelijä on jaettu lataus- ja tallennushaaroihin. Lähes aina käytössä on lataushaara, mutta tallennusversio tarvitaan, sillä osalla AST-solmuista on erilainen tulos riippuen, kummalla puolella operaatiota ne ovat. Esimerkiksi muuttujaan tallennus on erilainen kuin muuttujasta lataus ja se tarvitaan vain, jos muuttujan nimi on asetusoperaattorin vasemmalla puolella. Asetusoperaattori käyttää sen takia tallennushaaraa kävelijästä vasemman lapsisolmun käsittelyssä.

Generoinnissa jokainen solmu kävelee omat lapsisolmunsa ja kirjoittaa oman komentonsa käskytaulukkoon. Esimerkiksi Add-solmu ensin kävelee lapsensa, ottaa niiden kohderekisterit muistiin, vapauttaa lapsista kaikki mahdolliset rekisterit ja lopuksi kirjoittaa Add-käskyn uuden varatun rekisterin kanssa ja lisää parametreiksi lapsien rekisterit.

Generoinnin tuloksena on keskimuotoinen IR-koodi, josta esimerkin antaa Kuva 10. Yleensä tämä koodin muoto muokataan mahdollisimman nopeaksi ennen suoritusta, mutta kääntäjien optimointistrategiat ovat niin laajoja, ettei niitä oteta huomioon tässä työssä.

```
01 01 00 00
09 02 07 00
01 01 01 00
0a 03 01 00
01 04 02 00
06 03 00 03
01 04 03 00
06 01 00 03
08 03 01 00
0f 03 05 00
0f 01 01 00
0c 00 02 00
01 04 04 00
10 04 01 00
0c 00 02 00
01 04 02 00
10 04 01 00
01 03 02 00
12 03 01 03
10 03 01 00
```

Kuva 10 Funktio bittikoodina kolmiosoitemuodossa. Jokainen rivi on yksi instruktio

6.7.1 Tyypitys

Koodin generoinnin aikana kääntäjä tekee arvojen tyypitystä mahdollisuuksien mukaan. Tietyt käskyt, kuten aritmetiikka, voidaan optimoida tehokkaammin, jos kaikki datatyypit ovat tiedossa, joten kääntäjä pitää kirjaa kaikkien rekisterien ja arvojen tyypeistä. Tyypitystä vaikeuttaa kielen vapaa muoto, jossa itse tyypin kirjaaminen on vapaaehtoista. Koska kieli sallii myös tuntemattomia muuttujia, joita ei ole käännetty vielä, kaikkea vaadittua tyypitystä ei voi tehdä ja siksi suorituksen aikana tehdään myös tyyppitarkistuksia kriittisissä kohdissa.

Kieli sisältää erityisen datatyypin nimeltään Undefined (määrittämätön), jota käytetään merkitsemään tuntemattomia arvoja tai viallisia laskuja. Jos tähän arvoon törmätään tyypityksen aikana, tyypitys usein päätyy sanomaan, että kaikki sitä seuraavat arvot tulevat olemaan myös määrittämättömiä. Kielen tyypitys antaa enemmän merkitystä operaatioiden vasemmalle puoliskolle. Jos

sen tyyppi on tiedossa, operaation tulos tulee olemaan samaa tyyppiä. Oikean puolen tyypillä on hyvin harvoin merkitystä, vaan se muutetaan aina vasemman puolen tyyppiin. Tästä seuraa, että määrittämätön tyyppi vasemmalla puolella muuttaa koko operaatioketjun määrittämättömäksi, mutta oikealla puolella sille lisätään muunnos oikeaan tyyppiin.

Funktiokutsuissa verrataan argumenttien tyyppejä funktion pyytämiin tyyppeihin. Toiminnallisuus on kuitenkin vajaata, ja kaikkia funktiotyyppejä ei aina verrata. Myöskään funktioiden ylikuormitus ei ole mahdollista vielä, vaan se vaatii paremman tyypityksen ollakseen hyödyllinen ja luotettava.

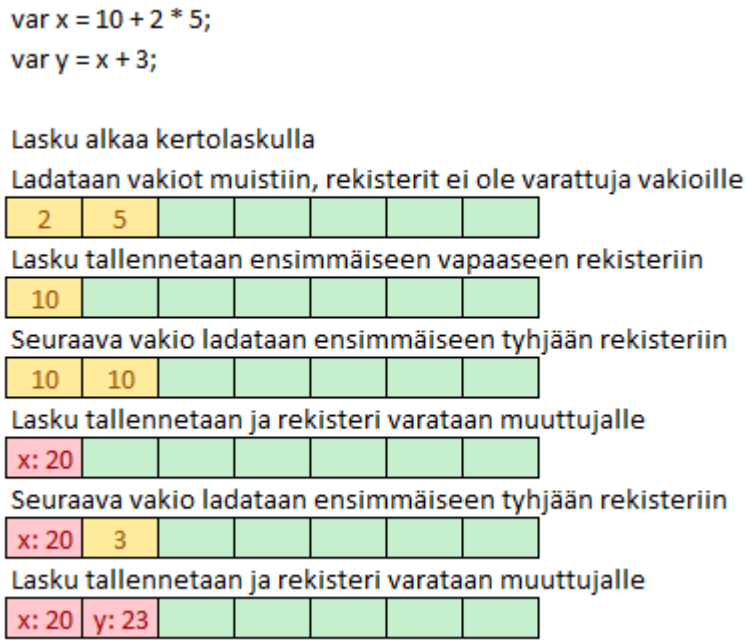
6.7.2 Paikalliset muuttujat

Rekisteripohjaisessa virtuaalikoneessa paikalliset muuttujat ovat hyvin yksinkertaisia toteuttaa. Jokaiselle muuttujalle on varattu symboli paikallisen näkyvyysalueen symbolitaulukosta ja generoinnin aikana tälle symbolille varataan ensimmäinen vapaa rekisteri. Aina kun muuttujaan viitataan myöhemmin, operaatiossa käytetään vain muuttujan omaa rekisteriä uuden sijaan. Tällä minimoidaan ylimääräiset kopioinnit rekisterien välillä.

Tämän lähestymistavan heikkous on bittikoodissa käytetty kolmiosoitejärjestelmä, joka rajoittaa käytön 256 rekisteriin. Käytännössä yhtä aikaa aktiivisia paikallisia muuttujia voi olla vain alle rekisterien määrä, mutta tässä auttaa niin kutsuttu “liveness analysis”, eli elinajan mittaus. Jos kääntäjä huomaa, että muuttujaa ei käytetä tai jos kaksi muuttujaa osoittaa samaan rekisteriin, niiden rekisterit voidaan vapauttaa viimeisen käyttökerran jälkeen ilman vaikutusta ohjelman toimintaan. Tätä ei ole kuitenkaan lisätty EMI-kääntäjään.

6.7.3 Rekisterien varaaminen

Jokainen käsky vastaa itse rekisterien varaamisesta ja vapauttamisesta. Kun AST:tä käydään läpi, jokainen solmu etsii vapaan rekisterin ja vapauttaa lapsisolmut. Se voi myös vaihtaa lapsisolmun kohderekisterin tarvittaessa. Koska rekistereitä varataan ja vapautetaan jo generointivaiheessa, rekisterien määrän voi minimoida ilman erillisiä optimointeja. Kuva 11 esitetään rekisterien varausprosessi pienelle koodipalaselle.



Kuva 11 Esimerkki rekisterien varaamisesta

Kuten kuvasta näkyy, rekistereitä ei varata pysyvästi vakioille. Ainoastaan muuttujien ja muiden symbolillisten solmujen rekisterit pidetään varattuina pysyvästi. Vakioiden ja vakio-operaatioiden rekisterit vapautetaan aina solmun käsittelyn jälkeen, jotta rekisterien määrä saataisiin minimoitua. Todellisuudessa tätä pitää optimoida lisää, sillä kaikkia vakiokutsuja ja globaalien muuttujien latauksia ei tarvitsisi tehdä aina uudestaan, vaan niille voisi varata pysyvän rekisterin.

6.8 Virtuaaliympäristö

Virtuaalikone toteutetaan yhdessä luokassa, joka sisältää kaiken suoritukseen tarvittavan datan. Kaikilla suorittajasäikeillä on pääsy virtuaalikoneen dataan, mutta tätä vapautta voidaan rajoittaa tulevaisuudessa. Monet suorituksen kannalta tärkeät asiat ovat kuitenkin globaaleja eivätkä virtuaalikonekohtaisia ja näistä kaikista yhdessä koostuu niin kutsuttu virtuaaliympäristö. Globaaliin dataan sisältyy muun muassa sisäänrakennettujen funktioiden listat ja käyttäjien sitomat funktiot.

6.8.1 Suoritus

Virtuaalikoneen suoritus tapahtuu yhden silmukan sisällä. Kun virtuaalikoneen kutsupinoon lisätään kutsu, se aloittaa suorituksen uudella säikeellä. Suorituksen aikana silmukka ajaa bittikoodia läpi yksi käsky (opcode) kerrallaan. Itse bittikoodi on binäärimuodossa ja se haetaan uint32-tyyppisen osoittimen kautta. Jokainen käsky sisältää opcoden vaatimat rekisterit, joita virtuaalikone sitten käyttää operaatioissa. Suorituksen alussa lasketaan sekä suoritus- että lopetusosoittimen sijainti bittikoodin alku- ja loppukohdan mukaan.

Itse suoritussilmukka on teoriassa vain suuri switch-rakenne, joka sisältää jokaisen käskyn peräkkäisenä listana. Kaikki ylimääräinen laskutoimitus on jätetty pois, joten käskyn ensimmäinen tavu määrää hypyn osoitteen ja käskyn käsittelyn jälkeen osoitinta siirretään eteenpäin 32-bitin verran ja seuraava käsky suoritetaan. Tätä jatketaan, kunnes suoritusosoitin saavuttaa lopetusosoittimen. Käytännössä switch-rakenne ei ole optimaalinen tähän, vaan hypyt tehdään goto-lauseella ja hyppytaulukolla.

