C-ohjelmointikielen korvaaminen muilla ohjelmointikielillä

Jaakko Hannikainen

Helsinki 28.2.2020

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Jaakko Hannikainen

Työn nimi — Arbetets titel — Title

C-ohjelmointikielen korvaaminen muilla ohjelmointikielillä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	28.2.2020	42 sivua + 5 liitesivua

Tiivistelmä — Referat — Abstract

Ohjelmointikielen valinta on tärkeä osa ohjelmistoprojektien toteutusta. Vaikka ohjelmointikielet uudistuvat nopeaan tahtiin, nykypäivänä on yhä tavallista valita ohjelmiston toteutukseen C-ohjelmointikieli, joka on standardoitu yli 30 vuotta sitten. Tutkielmassa tutkitaan syitä, miksi C on nykypäivänä vieläkin laajassa käytössä uudempien ohjelmointikielten sijaan.

Tutkielmassa C:hen verrattaviksi ohjelmointikieliksi valitaan Ada, C++, D, Go sekä Rust. Kaikki viisi kieltä ovat tehokkaita. Tämän lisäksi jokaisen kielten historiassa on ollut tavoitteena korvata C:n käyttö. Ohjelmointikieliä verrataan C:hen suorituskyvyn, muistinkäytön sekä C-yhteensopivuuden osalta. Tämän lisäksi tutkielmassa selvitetään tärkeimpiä C:n ominaisuuksia sekä C:n kehitettävissä olevia ominaisuuksia. Tuloksia käytetään uuden Purkka-ohjelmointikielen suunnitteluun.

Muut ohjelmointikielet todetaan suorituskykymittauksissa C:tä hitaammiksi. Tämän lisäksi muiden ohjelmointikielten ominaisuudet, kuten automaattisen muistinhallinnan, todetaan aiheuttavan ongelmia C-yhteensopivuudelle.

C:n tärkeimmiksi ominaisuuksiksi nousevat esiin yksinkertaisuus, tehokkuus sekä alustariippumattomuus. Nämä ominaisuudet otetaan huomioon Purkka-kielen suunnittelussa, jossa painotetaan näiden lisäksi yhteensopivuutta C-ohjelmointikielen kanssa.

Tutkielmaa varten kehitetty Purkka-kieli on suunniteltu C:n kaltaiseksi ohjelmointikieleksi, jossa on muutettu C:n syntaksia yksinkertaisemmaksi ja johdonmukaisemmaksi. Suorituskykymittauksissa todetaan, että Purkan muutokset C:hen eivät aiheuta suoritusaikaisia rasitteita. Koska Purkka-kieli käännetään C:ksi, se on mahdollisimman yhteensopiva nykyisten kääntäjien kanssa.

ACM Computing Classification System (CCS):
Software and its engineering → Software notations and tools → General programming languages → Language types → Imperative languages
Professional topics → Management of computing and information systems → Software management → Software selection and adaptation

Avainsanat — Nyckelord — Keywords
C, ohjelmointikielet

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 C-ohjelmointikielen taustaa **3**

2.1 C-ohjelmointikieli lyhyesti . . . 3

2.2 C-ohjelmointikielen historiaa ja nykypäivää . . . 5

2.3 Tärkeimmät C-ohjelmointikielen ominaisuudet . . . 7

2.4 Kehitettävissä olevat ominaisuudet C-ohjelmointikielessä . . . 9

3 Ohjelmointikielten vertailun määritelmät **12**

3.1 Ohjelmointikielten vertailun kriteerit . . . 12

3.2 C:hen verrattavissa olevat ohjelmointikielet . . . 13

3.3 Kielten suosioon vaikuttavat tekijät . . . 14

4 Ohjelmointikielten vertailu **16**

4.1 Yleisiä vertailtavien ohjelmointikielten ominaisuuksia . . . 16

4.2 Ada . . . 17

4.3 C++ . . . 17

4.4 D . . . 19

4.5 Go . . . 19

4.6 Rust . . . 20

4.7 Yhteenveto . . . 23

5 Purkka-ohjelmointikieli **26**

5.1 Purkan suunnitteluperiaatteet . . . 26

5.2 Tyypit . . . 27

5.3 Syntaksi . . . 29

5.4 C-yhteensopivuus . . . 30

6 Uuden ohjelmointikielen vertaaminen C:hen **32**

6.1 Vertailun tulokset . . . 32

6.2 Johtopäätökset ja vertailun arviointi . . . 35

7 Yhteenveto **37**

Lähteet **38**

Liitteet

1 Mittaukset

2 Purkka-kielen tyypit

1 Johdanto

C (ISO/IEC, 2018) on nykypäivänä yksi eniten käytetyimmistä ohjelmointikielistä. C on ollut vallitseva ohjelmointikieli järjestelmäohjelmoinnissa kielen alkuajoista lähtien. Useita ohjelmointikieliä on luotu historian saatossa, joiden oli tarkoitus syrjäyttää C, mutta C on vieläkin johtavana kielenä varsinkin sulautetuissa järjestelmissä ja Unix-pohjaisten käyttöjärjestelmien vallitsevana ohjelmointikielenä. C on myös käytössä Windows-käyttöjärjestelmäperheen ydinkomponenttien toteutuksessa.