6.8.2 Funktioiden sitominen

Natiivien C++- ja C-funktioiden sitominen on haastavaa. EMI-ympäristön toteutus ei tue vielä puhdasta C-käyttöliittymää, mutta C++-funktioista kaikki ovat tuettuja. Käyttöliittymä sisältää makron funktioiden rekisteröintiin, ja käyttäjän ei tarvitse kirjoittaa muuta kuin haluttu nimiavaruus, funktion nimi ja funktion osoite. Parametrit ja niiden tyypin määritetään C++-template-strategialla käyttäen konsepteja ja yksinkertaista funktiosäiliörakennetta. Yksinkertaisinta olisi käyttää std::function-säiliötä, mutta sitä ei ole turvallista lähettää DLL-rajapinnan yli muistiasettelun erojen vuoksi. Tämän takia rajapinta sisältää oman luokan, joka voi sisältää kaikki std::functionin tukemat funktiot. Rajapinnan avustusfunktiot (Kuva 12) luovat tämän luokan automaattisesti, minkä jälkeen se annetaan virtuaalikoneen käsiteltäväksi ja sidottavaksi. Virtuaalikone pitää huolen funktioiden vapauttamisesta ja kutsumisesta.

```
template<class V, class F, typename ...Args, size_t... S> requires std::is_void_v<V>
constexpr auto __make_caller(std::index_sequence<S...>) {
    return +[](void* ptr, size_t, InternalValue* args)->InternalValue {
        (*(F*)(ptr))((args[S].as<Args>())...); return {}; };
}

template<class F, class...Args> requires (std::is_convertible_v<F, InternalValue> ||
    std::is_void_v<F>) && ((std::is_convertible_v<Args, InternalValue>) && ...)
bool RegisterFunction(const std::string& space, const std::string& name, std::function<F(Args...)>&& f) {
    auto retval = new __internal_function();
    constexpr size_t size = sizeof...(Args);
    constexpr auto seq = std::make_index_sequence<size>();
    char* c = new char[name.length() + 1];
    strcpy_s(c, name.length() + 1, name.c_str());
    retval->name = c;
    retval->space = nullptr;
    if (space.length() > 0 && space != "Global") {
        c = new char[space.length() + 1];
        strcpy_s(c, space.length() + 1, space.c_str());
        retval->space = c;
    }
    retval->arg_types = new ValueType[size]{type<Args>()...};
    retval->return_type = type<F>();
    retval->arg_count = size;
    retval->state = new std::decay_t<std::function<F(Args...)>>(std::forward<std::function<F(Args...)>>(f));
    retval->operate = __make_caller<F, std::decay_t<std::function<F(Args...)>>, Args...>(std::make_index_sequence<size>());
    retval->cleanup = +[](void* ptr) {delete static_cast<std::decay_t<std::function<F(Args...)>>*>(ptr); };
    return __internal_register(retval);
```

Kuva 12 Yksi automaattisen funktiorekisterin käyttämistä apufunktioista

Käyttäjä voi myös poistaa sidottuja funktioita. Virtuaalikone säilyttää hajautuslistan kaikista käytössä olevista funktio-osoittimista, ja kun käyttäjä poistaa funktion, se poistetaan myös aktiivisten funktioiden listalta. Funktioita kutsuttaessa virtuaalikone tarkistaa aktiivisten funktioiden listan, ja jos osoitetta ei löydy funktio asetetaan poistetuksi.

6.8.3 Funktiokutsut

Funktioiden kutsuminen on jaettu kolmeen osaan: bittikoodin, sisäänrakennettujen ja C++-funktioiden suoritukseen. Jako on tärkeää optimoinnin kannalta, koska bittikoodin kutsuminen ei vaikuta ohjelman omaan kutsupinoon, vaan suoritus pysyy saman silmukan sisällä. Sidottuja C++-funktioita kutsuttaessa sen sijaan pitää siirtää muuttujat muistiin, käsitellä kutsupinoa ja siirtyä pois virtuaalikoneen sisältä. Sisäänrakennetut funktiot ovat kaikkein nopeimpia kutsua, sillä ne saavat suorat osoittimet kohderekistereihin.

Bittikoodia suorittaessa funktion kutsuminen on yksinkertaista. Funktiokutsun operaattori sisältää tavun mittaisen parametrin, joka kertoo indeksin paikalliseen funktio-osoitetaulukkoon. Jos taulukon indeksi ei sisällä oikeaa osoitetta, se haetaan nimitaulukon kautta dynaamisesti. Seuraavaksi tarkistetaan funktion parametrit, jolloin kaikkien tyypitettyjen argumenttien täytyy vastata funktion odottamia tyyppejä tai funktiota ei kutsuta. Jos kaikki argumentit ovat sopivia, uusi funktio lisätään suorittajan kutsupinoon, suorituksen osoitin vaihdetaan uuden funktion bittikoodin

alkuun ja rekisteripinoa kasvatetaan riittävän paljon. Myös paikalliset vakio- ja symbolitaulukot vaihdetaan uuden funktion taulukoihin.

Rekisteripinoa kasvatetaan niin paljon, että kaikki funktion tarvitsemat rekisterit ovat saatavilla. Näihin rekistereihin sisältyy myös mahdolliset argumentit, joita funktio odottaa. Kutsuva käsky määrittää yhden tavun kokoisella argumentilla funktiokutsun argumenttien ensimmäisen rekisterin indeksin. Ensimmäinen rekisteri-indeksi on tärkeä optimointimenetelmä, joka liittyy funktion paikallisten rekisterien allokointiin.

Funktiokutsun argumentit käännetään bittikoodiin erityisellä menetelmällä. Ensimmäinen argumentti valitsee ensimmäisen vapaan rekisterin, jota seuraavat rekisterit ovat myös vapaana, ja asettaa tämän rekisterin käskyn kohteeksi. Seuraavat argumentit suoritetaan peräkkäisiin rekistereihin, ja jos kyseessä on muuttuja, se kopioidaan uuteen rekisteriin. Tästä menetelmästä seuraa se, että kaikki argumentit ovat järjestyksessä rekisteripinossa ensimmäisestä alkaen. Kun seuraavan funktion rekistereitä varataan, se voidaan aloittaa näiden argumenttien ensimmäisestä rekisteristä, missä tapauksessa argumentit ovat aina ensimmäisissä funktion lokaaleissa rekistereissä ilman ylimääräistä kopiointia. Jos funktiota kutsutaan myös pienemmällä määrällä argumentteja kuin odotettu, muut argumentit on aina asetettu määrittämättömiksi automaattisesti rekisteriallokaation ohella.

Sisäänrakennettuja funktioita kutsutaan antamalla niille kohderekisterin osoite, ensimmäisen argumentin osoite ja argumenttien määrä. Koska kaikki rekisterit ovat säilytetty peräkkäin muistissa, funktio voi iteroida kaikkien argumenttien yli käyttämällä ensimmäisen osoitetta ja argumenttien määrää. Ne ovat myös huomattavan vapaita argumenttien määrän ja tyyppien kanssa, mutta sitä voi rajoittaa käyttämällä erillistä funktiotyyppitaulukkoa sekä käännösvaiheessa että ajonaikaisesti.

Käyttäjän lisäämien funktioiden kutsuminen on suhteellisesti erittäin raskasta. Koska käyttäjällä ei ole pääsyä virtuaalikoneen sisäiseen muistiin, kaikki rekisterit kopioidaan väliaikaiseen tauluun ja muutetaan ulkoisiksi arvoiksi. Tätä uutta taulukkoa käytetään sitten ulkoisen funktion kutsun argumentteina.

6.8.4 Käskysetti

Tärkeä osa virtuaalikoneen toteutusta on käskysetin kasaaminen. Virtuaalikoneen tulee tukea kaikkia operaatioita, joita voi tarvita ohjelman suorittamiseen. Näihin sisältyy esimerkiksi aritmeettiset operaatiot, symbolien lataaminen ja tallentaminen, funktiokutsut ja niin edelleen. EMI tarjoaa monta erikoistettua käskyä optimoituun koodin suorittamiseen, mutta tässä käsitellään vain tärkeimmät. Kaikki ympäristön käskyt listataan liitteessä 1.

Aritmeettiset operaatiot, kuten lisäys ja vähennys, on optimoitu argumenttien datatyyppien mukaan. Esimerkiksi operaatiot numeroille on optimoitu siten, ettei niiden tarvitse erikseen tarkistaa ja muuttaa argumenttirekisterien tyyppejä, jos ne ovat jo ennestään tiedossa kääntäjän ansiosta. Kaikista operaattoreista on kuitenkin saatavilla muoto, joka tarkistaa muuttujien tyypin jokaisella suorituskerralla ja muuttaa käyttäytymistään sen mukaan.

Silmukoita varten kieleen on lisätty useita erilaisia käskyjä, jotka kuitenkin kaikki jakautuvat kahteen ryhmään. Kaikki silmukkatyypit on mahdollista toteuttaa käyttämällä vain hyppy- sekä vertailukäskyjä. Näihin kahteen käskytyyppiin on kuitenkin lisätty useita erilaisia käskyjä, kuten yhtäsuuruus- tai suhdevertausoperaattorit. Hyppyoperaattoreista esimerkkejä on hypyt eteenpäin, taaksepäin ja tiettyyn osoitteeseen. Jokainen näistä ottaa hieman erilaiset parametrit ja niiden toiminta on optimoitu niiden käyttötarkoitukseen. Hyppykäskyt toimivat siten, että ne siirtävät suoritusosoitinta joko eteen- tai taaksepäin käskystä riippuen. Ne voivat myös siirtää suoritusosoittimen satunnaiseen kohtaan suoritusta, missä tapauksessa uusi kohta etsitään käyttämällä indeksiä bittikoodin alusta.

Vastaavasti kuin silmukat, myös if-lauseet on optimoitu käyttämään vain vertailu- ja hyppyoperaattoreita. Vertailuoperaattorilla testataan halutun rekisterin totuusarvo ja sen perusteella hypätään tiettyyn osoitteeseen koodissa. Hypyn sijainti riippuu siitä, onko rakenteessa vain if-lohko vai myös else-lohko.

Kieli sisältää myös käskyt jokaisen eri datatyypin perusarvon luomista varten. Jokainen näistä käskyistä luo halutun datatyypin perusarvon ja asettaa sen kohderekisteriin. Käyttö näille käskyille on lähinnä uusien muuttujien alustuksessa, kun käyttäjä on määritellyt vain muuttujan tyypin.