Tutkielmassa selvitetään C:n ominaisuuksia, joiden takia se on ollut suosituimpien ohjelmointikielten joukossa vuosikymmeniä, kuten myös ominaisuuksia, joita C:stä voisi kehittää. Vaihtoehtoisista kielistä selvitetään, mitkä ominaisuudet ovat voineet estää kielen käytön C:n sijaan uusissa ja olemassa olevissa projekteissa ja mitkä ominaisuudet ovat taas olleet parannuksia C:n ominaisuuksiin verrattuna. Tutkielmassa suunnitellaan näiden tulosten pohjalta uusi ohjelmointikieli, Purkka.

C:n vaihtoehdoiksi tutkitaan seuraavia tehokkaaseen ohjelmointiin tarkoitettuja kieliä: Ada (ISO/IEC, 2012), C++ (ISO/IEC, 2017b), D (D Language Foundation, 2020c), Go (Google, Inc., 2020c) sekä Rust (Rust Project Developers, 2020c). Näistä kielistä tutkitaan, mikä tai mitkä ominaisuudet ovat estäneet C:n korvaamisen ja mitkä ominaisuudet ovat olleet parannuksia C:hen verrattuna. Vertailun tuloksia käytetään uuden ohjelmointikielen suunnitteluun, jossa otetaan tavoitteeksi luoda C:tä parempi ohjelmointikieli tutkielman määrittelyjen puitteissa.

Kaikkia verrattavia kieliä tutkitaan sekä analyyttisesti että suorituskykymittausten muodossa. Suorituskykymittauksiin käytetään Benchmarks Gamea (Gouy, 2020a), johon on toteutettu lukuisia pieniä ohjelmia suorituskyvyn mittaamiseen. Vaikka suorituskykymittaukset eivät välttämättä tarjoa absoluuttisia vastauksia kielten paremmuudesta, niitä voidaan käyttää suuntaa-antaviin arvioihin. D on ainoa tutkielman käsittelemä kieli, jota Benchmarks Game ei sisällä. Benchmarks Gamen Purkka-ohjelmat on toteutettu nopeimpien C-ohjelmien pohjalta, jotta ohjelmien arkkitehtuuriset valinnat eivät vaikuta Purkan ja C:n vertailuun.

Tutkielman toisessa luvussa käsitellään C-ohjelmointikieltä. Kielestä käsitellään perusteiden lisäksi kielen historiaa ja nykypäivää, tärkeimpiä ominaisuuksia ja kehityskohteita. Historian käsitteleminen mahdollistaa ymmärryksen siitä, mitkä C:n ominaisuudet ovat tärkeitä kielen nykykäytön kannalta ja mitkä ovat jäänteitä historiallisista syistä. Tärkeimpien ominaisuuksien ja kehitettävien ominaisuuksien tunnistamiseen käytetään ohjelmointikielten tulevaisuutta käsittelevää artikkelia

The Next 7000 Programming Languages (Chatley, Donaldson, & Mycroft, 2019), C-kielen suosiota käsittelevää artikkelia *Some Were Meant For C* (Kell, 2017) sekä Dennis Ritchien artikkelissa *The Development of the C Language* (Ritchie, 1993) esiin nostettuja C:n ominaisuuksia.

Kolmannessa luvussa määritetään toisen luvun esiin nostamien ominaisuuksien pohjalta vertailukriteerit kielten vertaamiseen, käsitellään lyhyesti tutkielmaan valittuja vertailtavia kieliä sekä käsitellään erilaisia yleisiä ohjelmointikielten valintaan liittyviä tekijöitä. Edellä mainitut kielet valitaan, sillä ne ovat suosittuja (TIOBE software BV, 2020) sekä kunkin kielen historiassa on ollut tavoitteena korvata C:n käyttö. Ohjelmistojen toteutuskielen valintaprosessia käsitellään artikkelin *Empirical Analysis of Programming Language Adoption* (Meyerovich & Rabkin, 2013) avulla, jossa tutkitaan kyselyillä erilaisia syitä ohjelmointikielen valintaan.

Neljännessä luvussa esitellään vertailtavat kielet ja käsitellään näiden kielten ominaisuuksien tehokkuutta ja yhteensopivuutta C:n kanssa. Kaikki vertailtavat kielet sisältävät ominaisuuksia, jotka haittaavat yhteensopivuutta C:n kanssa. Tämän lisäksi suorituskykymittauksista ilmenee, kuinka kaikkien kielten toteutukset käyttävät enemmän muistia Benchmarks Gamen vertailuissa.

Viidennessä luvussa esitellään tutkielmaa varten kehitetty uusi ohjelmointikieli, Purkka. Luvussa käsitellään ensin suunnitteluperiaatteita, joita noudattamalla Purkasta voisi tulla C:tä parempi ohjelmointikieli, jonka jälkeen kielestä esitellään yksittäisiä C:stä poikkeavia ominaisuuksia. Kielen suunnittelussa on painotettu erityisesti C-yhteensopivuutta esimerkiksi C-kielen esikäsittelijätuen kautta. Suunnittelussa on pyritty parantamaan C:tä erityisesti syntaksin osalta, mutta myös tarjoamalla vahvempaa tyypitystä. Purkka-kieli käännetään C-kieleksi, jotta kielen yhteensopivuus olisi mahdollisimman hyvä nykyisten ohjelmistojen yhteydessä.

Kuudennessa luvussa verrataan Purkka-kielellä toteutettuja Benchmarks Gamen ohjelmia muihin kieliin. Suorituskykymittauksissa Purkka pysyy yhtä tehokkaana kuin C, mutta Purkan lähdekooditiedostot ovat noin kuusi prosenttia pienempiä verrattuna vastaaviin C-tiedostoihin. Kuudennessa luvussa myös arvioidaan tutkielman oikeellisuutta ja pohditaan jatkotutkimuskohteita.

Seitsemännessä luvussa kerrataan tutkielman tulokset.

2 C-ohjelmointikielen taustaa

C on matalan tason ohjelmointikieli, jossa yhdistyy yksinkertaisuus, tehokkuus ja alustariippumattomuus. C on nykypäivänä yksi maailman käytetyimmistä ohjelmointikielistä (GitHub, Inc., 2019; TIOBE software BV, 2020) johtuen kielen ominaisuuksista sekä historiallisista syistä. Tässä luvussa kerrotaan C-ohjelmointikielestä, sen historiasta ja nykypäivästä sekä käsitellään mahdollisia muutoksia, joilla C:tä voisi parantaa.