Poikkeuksena muiden datatyyppien alustuksesta, objektien alustus voi ottaa argumenttina myös rekisteriosoitteen, jota se sitten käyttää objektin kenttien täyttämiseen. Tämä noudattaa samoja

sääntöjä kuin funktioiden argumenttirekisterien käyttö. Toisin sanoen objektikäsky ottaa ensimmäisen rekisterin osoittimen ja käyttää halutun määrän sitä seuraavia rekistereitä täyttämään kaikki objektin kentät. Näin virtuaalikone pystyy käyttämään mahdollisimman vähän aikaa arvojen kopiointiin ja objektin alustukseen ja mahdollisimman paljon aikaa suoritukseen.

Alustuksen lisäksi ympäristössä on myös käskyjä eri arvojen lataamiselle ja niiden asettamiselle rekistereihin. Käskyjä löytyy esimerkiksi numeroiden ja merkkisarjojen lataamiseen kääntämisen aikana kasatuista vakiotaulukoista. Toinen käyttökohde on globaalien muuttujien lataaminen ja tallentaminen virtuaalikoneen muuttujataulukkoon. Sama lataaminen ja tallentaminen koskee myös kaikkien objektien kenttiä sekä taulukkojen indeksejä. Koska tällä hetkellä kaikki nämä käskyt lataavat manuaalisesti arvon tietystä osoitteesta, niiden käyttö ei ole suositeltavaa, sillä se hidastaa ohjelman yleistä suorituskykyä ylimääräisten muistihyppyjen kautta. Tulevaisuudessa varsinkin globaalien muuttujien arvojen lataaminen ja tallentaminen täytyy optimoida.

Viimeinen erilainen käsky on kopiointi, joka kopioi halutun rekisterin toiseen rekisteriin. Tätä käytetään silloin, kun ei haluta luoda referenssiä funktiokutsussa. Koska kutsuttu funktio ottaa tietyn alueen edellisen funktion rekisterialueesta itselleen argumentteja varten, rekisterit käyttäytyisivät kuin referenssit, eli kutsuttu funktio pystyisi muokkaamaan edellisen funktion arvoja. Tämä ei ole haluttua, joten kopiointioperaattoria käytetään poistamaan tämä toiminta siirtämällä arvo pois edellisen funktion rekisterialueelta. Tämä toiminta koskee kuitenkin vain edellisessä funktiossa määriteltyjä muuttujia, ja vakiarvot ja lausekkeiden tulokset vain asetetaan suoraan haluttuun rekisteriin.

6.8.5 Muistinhallinta

Muistinhallinta on toteutettu käyttämällä allokaattoriluokkaa jokaiselle objektityypille. Allokaattorilta voi pyytää objektia, jolloin se luo uuden objektin, alustaa sen ja säilöö osoittimen omaan listaansa. Objektia voi sen jälkeen käyttää huolehtimatta sen tuhoamisesta. Objektiallokaattorit ovat vielä prosessikohtaisia, joten kaikki käyttäjän luomat virtuaalikoneet käyttävät samaa allokaattoria. Optimaalisessa tilanteessa jokainen virtuaalikone käyttäisi omia allokaattoreitaan.

Automaattinen tuhoaminen tehdään referenssilaskennan avulla. Jokainen muuttuja, joka osoittaa objektiin, nostaa kyseisen objektin referenssilaskuria yhdellä, ja muuttujan tuhoaminen laskee sitä. Virtuaalikoneen taustalla pyörii roskankeruusäie, joka etsii objekteja, joiden referenssiluku on nolla tai alle. Kerääjä merkkaa nämä objektit tuhotuiksi ja vapaiksi uudelleenallokointia

varten. Uusien objektien allokoinnin aikana allokaattori voi sitten halutessaan käyttää yhtä näistä tuhotuista objekteista säästäen muistia.

Referenssilaskuri ei ole optimaalinen lähestymistapa virtuaalikoneen roskankeruuseen. Referenssien laskeminen on epävakaata useamman säikeen kanssa ilman muteksien käyttöä, ja niiden kanssa se on liian hidas prosessi. Parempi tapa olisi niin kutsuttu “Merkitse ja pyyhi”-strategia. Merkitse-vaiheen aikana kaikki objektit merkataan poistetuiksi ja pyyhi-vaiheessa kaikki koodista saavutettavat objektit käsitellään ja merkataan saavutettaviksi. Myös jokaisen objektin lapset käsitellään samalla tavalla. Kun pyyhkäisy on ohi, jokainen saavuttamaton eli poistettu objekti voidaan poistaa lopullisesti.

Vaikka tämä onkin parempi strategia, sen implementointi on haastavaa. On myös saatavilla kirjastoja kuten BDWGC (Boehm-Demers-Weiser Garbage Collector [38]), jotka hoitavat automaattisen roskankeruun. Kuitenkin EMI-ympäristön optimointien takia nämä lähestymistavat eivät aina toimi, tai ne vaatisivat suuria muutoksia arkkitehtuuriin. Parempi roskankeruualgoritmi on ehdottoman tärkeä EMI:n toiminnan takaamiseksi, joten sen kehitys on jatkossa prioriteettina.

6.8.6 Reflektointi

Reflektointi ei ole vielä osa EMI-ympäristöä. Sen lisäys olisi erittäin hyödyllistä dynaamista ohjelmointia ajatellen, ja nykyinen arkkitehtuuri tukee sitä lähes ilman muutoksia. Käytännössä reflektiolla tarkoitetaan objektien ja niiden kenttien löytämistä skriptin sisällä käyttämällä merkkijonoja.

Toteutukseen voisi käyttää nykyisiä symbolitaulukkoja, joita käytetään jo valmiiksi dynaamiseen linkitykseen. Sen sijaan, että symbolin haussa käytetään käännöksen aikana kerättyä merkkijonoa, olisi mahdollista sallia skriptissä määritettyjä ja operaatioiden tuloksena luotuja merkkijonoja. Kielen käyttäjä saisi tällä mahdollisuuden valita dynaamisesti objektityyppejä tai funktioita.

6.9 Optimointeja

Virtuaalikoneen toiminnassa jopa pienet asiat voivat olla merkittäviä suoritushyvin kannalta. Yleisiä huomioita suoritussilmukan ohjelmointiin olivat esimerkiksi funktiokutsujen välttäminen, yli-

määräisien haarojen poistaminen ja datan lokaalisuus. Varsinkin datan sijainti muistissa on merkittävä optimointikohde. Mitä tiheämmäksi suoritettavan bittikoodin ja virtuaalikoneen rekisterit ja muistin saa, sitä vähemmän syntyy välimuistihuteja, joissa CPU joutuu lataamaan datan hitaasti RAM:in puolelta. Hudilla tarkoitetaan tilannetta, jossa CPU:n haluamaa dataa ei ole saatavilla välimuistissa tai se on vanhentunutta, jolloin CPU joutuu lukemaan datan RAM:ista.

Funktiokutsut käyttävät C++-kutsupinoa ja hidastavat suoritussilmukkaa hieman. Niiden käytön välttäminen on suositeltavaa aina kun mahdollista. Jos lopullisen silmukan konekielinen koodi ei joudu hyppimään eri funktioiden välillä, itse skriptikielen suoritus tulee olemaan nopeaa. Yleisessä ohjelmoinnissa funktiokutsujen hinta on merkityksetön, mutta suoritussilmukka suoritetaan niin monta kertaa sekunnissa, että jopa pienikin optimointi on merkittävä.

Funktiokutsut, erityisesti virtuaaliset funktiokutsut, ovat erittäin haitallisia suorituskyvylle. Virtuaalista kutsua suorittaessa CPU joutuu tekemään huomattavasti enemmän muistihakuja, ja siten vähentää aikaa itse koodin suorittamiseen. Niitä ei siksi käytetä ollenkaan skriptin suorituksen yhteydessä.

6.9.1 Dynaamisen linkityksen välimuisti

Dynaamisessa linkityksessä ympäristö joutuu etsimään oikean funktion nimen perusteella ajonaikaisesti. Nimen käyttäminen on kuitenkin suhteellisen hidasta, jotain kaikissa paikoissa, joissa suoritin joutuu etsimään tietoja, käytetään erillistä välimuistia tallentamaan löydetyn objektin sijainti muistissa. Jos tämä sijainti on tallennettuna myöhemmissä suorituksissa, suoritin hyppää etsinnän yli ja käyttää vanhaa löydettyä arvoa.

Tästä lähestymistavasta seuraa tiettyjä ongelmia, joita ei ole ratkaistu nykyisessä EMI:n versiossa. Jos suoritin on tallentanut funktion sijainnin välimuistiin ja funktio poistetaan myöhemmin, välimuistia ei vielä tyhjennetä. Seurauksena funktioiden kutsuminen muista käännösyksiköistä ei ole turvallista, jos näitä yksiköitä poistetaan koskaan. Nopea ratkaisu ongelmalle olisi vain tyhjentää kaikkien funktioiden välimuistit käännösyksikön poiston yhteydessä, mutta suuremmilla skriptimäärillä se voi vaikuttaa vakavasti suorituskykyyn. Funktioiden seuraava suoritus joutuu myös tekemään dynaamisen linkityksen alusta lähtien. Lähestymistapa korjaukseen vaatii ohjelman ja erilaisten ratkaisujen profilointia.

6.9.2 Nan-boxing arvoille

Kaikki virtuaalikoneen käyttämät arvot säilytetään 64-bittisen muuttujan sisällä. Suurin yksittäinen käytetty datatyyppi on double, jonka koko on tasan 64-bittiä, joten samat bitit voidaan käyttää esittämään kaikkia ensimmäisen tason datatyyppejä.

IEEE 754 standardin mukaan liukulukujen formaatti käyttää eksponenttibittejä merkkaamaan NaN-lukuja (Not-a-Number, epäluku). Jos kaikki eksponentit on asetettu päälle, luku on NaN, mutta loput bitit määräävät NaN-tyypin. Jos bitti 52, eli korkein mantissan bitti, on asetettu, CPU käsittelee lukua merkitsevänä NaN-lukuna, joka voi vaikuttaa ohjelman toimintaan. Sen sijaan, jos mantissan korkein on nolla, kyseessä on hiljainen NaN, joka ei vaikuta ohjelman käyttäytymiseen. Tämän erittelyn takia hiljaisia NaN-arvoja voidaan käyttää ohjelman valitsemalla tavalla, esimerkiksi säilyttämään eri datatyyppejä. Jos alimmat 48 bittiä varataan datalle, käyttöön jää vielä merkkibitti ja kolme bittiä keskeltä. Näitä bittejä voidaan käyttää esimerkiksi lippuina merkkaamaan eri datatyyppejä ja niiden mukaan käsitellä alimpia databittejä. Jos esimerkiksi bitti 50 on asetettu päälle, luku voidaan käsitellä totuusarvona. Jos taas ylin bitti on asetettu, loput 48 bittiä jättävät tarpeeksi tilaa osoittimelle. Vaikka 64-bittisessä ympäristössä osoittimen koko on täydet 64 bittiä, vain 48 bittiä on oikeasti käytössä yleisimmissä ympäristöissä. Tulevaisuudessa implementaatio ei enää tule toimimaan, mutta nykyään se on vielä luotettava. [39.]