2.1 C-ohjelmointikieli lyhyesti

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, World");
    return 0;
}
```

Ohjelma 2.1: Yksinkertainen hello world -ohjelma toteutettuna C:llä.

C on Dennis Ritchien 1970-luvun taitteessa kehittämä yksinkertainen matalan tason ohjelmointikieli (Ritchie, 1993), jota käytetään nykypäivänä erityisesti järjestelmäohjelmoinnissa sekä sulautetuissa järjestelmissä. C ei aseta ohjelmoijalle rajoituksia muistinkäsittelyyn, mikä mahdollistaa tehokkaan mutta turvattoman ohjelmoinnin. C on TIOBE software BV:n (2020) mukaan yksi tällä hetkellä käytetyimmistä ohjelmointikielistä. C ei ole vuoden 1989 ANSI-standardin jälkeen muuttunut mainittavasti (ISO/IEC, 2018; Ritchie, 1993), vaan ohjelma 2.1 näyttää samalta kuin vuonna 1988 julkaistussa *The C Programming Language* -kirjassa (Kernighan & Ritchie, 1988). Ohjelma tulostaa näytölle merkkijonon *Hello, World*.

C kehitettiin B- ja BCPL-ohjelmointikielten pohjalta näiden osoittautuessa kömpelöiksi vaihtaessa laitteistoarkkitehtuuria, kuten luvussa 2.2 kerrotaan. Uudella ohjelmointikielellä tavoiteltiin B:tä nopeampaa, mutta BCPL:ää yksinkertaisempaa kieltä Unix-käyttöjärjestelmän kehitykseen (Ritchie, 1993).

Toisin kuin monissa nykypäivänä suosituissa kielissä, C ei tarjoa automaattista muistinhallintaa, vaan ohjelmoijan täytyy itse hallita sekä muistin varaaminen et-

tä sen vapauttaminen. Tämä on pitänyt kielen toteutuksen hyvin tehokkaana ja yksinkertaisena, mikä on mahdollistanut C:n leviämisen järjestelmästä toiseen. Kääntöpuolena muistinhallinta jätetään ohjelmoijan vastuulle, monimutkaistaen ohjelmien toteutusta.

C:n mahdollistama suorituskyky on johtanut kielen suosioon tehokkuutta vaativissa sovelluksissa, kuten verkkopalvelimissa ja tietokannoissa. C:n yksinkertainen lähestymistapa muistinkäsittelyyn on taas mahdollistanut tarkkaa muistinhallintaa vaativien ohjelmistojen toteutuksen, kuten käyttöjärjestelmien ytimien tai sulautettujen järjestelmien ohjelmien. Tämä yhdistelmä on johtanut kielen laajaan käyttöön käyttötarkoituksesta riippumatta.

Koska C:llä on helppoa saada aikaan erilaisia muistinkäsittelyyn liittyviä tietoturvaongelmia, lukuisia työkaluja on kehitetty havaitsemaan ja estämään näitä (mm. Nethercote & Seward, 2007; Serebryany, Bruening, Potapenko, & Vyukov, 2012). Myös lukuisia ohjelmointikieliä on luotu toteuttamaan "turvallinen C", usein lisäämällä jokaisen osoittimen käytön yhteyteen tarkistuksen, osoittaako osoitin ohjelman omistamaan muistiin.

Koska C on suunniteltu mahdollisimman yksinkertaiseksi kieleksi, siitä ei löydy omaa moduulijärjestelmää. Mikäli ohjelmoija haluaa käyttää jonkun toisen tiedostojen sisältämää funktiota, ohjelmaan pitää sisällyttää käytettyjen funktioiden määrittelyt eli funktioprototyypit. Yleensä nämä löytyvät erityisistä otsikkotiedostoista, joiden sisältö kopioidaan ohjelmaan käyttäen C:n makrojärjestelmää.

C on vaikuttanut lukuisien kielten kehitykseen ja useiden kielien sanotaankin olevan osa C-kieliperhettä. C-kieliperheeseen kuuluviin kieliin liittyy usein muun muassa imperatiivinen ohjelmointityyli, perinteinen merkintätapa (eli infix) lausekkeiden muodostamiseen, muuttujien näkyvyysalueiden rajoittaminen kaarisuluilla sekä staattinen tyypitys. Näitä piirteitä näkyy ohjelmassa 2.1: kaarisuluilla rajoitettu `main`-funktio kutsuu `printf`-funktiota, joka tulostaa käyttäjälle merkkijonon *Hello, World*. Funktio ottaa parametriksi kokonaisluvun `argc` ja osoittimen merkkijonolistaan `argv`.

Suurin osa nykypäivänä käytetyimmistä ohjelmointikielistä on osa C-kieliperhettä, kuten TIOBEn indeksissä (2020) kärkiviisikosta löytyvät C:n lisäksi Java, C++ ja C#. Python on ainoa kärkiviisikossa oleva ohjelmointikieli, joka ei suoraan ole osa C-kieliperhettä, mutta Pythonin referenssi-implementaatio CPython on nimensä mukaisesti toteutettu C:llä ja Pythonilla (Python Software Foundation, 2020).

2.2 C-ohjelmointikielen historiaa ja nykypäivää