Hyöty tästä menetelmästä tulee erityisesti rekisteripohjaiselle virtuaalikoneelle, sillä nyt jokainen arvo mahtuu suoraan samaan kahdeksaan tavuun ja C++-kääntäjä voi optimoida virtuaalikoneen komentosetin, kun kaikki arvot mahtuvat 64-bittiseen rekisteriin prosessorilla. Se myös tarkoittaa, että virtuaalikone käyttää huomattavasti vähemmän muistia rekistereille ja objektilistoille.

6.9.3 Objektihallinta ja allokointi

Jokaisella objektityypillä, mukaan lukien funktiot ja taulukot, on oma allokaattori, joka pitää osoittimia kaikkiin varattuihin objekteihin. Erityinen optimointi syntyy siitä, että allokaattori tietää aina oman objektityyppinsä koon ja voi antaa vanhan muistiosoitteen käyttämällä niin kutsuttua placement new -operaattoria [40]. Edellinen objekti muistiosoitteessa poistetaan ja uusi varataan ilman, että muistia vapautetaan. Useiden objektien luominen ja vapauttaminen nopeutuu huomattavasti tällä menetelmällä.

6.9.4 Taulukot

Taulukot toteutettiin käyttämällä objektirajapintaa. Taulukko-objekti toimii vain säiliönä std::vector-objektille, joka sisältää virtuaalikoneen arvoja. Koska taulukkoja käsitellään objekteina, kaikki taulukkomuuttujat ovat vain referenssejä ja taulukkoja ei koskaan kopioida ilman erillistä funktiokutsua. Tästä voi aiheutua ongelmia funktioiden kanssa, jos käyttäjä ei odota taulukkoargumentin olevan referenssi ja muokkaa sitä ilman kopiointia funktion sisällä.

EMI-ympäristö tarjoaa jo sisäänrakennettuja funktioita taulukoiden käsittelylle, esimerkiksi koon muuttamiselle, elementtien poistamiselle ja kopioinnille.

6.9.5 Käskyjen suoritus virtuaalikoneessa

Virtuaalikoneen käskysilmukka on teoriassa vain switch-lause, mutta todellisuudessa se on liian hidas ratkaisu tehokkaan suorituksen kannalta. C++-kääntäjä generoi switch-lauseesta hyppytaulukon, mutta se lisää siihen etäisyystarkistuksia ja muuta ylimääräistä koodia, joka hidastaa suoritusta. Vaihtoehtoina tälle on esimerkiksi suora ja epäsuora suoritus (direct and indirect threading). Suorassa suorituksessa bittikoodin käskyt osoittavat haluttuun konekieliseen koodiin, kun taas epäsuorassa käskyt ovat indeksejä taulukkoon, joka sisältää koodin osoitteet. Suora lähestymistapa on usein nopeampi, sillä se välttää yhden hypyn osoitteen haussa, mutta se myös vaatii suurempaa bittikoodia tai erityisiä ympäristön rakenteita [41]. Näistä syistä EMI-ympäristö pyrkii käyttämään epäsuoraa suoritusta, vaikka nykyinen suoritus ei täytä sitä täysin.

Labels-as-values-lisäosa GCC-kääntäjässä mahdollistaa tehokkaan lähestymistavan epäsuoraan suoritukseen. Sen avulla C++-nimikkeitä voi käyttää arvoina taulukossa, ja switch-lauseen sijaan joka käskyn lopussa voi hypätä suoraan taulukon alkion osoittamaan nimikkeeseen. Tämä mahdollistaa paremman haaraennakoinnin nykyisillä CPU-arkkitehtuureilla sekä nopeamman koodin suorituksen. [41.]

Kääntäjillä, jotka eivät tue kyseistä lisäystä, voidaan käyttää lähes vastaavaa rakennetta ja toivoa, että kääntäjä optimoi lopullisen koodin lähes samaksi. Käytännössä switch-lauseella hypätään goto-lauseeseen, joka hyppää suoritettavaan koodiin. Kääntäjä voi tämän avulla optimoida ensimmäistä switch-lausetta hyvin tiheäksi hyppytaulukoksi ja sen kautta avustaa CPU:n haaraennakointia. Tämä kuitenkin riippuu kääntäjän toiminnasta eikä aina vaikuta suorituskykyyn.

Bittikoodin loppuun lisättään erityinen lopetuskäsky, jonka avulla voi jättää pois etäisyystarkistuksen koodin suorituksesta. Kun lopetuskäskyyn törmätään, se lopettaa bittikoodin lukemisen ja nollaa nykyisen säikeen. Yleensä käskyä ei kuitenkaan koskaan käytetä, sillä kaikki funktiot loppuvat aina return-käskyyn.

6.10 Bittikoodin kehitysprosessi

Kun ensimmäiset käskyt ja alustava rakenne ohjelmalle olivat valmiita, uusien käskyjen ja ominaisuuksien lisääminen oli suhteellisen yksinkertaista. Kehitysprosessi alkoi kirjoittamalla parseri- ja AST-generointi, minkä jälkeen projektiin lisättiin bittikoodin generointi. Bittikoodi ja virtuaalikone luotiin pitkälti yhtä aikaa, sillä niiden rakenne ja toiminta riippuvat pitkälti toisistaan.

Kehityksen aikana tapahtui useita muutoksia suunnitelmaan. Alun perin projektissa oli tavoitteena luoda pinopohjainen virtuaalikone, mutta ensimmäisten käskyjen jälkeen rakenne vaihdettiin rekisteripohjaiseksi suorituskyvyn parantamiseksi ja kehityksen nopeuttamiseksi. Muutoksessa ja rekisterien suunnittelussa kesti suhteellisen pitkään, mutta uusien käskyjen lisääminen oli paljon yksinkertaisempaa kuin pinopohjaisessa arkkitehtuurissa.

Yksinkertaisuudessaan käskyjen lisääminen seurasi aina samaa prosessia. Koska kääntäjä pystyi jo kääntämään kaikki kielen käyttämät rakenteet, testikoodiin lisättiin uusia käskyjä käyttävä rakenne. Esimerkiksi jos tavoitteena on lisätä if-rakenne kieleen, testikoodiin lisättiin yksi tai useampi if-lause. Tämän jälkeen bittikoodin generaattoriin lisättiin uuden AST-solmun käsittely, jossa lisätään solmun lapsien käsittely, bittikoodin lisäys, rekisterivarausalgoritmi ja rekisterien vapautus. Käytännössä käskylistaan lisätään usein uusi käsky ja sen argumentit haetaan lapsisolmuista.

Bittikoodin jälkeen käskylle luodaan toiminta virtuaalikoneen sisälle. Kun käsky lisätään käskyjen enumeraattoriin, sille luodaan automaattisesti alkio hyppytaulukkoon. Itse toiminta määritellään muiden käskyjen rinnalle välttäen ylimääräisiä funktiokutsuja. Toiminnan testaus tapahtuu ajamalla testikoodia useamman kerran ja tarkkailemalla tuloksia. Myös aikaisempia käskyjä testataan usein, sillä kehityksen aikana huomattiin, että yhden käskyn rekisteriallokointi rikkoi hyvin herkästi aikaisempien käskyjen rekistereitä.

Projektin lopussa rekisteriallokointi suunniteltiin pitkälti kokonaan uudella tavalla. Alussa jokainen solmutyyppi oli vastuussa rekisterien vapauttamisesta, mutta se vaihdettiin toimimaan siten, että jos lapsisolmulla ei ole symbolia asetettuna, sen kohderekisteri vapautetaan aina. Tuloksena

rekisterigenerointi on luotettavampi ja nopeampi, mutta myös hieman epäoptimaalisempi. Koska myöhemmin tarkoituksena on lisätä erillinen IR-optimointivaihe, nyt luotettavuus on tärkeämpi kehityskohde.

Itse varauksen muutos oli nopea tehdä, sillä bittikoodin generointi on toteutettu käyttämällä suurta määrää C++-makroja. Vaikka niiden käyttö on yleisesti epäsuosittua epäselvyyden ja ongelmallisen käytön takia, tässä yhteydessä ne toimivat erityisen hyvin. Koska kaikki generointivaiheet käyttivät samoja makroja, varauksen muuttamiseksi oli tarpeeksi muuttaa vain makron rakenne. Saman pystyisi tietysti toteuttamaan inline-funktioilla, mutta makrot tarjosivat enemmän vapautta generoinnin suunnitteluun. Kuva 13 on esimerkki makrojen käytöstä bittikoodin generoinnissa.

```
case Token::Not: {
    Walk;
    Op(Not);
    In8 = First()->regTarget;
    FreeChildren;
    Out;
    Type = VariableType::Boolean;
}break;
```