Dennis Ritchie (1993) käsittelee C-kielen historiaa ja tekemiään suunnittelupäätöksiä artikkelissaan *The Development of the C Language*. Artikkelin mukaan C kehitettiin pitkälti 70-luvulla B- ja BCPL-ohjelmointikielien pohjalta. Ritchie tavoitteli uudella ohjelmointikielellä B-ohjelmointikielen tehokkuuden parantamista tarkemmalla tyypityksellä. B-kielen ainoa primitiivityyppi oli sana (engl. *word*, B-kielen tyyppinä `cell`), sillä B oli suunniteltu ajettavaksi tietokoneilla, joissa muistiosoittimet osoittivat aina yksittäisiin sanoihin. Tämä kuitenkin käytännössä osoittautui haastavaksi esimerkiksi merkkijonojen käsittelyyn, sillä jokaiseen sanaan mahtuu useita tavun kokoisia merkkejä. Tämä tarkoittaa ohjelmoidessa sitä, että ohjelmoija joutuu purkamaan käsin yksittäisen sanan merkeiksi, käsittelemään näitä yksittäin ja lopuksi yhdistämään merkit uudelleen sanoiksi. Epäkäytännöllisyyden lisäksi tämä oli tehotonta, kun B-kieli muutettiin tavupohjaiselle PDP-11 -tietokoneelle säilyttäen kuitenkin sanapohjaiset muistiosoittimet – ohjelmoijan oli pakko käyttää sanarajoilla toimivia muistiosoittimia, vaikka tavupohjainen muistiosoitin olisi tehokkaampi ratkaisu.

B-kielen taulukot poikkesivat huomattavasti nykyisistä C:n taulukoista. Jos B-koodissa luodaan kymmenen alkion taulukko `A`, niin ohjelmaa ajettaessa varattaisiin kymmenen sanan kokoinen taulukko ja `A`:han asetettaisiin tämän taulukon osoitin. C:ssä `A` muutettaisiin osoittimeksi määrittelyhetken sijaan vasta, kun muuttujaa käytettäisiin lausekkeessa.

Tämän lisäksi Ritchien mukaan B-ohjelmointikielestä puuttui kokonaan liukulukujen käsittely. Vaikka PDP-11 ei tukenut liukuluvuilla laskentaa, valmistaja oli luvannut tuen tälle. BCPL-ohjelmointikieleen lisättiin liukulukulaskenta olettaen, että liukuluku mahtuisi yhteen sanaan, mikä ei pitänyt paikkaansa 16-bittisellä PDP-11 -tietokoneella.

Artikkelin mukaan C-ohjelmointikielen kehitys alkoi vahvemmalla tyypityksellä: C-kielen ensimmäiseen varsinaiseen esiasteeseen oli lisätty `char`- ja `int`-tyypit sekä muistiosoittimet näihin tyyppeihin. Tässä vaiheessa C-tyylisten taulukoiden sijaan taulukot toimivat kuin B-kielen taulukot. Kun C:hen lisättiin tietuetyypit, tämä muistimalli ei toiminut enää, vaan taulukoiden käsittelyä muutettiin vastaamaan nykyistä C:n mallia. Nyt taulukot muunnettiin muistiosoittimiksi vasta, kun taulukkoa käytettiin lausekkeissa, ja taulukon määrittelyssä alkioiden ja osoittimen sijaan varattiin muistia vain taulukon alkioille.

Tästä tavasta käsitellä taulukoita seurasi kuitenkin ominaisuus, joka on myös nykypäi-

vän C:ssä: jos funktio ottaa parametrikseen taulukon, kääntäjä muuttaa parametrin tyypin muistiosoittimeksi, sillä funktiokutsut ovat lausekkeita[1].

Tämän lisäksi C-kieleen lisättiin tyyppi funktio-osoittimille. Määrittelysyntaksin logiikan perusteena toimi lausekkeiden syntaksi. Jos jostain muuttujasta saa `int`-arvon, kun lausekkeessa lukee `(*muuttuja)()`, niin `muuttuja` määritellään kirjoittamalla `int (*muuttuja)();`. Monimutkaisissa tapauksissa on kuitenkin hankala erottaa tyyppejä toisistaan, kuten erot tyyppien ”osoitin taulukkoon kokonaislukuja” eli `int (*muuttuja)[]` ja ”taulukko osoittimia kokonaislukuihin” eli `int *muuttuja[]`.

B-yhteensopivuuden tavoittelu ohjasi kielen suunnittelua syntaksin osalta mahdollisimman B:n kaltaiseksi. B-lause `if(a & b)` vastaa C-lausetta `if(a && b)`, mutta B-kielessä `&`:tä käytettiin myös bittioperaatioihin loogisten operaatioiden lisäksi. Koska B-ohjelmien haluttiin toimivan C-ohjelmien tavoin mahdollisimman pienillä muutoksilla, C-idiomissa `(a&mask) == b` lausekkeessa käytetyn `&`-bittioperaattorin ympärille joutuu lisäämään sulut. Tämä johtuu B-kielen `if(a == b & c)` -tyylisistä lausekkeista, joiden haluttiin toimivan ilman muutoksia C:ssä kielen vaihtamisen helpottamiseksi.

Seuraavaksi alkoi C-kielen esikäsittelijän kehitys. Aluksi esikäsittelijässä oli vain toiminnot tiedostojen sisällyttämiseen (`#include`) ja yksinkertaiseen korvaamiseen (`#define`), mutta hyvin nopeasti kieleen lisättiin funktiomakrot sekä `#if`-lauseet. Aluksi esikäsittelijää pidettiin vain vapaaehtoisena laajennoksena C:hen, mikä selittää myös nykypäivänä esikäsittelijän huomattavat erot muuhun C-kieleen verrattuna. Ensimmäisen C-standardin jälkeen esikäsittelijä on pysynyt lähes koskemattomana. Ainoa lisätty ominaisuus oli C99-standardin mukana tulleet funktiomakrot, jotka voivat ottaa mielivaltaisen määrän argumentteja.

Myöhemmin, kun C oli levinnyt usealle eri alustalle, alkoi olla selkeää, että C tarvitsi standardin. Brian Kernighan, jonka kanssa Ritchie oli kirjoittanut *The C Programming Language* -kirjan (Kernighan & Ritchie, 1978), kirjoitti Ritchien kanssa C:n ensimmäisen standardin ANSIn X3J11 -työryhmässä. Kuuden vuoden jälkeen työryhmä sai valmiiksi nk. C89 -standardin, joka tunnetaan myös ANSI C:nä[2] (ISO/IEC, 1990). Samoihin aikoihin valmistui myös toinen painos *The C*

[1] Jos funktio ottaa esimerkiksi `char[2]` -tyyppisen parametrin, funktio saakin parametrikseen muistiosoittimen taulukon sijaan. Tässä tapauksessa funktio saa siis moderneilla tietokoneilla kahden tavun sijaan kahdeksan tavun kokoisen parametrin. Tämän ominaisuuden voi kiertää säilömällä taulukon tietueen sisään.

[2] ISO-järjestö hyväksyi standardin pienillä muutoksilla vuonna 1990, jonka vuoksi standardi tunnetaan myös C90-standardina.

Programming Language -kirjasta, jossa korjattiin lukuisia eroja ensimmäisen version ja C-standardin välillä (Kernighan & Ritchie, 1988).

Nykypäivänä C:tä käytetään käytännössä jokaisessa tietokoneessa käyttöjärjestelmästä riippuen joko pelkästään ydinkomponenttien toteutukseen tai koko käyttöjärjestelmän toteutukseen. Sulautetuissa järjestelmissä C on yksi suosituimmista kielistä johtuen kielen yksinkertaisuudesta ja suorituskyvystä. C:n suosion myötä myös kielen huonot puolet nousevat esiin erilaisissa tietoturvaongelmissa, jotka johtuvat kielen sallimasta rajoittamattomasta muistinkäsittelystä yhdistettynä ohjelmoijan tekemiin virheisiin. Esimerkiksi puskuriylivuodoissa C:llä kirjoitettu ohjelma tallentaa tietoa muualle tai lukee muistia muualta kuin mitä ohjelmoija on tarkoittanut, kun ohjelmoija jättää tekemättä kriittisen syötteen oikeellisuustarkistuksen. C-kääntäjä voi optimoidessa poistaa tällaisia tarkistuksia ja aiheuttaa tietoturvaongelmia, jos kääntäjä päättelee tarkistusten olevan ”turhia” (Red Hat, Inc., 2019).

C on hyvin yksinkertainen kieli ja se on selviytynyt nykypäivään asti lähes identtisenä ANSI C:hen. Uudemmat standardit (ISO/IEC, 1999, 2011, 2018) ovat lähinnä tehneet pieniä parannuksia kielen tehokkuuteen esimerkiksi lisäämällä `restrict`-avainsanan. Erilaiset kääntäjät ovat kuitenkin tuottaneet omia laajennoksiaan kieleen mahdollistaen tehokkaampien mutta kääntäjäriippuvaisten C-ohjelmien kirjoituksen. Moderni esimerkki kääntäjäriippuvaisesta syntaksia muokkaavasta laajennoksista on vektorityypit, joita esimerkiksi GCC-kääntäjän ymmärtää laajennetun C-standardinsa `__attribute__(())` -syntaksilla. C:stä löytyy myös standardien mukainen tapa käyttää kääntäjäriippuvaisia ominaisuuksia, `#pragma`. Pragmoja käytetään erityisesti OpenMP-kirjaston (OpenMP, 2018) yhteydessä.

Nykypäivänä käytännössä jokainen alusta tukee C:tä. C:tä käytetään alustoilla muun muassa ohjelmointikielten väliseen kommunikaatioon – jos C#-ohjelma haluaa käyttää Java-ohjelman kirjastorutiineja, C#-ohjelman on helpointa käyttää Java-ohjelman C-rajapintaa.

C on nykyään käytössä erityisesti matalan tason ohjelmoinnissa, kuten käyttöjärjestelmien ytimissä, sulautetuissa järjestelmissä, Unix-työkaluissa, vapaan lähdekoodin ohjelmistoissa, tietokannoissa ja muissa tehokkuutta vaativissa ohjelmistoissa.

2.3 Tärkeimmät C-ohjelmointikielen ominaisuudet

Artikkelissa *The Next 7000 Programming Languages* (Chatley ja muut, 2019) käsitellään ohjelmointikielten kehitystä ja pohditaan mahdollisia ominaisuuksia tulevissa

ohjelmointikielissä, joita nykyiset ohjelmointikielet eivät sisällä. Artikkelissa selitetään C:n nykyistä suosiota kielen yksinkertaisuudella ja tehokkuudella. Nämä ominaisuudet ovat mahdollistaneet C:n laajan käytön käyttöjärjestelmistä työkaluihin huolimatta C:n turvattomuudesta.

Artikkelin mukaan C:n (ja C++:n) korvaaminen lyhyellä tähtäimellä on mahdotonta johtuen kielen suosiosta. Koska lukuisat työkalut kääntäjistä virheenjäljittäjiin (engl. *debugger*) on kirjoitettu yksinomaan C:tä varten, vastaavien työkalujen luominen muita ohjelmointikieliä varten on huomattava investointi. C:n yleisyydestä myös seuraa suuri määrä ohjelmoijia, kirjastoja ja työkaluja, mikä tekee C:stä luonnollisen valinnan myös uusiin projekteihin. Artikkelissa kuitenkin todetaan, että useat ohjelmointikielet ovat vähentäneet C:n suosiota tarjoten yksittäisillä osa-alueilla parannuksia. Yksi mainituista ohjelmointikielistä on Rust, joka mahdollistaa paremman ohjelmien käännösaikaisen oikeellisuustarkistamisen heikentämättä tehokkuutta C:hen verrattuna. Toiseksi vaihtoehdoksi tarjotaan suoritusaikaisia tarkistuksia turvallisuuden parantamiseksi.