Kuva 13 Not-solmun käskyn generointi makroilla toteutettuna

6.11 Lopullinen tulos

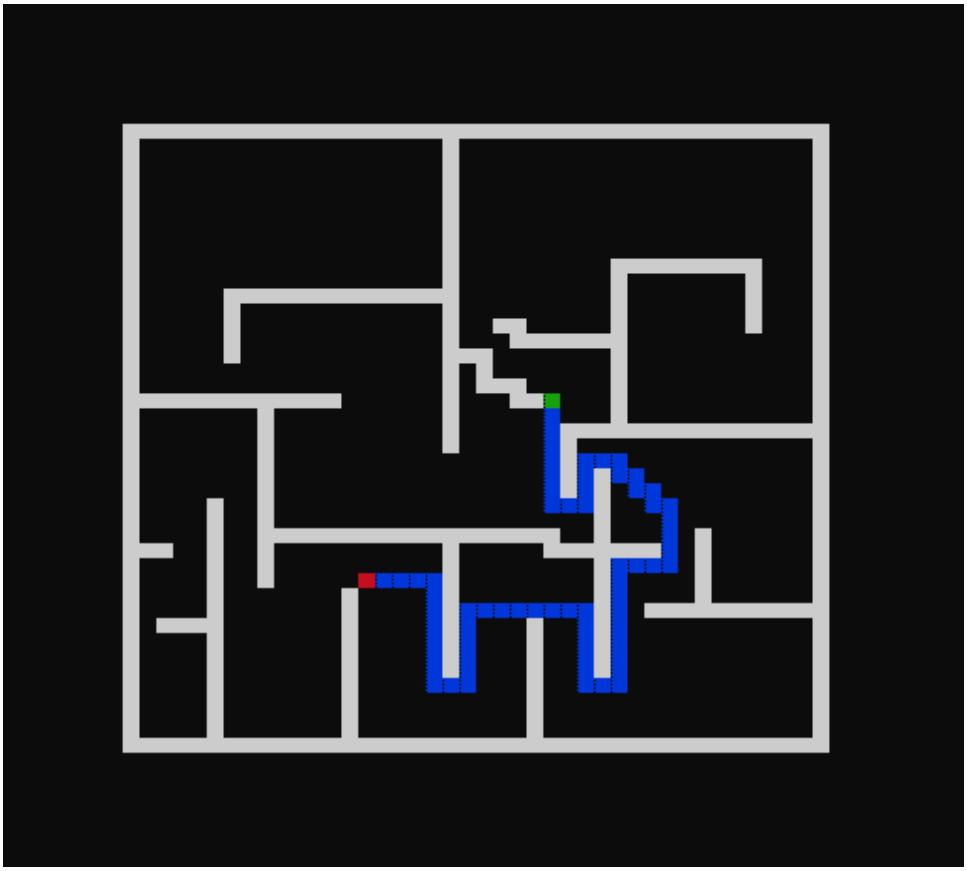
Tämän työn aikana valmistunut kieli sisältää kaikki toivotut ominaisuudet ja monet listatuista lisäominaisuuksista, ja sillä pystyy toteuttamaan monimutkaisia ohjelmia. Sitä on myös helppo laajentaa lisäämällä uusia sisäänrakennettuja objektityyppejä ja funktioita. Komentorivikäyttöliittymä tarvitsee vielä paljon kehitystä, mutta sitäkin voi jo käyttää kielen kääntämiseen.

Tagisysteemiä ei ollut mahdollista lisätä kieleen, mutta sen lisääminen tulevaisuudessa on hyvin nopea prosessi. Muuttuja-arvotyyppiä täytyy muuttaa siten, että se voi sisältää myös tagin arvon tai muita uusia ensimmäisen tason arvoja. Sen jälkeen niille täytyy vain lisätä toiminnallisuus jokaiseen paikkaan, joka vaatii erikoistettua koodia datatyypeille, kuten vertailu ja aritmetiikka.

Asynkroninen suoritus onnistui tietyissä määrin. Koodia voi suorittaa jo omilla säikeillään kutsumalla ympäristön määrittämiä funktioita C++-puolelta, mutta asynkronisuuden määrittäminen skriptin puolella on vielä mahdotonta. Sitä varten täytyy ensin lisätä objektityyppi, joka voi sisältää std::futuren ja käyttää sitä hakemaan palautettu arvo. Vielä parempi olisi, jos tästä objektista saisi tehtyä ensimmäisen tason arvon siten, että käyttäjä ei huomaisi eroa sen ja tavallisen paluuarvon välillä.

7 Käyttö todellisessa ympäristössä

Projektia testattiin kirjoittamalla A*-reitinhakualgoritmi [42], jonka kanssa käyttäjä pystyy olemaan vuorovaikutuksessa. Tällä saatiin testattua montaa projektin osa-aluetta, muun muassa säikeistystä, laskentaa, taulukkojen rakentamista ja käyttäjän vuorovaikutusta. Luodussa pelissä punainen pikseli yrittää etsiä reittiä vihreän pisteen luokse, ja pelaaja voi piirtää ja poistaa seiniä vapaasti. Jos piste törmää seinään tai ei pääse kohteeseen, se etsii uuden sijainnin ja reitin sen luokse.



Kuva 14 Peli, jossa pelaaja on piirtänyt seiniä esteiksi

Koko demo käytti noin 350 riviä skriptiä ja 250 riviä C++-koodia. C++-puolella luotiin funktiot konsolin muokkaamiselle, piirtämiselle ja käyttäjän syötteen lukemiselle, listattuna Kuva 15. Näitä ei

ole vielä mahdollista toteuttaa EMI-kielellä, ja samalla se todistaa, että funktioiden lisääminen toimii tehokkaasti.

```
EMI_REGISTER(Game, GetPixel, getPixel);
EMI_REGISTER(Input, WaitInput, wait_for_input);
EMI_REGISTER(Input, IsKeyDown, is_key_down);
EMI_REGISTER(Input, IsMouseButtonDown, is_mouse_key_down);
EMI_REGISTER(Input, GetMouseX, get_mouse_x);
EMI_REGISTER(Input, GetMouseY, get_mouse_y);
EMI_REGISTER(Global, setupconsole, setup_console);
EMI_REGISTER(Global, writepixel, write_pixel);
EMI_REGISTER(Global, writetext, write_text);
EMI_REGISTER(Global, writetextCentered, write_text_centered);
EMI_REGISTER(Global, render, render_frame);
```

Kuva 15 Demon käyttämien funktioiden sitominen skriptikieleen

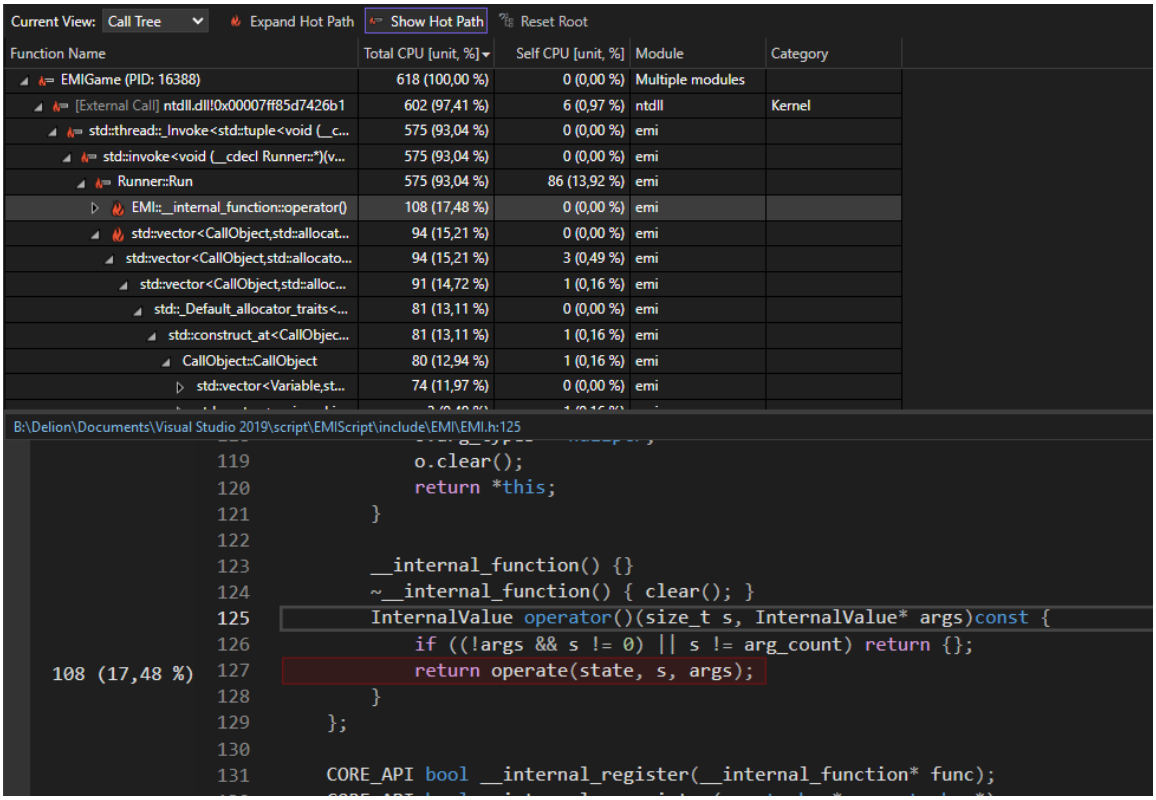
Lopullisen ohjelman suoritukseen ei käytetty erityisen paljon aikaa suhteessa muuhun projektiin, eikä sen suorituskykyä optimoitu juurikaan. Demo käyttää suoraan aikaisemmin kehitettyä C++-implementaatiota A*-algoritmista, mutta sitä ei ole optimoitu EMI-kielelle. Tästä huolimatta sen suorituskyky on huomattavan hyvä ottaen huomioon suuren määrän objektivarauksia. Neljän sadan ruudun kokoisella alueella esteiden kanssa reitinhaku kestää alle 150 ms moderneilla kannettavilla tietokoneilla.

Lopullinen demo vie suurimmillaan noin 5 megabittiä muistia, kaiken virtuaalikoneen käyttämän muistin mukaan lukien. Demon käyttämää muistijälkeä voisi vielä vähentää entisestään vähentämällä objektien varauksia tai vapauttamalla niitä enemmän. Suurin osa tästä muistista on kuitenkin pysyvästi virtuaalikoneen käytössä, sillä sen täytyy pitää muistissa kaikkien muuttujien ja funktioiden nimet. Tätä pystyisi myöhemmin optimoimaan enemmän, mutta tässä vaiheessa sillä ei ole erityisesti merkitystä kielen käyttökohteiden kannalta.

8 Profilointi

Profilointia toteutettiin ohjelmoinnin aikana jatkuvasti, mutta suurin osa siitä toteutettiin vasta demon ohjelmoinnin jälkeen. Profiloinnilla tarkoitetaan tässä yhteydessä ohjelman suorituskyvyn mittaamista ja suorituskykyä heikentävien koodikappaleiden, niin kutsuttujen pullonkaulojen, löytämistä. Tähän tarkoitukseen käytettiin useita eri työkaluja, mukaan lukien Visual Studion profilointityökaluja ja Valgrindin Cachegrind-ohjelmaa [43] Linux-käyttöympäristössä. Näiden työkalujen avulla ohjelman toimintaa oli helppo profiloida ja sen suorituskykyä mitata. Mittauksia tehtiin sekä Windows- että Linux-käyttöjärjestelmissä sekä virtuaaliympäristöissä että todellisilla laitteilla.