Artikkelissa puhutaan myös mahdollisuudesta oikeellisuuden tasaiseen parantamiseen (engl. *gradual verification*), joka mahdollistaisi ohjelmien ensimmäisten versioiden toteuttamisen ilman kattavaa käännösaikaista oikeellisuustarkistusta, mutta antaen kehityksen jatkuessa työkalut ohjelmiston oikeellisuuden varmistamiseen. Esimerkiksi TypeScript (Microsoft Corporation, 2020)-ohjelmointikieli mahdollistaa tyypittämättömien JavaScript-ohjelmien vaillinaisen tyypityksen (engl. *gradual typing*), jolloin ohjelmien oikeellisuutta voi käännösaikaisesti tarkistaa funktio kerrallaan.

Artikkelissa *Some Were Meant For C* Kell (2017) myös nostaa esiin tarpeen yksittäisten funktioiden kerrallaan muuntamisesta. Useissa kielissä ei ole saumatonta C-yhteensopivuutta, jolloin kielestä toiseen siirtyminen vaatii joko koko ohjelman uudelleenkirjoituksen tai erillisen yhteensopivuuskerroksen alkuperäisen ja uuden kielen väliin. Koska lukuisat työkalut ja kirjastot on toteutettu C:tä varten, artikkelin mukaan C tulee pysymään jatkossakin tärkeänä osana tietokoneita.

Yhteensopivuuden sijaan Kell kuitenkin painottaa C:n alusta-agnostisuutta kielen tärkeimpänä ominaisuutena. Useilla alustoilla voi normaalin välimuistin lisäksi käyttää laitteistoa tai tiedostojärjestelmää suoraan muistiosoitteina, minkä C:n yksinkertainen lähestymistapa muistinhallintaan mahdollistaa. Artikkelissa esitetään tästä esimerkkinä iteraation ohjelman omien konekielisen käskyjen yli, mikä muissa kielissä olisi kohtuuttoman hankalaa, mutta C:ssä triviaalia.

Molemmissa artikkeleissa käsitellään C:n määrittelemätöntä toimintaa kielen se-

kä hyvänä että huonona puolena. Lukuisat tietoturvaongelmat ovat johtuneet C:n turvattomuudesta, mutta toisaalta kielen turvattomuutta voi käyttää hyväksi mahdollisimman tehokkaiden ohjelmien toteutuksessa.

2.4 Kehitettävissä olevat ominaisuudet C-ohjelmointikielessä

Ritchie (1993) nostaa esiin kaksi usein keskustelua herättänyttä C:n ominaisuutta. Toinen näistä on C:n tyyppisyntaksi ja toinen on C:n tapa käsitellä taulukoita ja osoittimia keskenään. Ensimmäiselle näistä voi tehdä verrattain helposti jotain, sillä syntaksin muuntaminen käännösaikaisesti on triviaalia. Taulukoiden ja osoittimien välistä käytöstä on paljon hankalampi muuntaa, sillä nykyiset C-ohjelmat käyttävät osoittimia ja taulukoita sekaisin. Yksi C:stä puuttuva ominaisuus on taulukon antaminen funktion parametrina, jonka voi kuitenkin tehdä käärimällä taulukko tietueen sisään.

Muita syntaktisia parannuksia lausekkeisiin voi tehdä tietueiden kohdalla. Jos lausekkeessa käyttää tietuetta, niin tietueen `foo` jäsenen `bar` saa lausekkeella `foo.bar`, mutta jos `foo` onkin osoitin, lausekkeen tulee olla `foo->bar`. Jos `foo` olisi osoitin osoittimeen, lauseke olisi `(*foo)->bar`. Yksinkertaisempi syntaksi olisi käyttää jokaisessa tapauksessa lauseketta `foo.bar` ja jättää tarvittava muoto kääntäjän pääteltäväksi. Esimerkiksi Rust-ohjelmointikielen syntaksi tietueiden käsittelyyn on tällainen.

C:n `static`-avainsana on jaettu käytön mukaisesti kahteen avainsanaan. Funktiomäärittelyissä ja globaaleissa muuttujissa C:n `static`-avainsana vastaa muiden kielten avainsanaa yksityiselle funktiolle. Globaaleja `static`-määreellä määritettyjä funktioita ja muuttujia ei voi käyttää muusta kuin samasta tiedostosta, jossa kyseinen symboli on määritelty.

Toinen `static`-määreen käyttö on funktioiden sisällä muuttujien määrittelyyn. Staattiset muuttujat alustetaan vain kerran, vaikka funktiota kutsuttaisiin useita kertoja. Ohjelmassa 2.2 käytetään tällaista muuttujaa. Ohjelma tulostaa ensin numeron 0, jonka jälkeen ohjelma tulostaa numeron 1.

C:n tyypitys sallii monia ilmaisuja, jotka voivat johtaa salakavaliin ongelmiin suoritusaikaisesti. Koska C sallii suurempien kokonaislukutyyppien asettamisen pienempiin kokonaislukutyyppeihin ilman ohjelmoijan tekemää muunnosta, nämä arvot voivat suoritusaikaisesti muuttua ilman käännösaikaisia varoituksia. Mainittavasti lauseet `int a = 256; unsigned char b = a;` eivät aiheuta käännösaikaisesti edes varoituksia GCC-kääntäjällä, vaikka ”kaikki” kääntäjän varoitukset olisivat päällä `-Wall`