Ohjelman profilointi aloitettiin ensin Visual Studion työkaluilla, joiden avulla ohjelman suoritusta seurattiin. Jokaisen ohjelman suorituksen jälkeen profilointityökalusta näki, missä suurimmat suorituskykyä heikentävät koodirivit olivat. Työkalu näyttää tällaiset kohdat suoraan osana koodirivejä värikoodattuina niiden käyttämän CPU-ajan perusteella, kuten Kuva 16 näyttää. Kuvassa näkyy sekä eniten tehoa käyttänyt koodipolku puun muodossa, että koodirivien käyttämä suhteellinen aika. Näiden tietojen pohjalta oli hyvin nopea löytää hitaat koodirivit.



Kuva 16 Visual Studion profilointityökalu

Kun nämä ensimmäiset ja usein vakavimmat pullonkaulat oli paikannettu, niitä oli helppo lähteä poimimaan pois kohdasta ja muokkaamaan tehokkaammin toimiviksi. Jokaisen korjauksen jälkeen profilointityökalu ajettiin uudestaan, kunnes lopputulos oli hyväksyttävä. Profiloinnin aikana huomattiin myös paljon merkitystä debug- ja release-käännöksien välillä. Debug-käännöksessä standardikirjasto vei yllättävän paljon suoritusaikaa, kun sen sijaan kääntäjän optimointien kanssa se on erittäin hyvin optimoitu eikä käytä juuri yhtään ylimääräistä suoritusaikaa.

Tämän työkalun yksi huonoimmista puolesta on kuitenkin se, että se ei ole luotettava debug-käännöksessä Visual Studion debug-ympäristön takia ja release-käännös ei suoraan pysty näyttämään koodiriviä kääntäjän tekemien suurien optimointien seurauksena. Sen sijaan Valgrind Cachegrind -ohjelmalla saatiin enemmän tietoa ohjelman CPU-välimuistin käytöstä. Koska kyseessä on erittäin matalan tason ohjelma, jopa yksittäisellä muistin luvulla on paljon merkitystä pitkällä aikavälillä. Näiden lukujen määrä tulee minimoida ja mitä enemmän CPU pystyy lukemaan ja säilömään tietoa lokaalista välimuistista, sitä vähemmän sen täytyy palata hakemaan uutta dataa RAM:ista, josta dataa on hidasta lukea verrattuna välimuistiin. Jos halutut muistisijainnit sijaitsevat lähekkäin, CPU lukee ne kaikki välimuistiin ja käyttää dataa sieltä.

Cachegrind-ohjelma pystyy kertomaan, mitkä koodirivit käyttäjät eniten muistinlukuja ja missä tapahtuu suurin määrä muistilukuhuteja. Profiloinnin tuloksessa (Kuva 17 jaKuva 18) ohjelman muistin käyttö on yllättävän hyvää. Ohjelman ajon aikana välimuistihuteja ei havaita paljon ja määrä on lähes mitätön määrä kaikkiin muistinlukuihin verrattuna, ja raportoitu hutisuhde on lähellä nollaa. Katso LLd ja D1 miss rate Kuva 18.

```
--------------------------------------------------------------------------------
Ir                    I1mr           ILmr           Dr                    D1mr           DLmr           Dw                    D1mw           DLmw
--------------------------------------------------------------------------------
9,533,655,545 (100.0%) 10,463 (100.0%) 5,302 (100.0%) 3,143,679,122 (100.0%) 41,150 (100.0%) 12,583 (100.0%) 879,291,011 (100.0%) 17,175 (100.0%) 14,325

--------------------------------------------------------------------------------
Ir                    I1mr        ILmr        Dr                    D1mr          DLmr          Dw                    D1mw          DLmw
--------------------------------------------------------------------------------
3,310,446,049 (34.72%) 111 ( 1.06%)  86 ( 1.62%) 1,148,521,853 (36.53%)    98 ( 0.24%)     5 ( 0.04%) 523,591,811 (59.55%)     9 ( 0.05%)     5 ( 0.03%)
  928,950,124 ( 9.74%)   2 ( 0.02%)   2 ( 0.04%)    84,450,290 ( 2.69%)     0              0                    0              0              0
  557,562,291 ( 5.85%)   8 ( 0.08%)   2 ( 0.04%)   354,812,367 (11.29%)    15 ( 0.04%)     0                    0              0              0
  540,482,188 ( 5.67%)   0            0           135,120,712 ( 4.30%)     0              0          67,560,246 ( 7.68%)     0              0
  472,922,162 ( 4.96%)   5 ( 0.05%)   2 ( 0.04%)    67,560,246 ( 2.15%)     1 ( 0.00%)     0                    0              0              0
  422,249,840 ( 4.43%)  36 ( 0.34%)  25 ( 0.47%)   135,120,221 ( 4.30%)    33 ( 0.08%)     0          50,670,233 ( 5.76%)     2 ( 0.01%)     0
  405,360,470 ( 4.25%)   0            0            84,450,208 ( 2.69%)     0              0                    0              0              0
  405,354,456 ( 4.25%)   2 ( 0.02%)   0           101,338,614 ( 3.22%)     2 ( 0.00%)     1 ( 0.01%)  67,559,076 ( 7.68%)     0              0
  371,660,092 ( 3.90%) 827 ( 7.90%) 354 ( 6.68%)   371,657,423 (11.82%)   765 ( 1.86%)     8 ( 0.06%)       1,440 ( 0.00%)    29 ( 0.17%)    19 ( 0.13%)
  287,132,696 ( 3.01%)  14 ( 0.13%)  13 ( 0.25%)   135,121,119 ( 4.30%)     1 ( 0.00%)     0                  359 ( 0.00%)     2 ( 0.01%)     0
  270,241,712 ( 2.83%)   4 ( 0.04%)   1 ( 0.02%)    67,560,388 ( 2.15%)     0              0                    0              0              0
  219,566,971 ( 2.30%)   8 ( 0.08%)   6 ( 0.11%)    33,779,495 ( 1.07%)     0              0          33,779,555 ( 3.84%)     0              0
  202,681,926 ( 2.13%)   2 ( 0.02%)   2 ( 0.04%)    16,890,316 ( 0.54%)     5 ( 0.01%)     0                    0              0              0
t&)
  202,681,244 ( 2.13%)   4 ( 0.04%)   1 ( 0.02%)    67,560,468 ( 2.15%)     0              0                    0              0              0
  202,678,210 ( 2.13%)   1 ( 0.01%)   1 ( 0.02%)    67,559,516 ( 2.15%)     0              0          50,669,468 ( 5.76%)     0              0
const&)
  152,011,911 ( 1.59%)   0            0            67,560,331 ( 2.15%)    12 ( 0.03%)     0          16,890,316 ( 1.92%)     0              0
nst&)
  118,229,984 ( 1.24%)   0            0            50,669,620 ( 1.61%)     8 ( 0.02%)     0          16,889,873 ( 1.92%)     0              0
  101,338,614 ( 1.06%)   2 ( 0.02%)   1 ( 0.02%)    33,779,538 ( 1.07%)     0              0                    0              0              0
   84,449,902 ( 0.89%)   4 ( 0.04%)   1 ( 0.02%)             0              0              0                    0              0              0
lue const&)
   84,448,500 ( 0.89%)   1 ( 0.01%)   1 ( 0.02%)    16,889,700 ( 0.54%)     0              0                    0              0              0
td::function<int ()>>(std::integer_sequence<unsigned long>)::{lambda(void*, unsigned long, InternalValue*)#1}::_FUN(void*, unsigned long, InternalValue*)
   67,560,998 ( 0.71%)   1 ( 0.01%)   0            33,780,499 ( 1.07%)     0              0          33,780,499 ( 3.84%)     0              0
   50,669,108 ( 0.53%)   0            0            33,779,400 ( 1.07%)     1 ( 0.00%)     0          16,889,700 ( 1.92%)     0              0
std::integer_sequence<unsigned long>)::{lambda(void*, unsigned long, InternalValue*)#1}::_FUN(void*, unsigned long, InternalValue*)
   33,779,400 ( 0.35%)   1 ( 0.01%)   0            16,889,700 ( 0.54%)     0              0                    0              0              0
   16,889,700 ( 0.18%)   0            0            16,889,700 ( 0.54%)     0              0                    0              0              0
_Any_data const&)
   16,889,700 ( 0.18%)   0            0                     0              0              0                    0              0              0
 std::function<int ()>>(std::integer_sequence<unsigned long>)::{lambda(void*, unsigned long, InternalValue*)#1}::_FUN(void*, unsigned long, InternalValue
```

Kuva 17 Cachegrindin tulos funktiokohtaisesti luokiteltuna

```
inux-gnu/libstdc++.so.6.0.28)
==1927==    by 0x4882F51: VM::GetReturnValue(unsigned long) (atomic_futex.h:102)
==1927==    by 0x10AB75: main (EMI.h:210)
==1927==
==1927== I   refs:      16,485,051,838
==1927== I1  misses:            10,824
==1927== LLi misses:             5,310
==1927== I1  miss rate:           0.00%
==1927== LLi miss rate:           0.00%
==1927==
==1927== D   refs:       6,956,358,722  (5,436,157,209 rd   + 1,520,201,513 wr)
==1927== D1  misses:            59,046  (       41,603 rd   +        17,443 wr)
==1927== LLd misses:            26,988  (       12,587 rd   +        14,401 wr)
==1927== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==1927== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==1927==
==1927== LL refs:               69,870  (       52,427 rd   +        17,443 wr)
==1927== LL misses:             32,298  (       17,897 rd   +        14,401 wr)
==1927== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Kuva 18 Cachegrindin tuloksen yhteenveto

Suurimmat ongelmat olivat objektien referenssilaskennassa ja muuttujien vertailussa. Vertailu korjattiin siirtämällä entinen funktio kokonaisuudessaan virtuaalikoneen silmukkaan, mikä antaa kääntäjälle paremman mahdollisuuden optimoida sitä. Referenssilaskentaa ei pysty korjaamaan, vaan se vaatii roskankeruualgoritmin vaihdon.

Tarkkoja vertailuja optimoinnin vaikutuksesta ohjelmaan ei ole, mutta demo pyörii profiloinnin mahdollistaman optimoinnin jälkeen huomattavasti tehokkaammin. Demon ohjelmoinnin aikana suoritettiin myös niin kutsuttua muistiprofilointia, jonka merkityksenä on löytää kaikki muistivuorot ja optimoida muistin käyttöä ohjelmassa. Muistivuotoja korjattiin sitä mukaa, kun niitä löydettiin, ja tuloksena lopullisen ohjelman profiloinnissa ei enää löydetty uusia muistivuotoja.

Muistinvarauksen aiheuttamaa vaikutusta ohjelman suoritukseen ei mitattu tässä projektissa. Nykyinen muistioptimointi on suhteellisen hyvä ja sen parantaminen vaatisi huomattavasti enemmän aikaa kuin projektiin on käytettävissä. Objektit varataan vielä kekomuistista ja ei ole varmuutta niiden sijainnista toisiinsa nähden. Jos ne saisi varattua vierekkäin, muistinhaku olisi entistä nopeampaa. Tämä vaatii kuitenkin muutoksia allokaattorin puolella ja voi aiheuttaa vaikeuksia muuttujien antamien vaatimuksien takia, joten sitä ei vielä toteutettu.

Vastaavaa profilointia tulee jatkaa myös ohjelman jatkokehityksen aikana. Parempi ratkaisu olisi, jos kehityksen tueksi kehitettäisiin automaattinen profilointi- ja testaustyökalu, joka ajaisi useamman yksittäisen testin ohjelmalle jokaisen muutoksen jälkeen.

9 Jatkopohdintoja

EMI tarvitsee ehdottomasti IDE:n ennen kuin se on käytettävissä peleissä. Tähän tarkoitukseen voi joko kehittää oman editorin, mutta parempi idea on tehdä lisäosa tunnettuihin ja yleisiin ympäristöihin, kuten VS Code. Esimerkiksi juuri VS Code tarjoaa loistavat ohjeet lisäosan luomiseen ja intellisensen lisäämiseen kieleen. Vaihtoehtoinen lähestymistapa on luoda graafinen ohjelmointiliittymä Unreal Enginen käyttämien Blueprinttien tyylillä. Tämä olisi erityisen hyödyllinen esimerkiksi lapsien opetuksessa ohjelmoinnin perusteisiin. Se on myös helpommin ymmärrettävissä pelitaiteilijoille ja kenttäsuunnittelijoille. Jotta graafisen liittymän voi lisätä, täytyy luoda uusi parserirajapinta tai sallia AST-rakenteen lähettäminen kääntäjälle.

Toinen mahdollisuus olisi kehittää muita kehitystyökaluja kieltä varten. Tällä hetkellä virtuaalikone ei tue keskeytyksiä tai käsky tai rivi kerrallaan etenemistä. Olisi myös hyödyllistä, jos käyttäjä pystyy käynnistämään virtuaalikoneen käyttämällä eri asetuksia, esimerkiksi debug-symbolien kanssa tai ilman, käyttökohteen mukaan valittuna.

Mielenkiintoisia ongelmia tuli kääntäjän tekemistä optimoinneista NaN-boxingia käyttävään luokkaan. Kaikki C++-kääntäjät optimoivat NaN-boxingia käyttäviä arvoja liikaa ja tämän seurauksena jokaisella kerralla, kun arvoja sisältävä vektori uudelleenallokoitiin, se ei kutsunut arvoluokan konstruktoria uusille elementeille. Tämä johti siihen, että vektori yritti kopioida arvoja alustamattomaan muistiin. Kun tämä ongelma oli lopulta löydetty, kääntäjät pakotettiin alustamaan muisti aina Undefined-arvoon. Undefined-arvolla virtuaalikone tunnistaa aina alustamattoman arvon ja pystyy käyttämään sitä oikein laskuissa.

Mikäli virtuaaliympäristön muistin käyttöä ja suorituskykyä optimoisi vielä enemmän, sitä pystyisi ajamaan myös sulautetuissa järjestelmissä tai matalatehoisissa koneissa. Näitä käyttökohteita ei ole vielä testattu, mutta sen pitäisi olla mahdollista ottaen huomioon suorituskyvyn ja muistinkäytön. Hyötynä tästä olisi nopeutettu ohjelmointikyky, sillä koodia ei joka kerta tarvitsisi kääntää konekielelle ja konekieltä asettaa flash-muistiin. Lopulliseen tuotantoon tämä ei kuitenkaan sovi, sillä bittikoodin suorittaminen ja kääntäminen on todennäköisesti liian hidasta suljetuissa järjestelmissä.

Tämän sijaan kielelle on löydetty useita käyttökohteita muiden ohjelmien testauksessa. Pienien muutoksen jälkeen, EMI-kielellä pystyy kutsumaan funktioita ohjelmakirjastojen (DLL) funktiotaulukoista. Funktioiden kutsumista pystyisi esimerkiksi käyttämään ohjelmakirjastojen nopeaan ja

varmaan yksikkötestaukseen. Mahdollinen käyttökohde olisi myös automaatiotestaus Jenkins-alustalla, missä kieltä pystyttäisiin käyttämään lisäämään dynaamisia testejä eri ohjelmille automatisoitujen robottitestien tapaan.

Kielelle on siis useita mahdollisia käyttökohteita tulevaisuudessa ja sen kehitystä on toisin sanoen jatkettava. Seuraavana vaiheena kehityksessä on asynkronisen suorituksen jatkokehitys, visuaalisen käyttöliittymän ohjelmointi ja IDE:n rakentaminen. Myös testaus pelimoottoreissa ja käyttö pelien yhteydessä olisi suositeltavaa toteuttaa mahdollisimman pian kehityksen yhteydessä.

Lopullinen työ sisältää monipuolisen ja käytettävän kielen, jonka suorituskyky on kelvollinen yksinkertaisiin ratkaisuihin. Siinä on vielä paljon mahdollisuuksia jatkokehitykselle ja uusien työkalujen lisäämiseksi kehityksen helpottamiseksi. Projektin alussa asetetut tavoitteet saavutettiin kuitenkin täysin.

Lähteet

1. Ritchie DM. The Development of the C Language. SIGPLAN Not. 1993;: 201–208.
2. ECMAScript® 2023 Language Specification. [Online]. [Haettu 2023 Maaliskuu 19]. Saatavilla: https://tc39.es/ecma262/#sec-overview.
3. UDK - Design Goals of UnrealScript. [Online]. [Haettu 2023 Maaliskuu 19]. Saatavilla: https://docs.unrealengine.com/udk/Three/UnrealScriptReference.html#Design%20goals%20of%20_UnrealScript.
4. van Deursen A, Klint P, Visser J. Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Not. 2000 June; 35(6): 26–36.
5. Geeks for Geeks. [Online].; 2023 [Haettu 2023 Syyskuu 3]. Saatavilla: https://www.geeksforgeeks.org/what-is-high-level-language/.
6. Patterson DA, Hennessy JL. Instructions: Language of the Computer. In Computer Organization and Design: The Hardware/Software Interface.; 2014. s. 62-163.
7. Ousterhout JK. Scripting: higher level programming for the 21st Century. Computer. 1998; 31(3): 23-30.
8. The Elixir Team. Elixir. [Online].; 2021 [Haettu 2023 Syyskuu 12]. Saatavilla: https://elixir-lang.org/.
9. Stroustrup B. The C++ programming language. 3rd ed.: Addison-Wesley; 1997.
10. Geeks for Geeks. [Online].; 2023 [Haettu 2024 Maaliskuu 24]. Saatavilla: https://www.geeksforgeeks.org/static-and-dynamic-linking-in-operating-systems/.
11. Kaijanaho AJ. TIES448 Kääntäjätekniikka. [Online].; 2016 [Haettu 2024 Syyskuu 15]. Saatavilla: https://tim.jyu.fi/view/kurssit/tie/kate/2016/luentomatskut/Luento-1#k%C3%A4%C3%A4nt%C3%A4j%C3%A4n-rakenne.
12. Gaurav S. Scaler Topics. [Online].; 2024 [Haettu 2024 Helmikuu 22]. Saatavilla: https://www.scaler.com/topics/interpreted-vs-compiled-language/.
13. freeCodeCamp. freeCodeCamp.org. [Online].; 2020 [Haettu 2023 Maaliskuu 23]. Saatavilla: https://www.freecodecamp.org/news/just-in-time-compilation-explained/.
14. IBM. IBM. [Online].; 2024 [Haettu 2024 Maaliskuu 24]. Saatavilla: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler.
15. Parr T. ANTRL. [Online].; 2023 [Haettu 2024 February 22]. Saatavilla: https://www.antlr.org/.
16. Free Software Foundation, Inc. GNU Bison. [Online].; 2014 [Haettu 2024 February 22]. Saatavilla: https://www.gnu.org/software/bison/.
17. Mogensen TÆ. comp.compilers. [Online].; 2010 [Haettu 2024 Tammikuu 22]. Saatavilla: https://compilers.iecc.com/comparch/article/10-03-006.
18. Paulson JA. Harvard School of Engineering and Applied Sciences. [Online].; 2018 [Haettu 2023 Lokakuu 12]. Saatavilla: https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec06-LR-Parsing.pdf.
19. Johnson M. Stanford University. [Online].; 2012 [Haettu 2023 Lokakuu 28]. Saatavilla: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf.
20. Sutter H, Alexandrescu A. C++ Coding Standards: Addison-Wesley.

21. Paulson JA. Harvar School of Engineering and Applied Sciences. [Online].; 2019 [Haettu 2024 Tammikuu 29]. Saatavilla: https://groups.seas.harvard.edu/courses/cs153/2019fa/lectures/Lec14-Type-checking.pdf.

22. Toal R. Loyola Marymount University. [Online]. [Haettu 2024 Tammikuu 12]. Saatavilla: https://cs.lmu.edu/~ray/notes/ir/.

23. Meiners J, Pendleton R. Write your Own Virtual Machine. [Online].; 2018 [Haettu 2024 Maaliskuu 24]. Saatavilla: https://www.jmeiners.com/lc3-vm/#what-is-a-virtual-machine-.

24. Bergia A. Andrea Bergia's Website. [Online].; 2015 [Haettu 2023 Lokakuu 15]. Saatavilla: https://andreabergia.com/blog/2015/03/stack-based-virtual-machines-1/.

25. Rahti M. Creating the Bolt Compiler. [Online].; 2020 [Haettu 2024 Tammikuu 17]. Saatavilla: https://mukulrathi.com/create-your-own-programming-language/data-race-dataflow-analysis/.

26. Eisl J, Grimmer M, Simon D, Würthinger T, Mössenböck H. Trace-based register allocation in a JIT compiler. In Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools; 2016.

27. Chaitin GJ, Auslander MA, Chandra AK, Cocke J, Hopkins ME, Markstein PW. Register allocation via coloring. Computer Languages. 1981; 6(1): 47-57.

28. Braun M, Buchwald S,HS, Leißa R, Mallon C, Zwinkau A. Simple and Efficient Construction of Static Single Assignment Form. In Compiler Construction; 2013. p. 102-122.

29. Callan S. Elixir School. [Online].; 2021 [Haettu 2023 10 2]. Saatavilla: https://elixirschool.com/en/lessons/basics/pipe_operator.

30. Winters T. What is ABI, and What Should WG21 Do About It? Paper. The C++ Standards Committee, LEWG, EWG, WG21; 2021. Report No.: P2028R0.

31. Austin C. Binary-compatible C++ Interfaces. [Online].; 2002 [Haettu 2024 Lokakuu 15]. Saatavilla: https://chadaustin.me/cppinterface.html.

32. Aho AV, Sethi R, Ullman JD. Compilers, principles, techniques, and tools. Rep. with corrections ed.: Reading, Mass. : Addison-Wesley Pub. Co.; 1986.

33. 011c. [GitHub Gist].; 2022 [Haettu 2024 Tammikuu 7]. Saatavilla: https://gist.github.com/o11c/6b08643335388bbab0228db763f99219.

34. cppreference.com. [Online].; 2023 [Haettu 2024 Tammikuu 5]. Saatavilla: https://en.cppreference.com/w/cpp/string/basic_string_view.

35. cppreference.com. [Online].; 2022 [Haettu 2023 Lokakuu 14]. Saatavilla: https://en.cppreference.com/w/c/string/byte/isalnum.

36. greg. JSMachines. [Online].; 2012 [Haettu 2023 Syyskuu 27]. Saatavilla: https://jsmachines.sourceforge.net/machines/lalr1.html.

37. University of Illinois. The Grainger College of Engineering. [Online].; 2012 [Haettu 2023 Lokakuu 28]. Saatavilla: https://courses.engr.illinois.edu/cs421/sp2012/lectures/lecture7-8.pdf.

38. Boehm H, Weiser M, Demers A. A garbage collector for C and C++. [Online].; 2023 [Haettu 2024 Maaliskuu 2]. Saatavilla: https://www.hboehm.info/gc/.

39. Nystrom R. Crafting Interpreters: Genever Benning; 2021.

40. cppreference.com. [Online].; 2024 [Haettu 2024 Maaliskuu 3]. Saatavilla: https://en.cppreference.com/w/cpp/language/new.

41 Schinz M. Advanced Compiler Construction. [Online].; 2016 [Haettu 2024 Maaliskuu 3]. . Saatavilla: https://www.epfl.ch/labs/lamp/wp-content/uploads/2019/01/acc16_11_interpreters-vms_4.pdf.

42 Wikipedia. A* search algorithm. [Online].; 2024 [Haettu 2024 Maaliskuu 9]. Saatavilla: . https://en.wikipedia.org/wiki/A*_search_algorithm.

43 Nethercote N. Valgrind Documentation. [Online].; 2006 [Haettu 2024 Maaliskuu 3]. . Saatavilla: https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/RuleRefinement/bin/valgrind-3.2.0/docs/html/cg-manual.html.

Liitteet

Käskysetti

Käsky	Argumentit	Selitys
Noop	0	Tyhjä koodi, ei toiminnallisuutta
Break	-	Ei käytössä
LoadNumber	2	Lataa valitun indeksin numerotaulukosta merkattuun rekisteriin
LoadImmediate	2	Lataa parametrinä annetun pienen kokonaisluvun merkattuun rekisteriin
LoadString	2	Lataa valitun indeksin merkkitaulukosta merkattuun rekisteriin
LoadSymbol	2	Käytetään dynaamiseen linkitykseen, lataa globaalin muuttujan rekisteriin
StoreSymbol	2	Tallentaa annetun rekisterin globaaliin muuttujaan
LoadProperty	2+1	Lataa annetun indeksin rekisterissä olevasta objektista ja tallentaa tuloksen kohderekisteriin.
StoreProperty	2+1	Tallentaa kohderekisterin annetun rekisterin osoittaman objektin valittuun indeksiin.
Call	-	Ei käytössä
CallSymbol	3+1	Kutsuu tuntematonta symbolia käyttäen parhainta kolmesta seuraavasta käskystä
CallInternal	3+1	Kutsuu sisäistä C++-funktiota. Ensimmäinen parametri kertoo palautusarvon kohteen, toinen halutun funktion paikallisen indeksin ja kolmas argumenttien määrän
CallExternal	3+1	Kutsuu ulkoista C++-funktiota. Ensimmäinen parametri kertoo palautusarvon kohteen, toinen halutun funktion paikallisen indeksin ja kolmas argumenttien määrän
CallFunction	3+1	Kutsuu käännettyä funktiota. Ensimmäinen parametri kertoo palautusarvon kohteen, toinen halutun funktion paikallisen indeksin ja kolmas argumenttien määrän
PushUndefined	1	Asettaa kohderekisterin määräämättömäksi
PushBoolean	2	Asettaa rekisteriin halutun totuusarvon
PushArray	2	Luo uuden taulukon valittuun rekisteriin ja alustaa sen annettuun kokoon
PushTypeDefault	2	Asettaa rekisterin annetun tyypin perusarvoon
PushObjectDefault	2	Luo valitun objektin kohderekisteriin käyttäen objektikenttien oletusarvoja
InitObject	3+1	Luo valitun objektin kohderekisteriin käyttäen annettuja rekistereitä objektikenttien täyttämiseen
Copy	2	Kopioi rekisterin kohderekisteriin
JumpBackward	1	Hyppää taaksepäin annetun käskymäärän
JumpForward	1	Hyppää eteenpäin annetun käskymäärän

Jump	1	Hyppää hyppytaulukon indeksin antamaan osoitteeseen
JumpEq	2	Hyppää annettuun indeksiin, jos rekisteri on tosi
JumpNeg	2	Hyppää annettuun indeksiin, jos rekisteri ei ole tosi
Return	2	Palaa kutsupinossa edelliseen funktioon ja kopioi annetun rekisterin funktion palautusrekisteriin
Equal	3	Vertaa kahta rekisteriä ja asettaa kohteen
NotEqual	3	Vertaa kahta rekisteriä ja asettaa kohteen
Less	3	Vertaa kahta rekisteriä ja asettaa kohteen
Greater	3	Vertaa kahta rekisteriä ja asettaa kohteen
LessEqual	3	Vertaa kahta rekisteriä ja asettaa kohteen
GreaterEqual	3	Vertaa kahta rekisteriä ja asettaa kohteen
Not	2	Ottaa annetun rekisterin totuusarvon, kääntää sen ja asettaa kohteeseen
And	3	Vertaa kahta rekisteriä ja palauttaa loogisen AND-arvon
Or	3	Vertaa kahta rekisteriä ja palauttaa loogisen OR-arvon
NumAdd	3	Lisää kaksi numerorekisteriä ja asettaa tuloksen kohteeseen
NumSub	3	Vähentää kaksi numerorekisteriä ja asettaa tuloksen kohteeseen
NumMul	3	Kertoo kaksi numerorekisteriä ja asettaa tuloksen kohteeseen
NumDiv	3	Jakaa kaksi numerorekisteriä ja asettaa tuloksen kohteeseen
Add	3	Lisää kahden rekisterin arvot ja asettaa tuloksen kohteeseen
Sub	3	Vähentää kahden rekisterin arvot ja asettaa tuloksen kohteeseen
Mul	3	Kertoo kahden rekisterin arvot ja asettaa tuloksen kohteeseen
Div	3	Jakaa kahden rekisterin arvot ja asettaa tuloksen kohteeseen
PreMod	3	Suurentaa tai vähentää rekisterin arvoa yhdellä ja asettaa tuloksen kohteeseen
PostMod	3	Suurentaa tai vähentää rekisterin arvoa yhdellä ja asettaa alkuperäisen arvon kohteeseen
StrAdd	3	Lisää kaksi merkkijonoa yhteen ja asettaa tuloksen kohteeseen
PushIndex	3	Lisää rekisterin arvon annettuun taulukkoon
StoreIndex	3	Tallentaa rekisterin arvon annetun taulukon indeksiin
LoadIndex	3	Lataa arvon kohderekisteriin annetusta taulukon indeksistä
RangeFor	3	Tarkistaa indeksirekisterin arvoa annettuun rekisteriin ja jos indeksi on pienempi, sen arvoa nostetaan yhdellä. Asettaa kohderekisteriin totuusarvon operaation mukaan
RangeForVar	3+1	Tarkistaa indeksirekisterin arvoa annettuun rekisteriin ja jos indeksi on pienempi, sen arvoa nostetaan yhdellä. Asettaa kohderekisteriin totuusarvon operaation mukaan. Asettaa annettuun rekisteriin arvon muuttujan tyypin perusteella

EMIScript näytteitä

```
public def input() {

    var topOffset = Top;
    var leftOffset = Left;

    while (true) {
        var inp : number = Input.WaitInput();
        if (inp == -1) continue;
        var in_x : number = Input.GetMouseX();
        var in_y : number = Input.GetMouseY();
        var x = in_x - leftOffset;
        var y = in_y - topOffset;
        if (inp == 1) {
            if (x >= 0 && x < x_size && y >= 0 && y < y_size) {
                writepixel(in_x, in_y, wall, 0);
            }
        } else if (inp == 2) {
            if (x >= 0 && x < x_size && y >= 0 && y < y_size) {
                writepixel(in_x, in_y, empty, 0);
            }
        }
    }
}

object MapContainer
{
    keys : array;
    values : array;
    equality : function;
}

extend Map:

public def insert(map : MapContainer, key, value) {
    var itemFound = false;
    var index = 0;
    for (item in map.keys) {
        if (map.equality(item, key)) {
            index = _index_;
            itemFound = true;
            break;
        }
    }
    if (!itemFound) {
        index = Array.Size(map.keys);
        Array.Push(map.keys, key);
    }

    Array.Resize(map.values, Array.Size(map.keys));
    map.values[index] = value;
}
```