```
int foo() {
    static int i = 0;
    return i++;
}

void main() {
    printf("%d\n", foo());
    printf("%d\n", foo());
}
```

Ohjelma 2.2: Staattinen muuttuja C:ssä

-komentolipun avulla. Kääntäjälle pitää antaa erillinen `-Wconversion` -komentolippu, jotta kääntäjä edes varoittaisi mahdollisesti yllättävästä käytöksestä. Tämän ominaisuuden muuttaminen virheeksi tai edes varoitukseksi ei ole ongelmatonta, sillä esimerkiksi C-makrojen tuottama koodi voi olettaa tällaisten lausekkeiden toimivan.

C:hen voisi myös lisätä useita uusia tyyppejä, kuten tyypin ei-tyhjälle osoittimelle. Käännösaikaisesti tarkistettuja ei-tyhjiä osoittimia voi käyttää turvallisempien ja nopeampien ohjelmien kirjoittamiseen. GCC- ja Clang-kääntäjät tukevat ei-tyhjiä osoittimia `__attribute__((nonnull))` -määreellä. Kääntäjä voi käyttää märettä optimoidessa funktioita – erityisesti turhat tarkistukset tyhjien muuttujien varalta optimoidaan pois.

Muita hyödyllisiä tyyppejä ohjelmoinnin helpottamiseen olisivat monikot (engl. *tuple*) ja summatyypit (engl. *tagged union, sum type*). Nämä molemmat tyypit löytyvät esimerkiksi Rustista. Ensimmäisen saa käännettyä triviaalisti tietueeksi ja toisen saa käännettyä tietueeksi, jossa on sisällä vaihtoehdoista luetelman (engl. *enumeration*) ja yhdisteen (engl. *union*) yhdistelmä. Ohjelmassa 2.3 on Rustin ja C:n syntaksien mukaiset summatyypit tyypille, jossa on joko operaation onnistuessa jokin kokonaisluku tai epäonnistumisen yhteydessä epäonnistumisen syy merkkijonona. Erillisen summatyypin määritteleminen mahdollistaa yhtä aikaa sekä paremman oikeellisuustarkistuksen että tehokkaampien ohjelmien tuottamisen. C-esimerkissä `failure`-kentässä voisi olla arvo, vaikka `type`-kentän arvo olisi `_SUCCESS`. Tämä voi johtaa määrittelemättömään toimintaan, jos `success`-kentän arvoa yritetään käyttää ja sillä hetkellä alustettu kenttä on `failure`.

C:ssä ei ole erillistä moduulijärjestelmää, jonka lisääminen voisi nopeuttaa kään-

```
enum SuccessOrFailure {
    Success(i32),
    Failure(String)
}
```

```
struct SuccessOrFailure {
    union {
        int32_t success;
        char *failure;
    } value;
    enum {
        _SUCCESS,
        _FAILURE
    } type;
};
```

Ohjelma 2.3: Summatyyppi Rustissa ja C:ssä.

tämistä ja tehdä ohjelmien ymmärtämisestä yksinkertaisempaa. C-ohjelmat voivat käyttää kirjastojen funktioita kirjoittamalla kirjastofunktioista prototyypit, jotka yleisesti ottaen ovat kirjastojen otsikkotiedostoissa. Koska otsikkotiedostojen sisältö käytännössä ottaen kopioidaan `#include`-kutsun tilalle, kääntäjä joutuu käsittelemään samoja otsikkotiedostoja lukuisia kertoja käännösprosessin aikana[3]. Erillisen moduulijärjestelmän lisääminen voisi kääntämisen nopeuttamisen lisäksi yksinkertaistaa kirjastojen toteuttamista, sillä ohjelmoijien ei tarvitsisi kirjoittaa kirjastoilleen otsikkotiedostoja.

C:n makrojärjestelmässä on paljon parantamisen varaa. C:n esikäsittelijä toimii hyvin yksinkertaisissa tapauksissa, mutta sen rajoitteet tulevat nopeasti esille monimutkaisempia makroja kirjoittaessa. Voimakkaampi makrojärjestelmä mahdollistaa lyhyempien ja selkeämpien makrojen kirjoittamisen monimutkaisemmissa tapauksissa. Heathcote (2014) esittelee artikkelissaan tapoja hyväksikäyttää C:n esikäsittelijää esimerkiksi iteraation toteuttamiseen. Moderneissa makroprosessoreissa muun muassa iteraation toteuttaminen makroilla on suoraviivaisempaa.

[3]Käytännössä kääntäjät pystyvät optimoimaan tietyllä yleisellä tavalla kirjoitettuja otsikkotiedostoja ja jättämään jo kertaalleen luetut tiedostot kokonaan pois.

3 Ohjelmointikielten vertailun määritelmät

Ohjelmointikielten vertailemiseen on useita erilaisia tapoja, kuten kielten suosio, ominaisuudet, tehokkuus ja työkalujen määrä. Jotta tässä tutkielmassa voitaisiin verrata ohjelmointikieliä analyyttisesti, tässä luvussa määritellään vertailukriteerit ohjelmointikielille, valitaan C:hen verrattavissa olevat ohjelmointikielet sekä käsitellään muita mahdollisia syitä ohjelmointikielen valintaan.

3.1 Ohjelmointikielten vertailun kriteerit

Verrattavissa ohjelmointikielissä on pyritty parantamaan C:n huonoja puolia hyvien puolien kustannuksella, usein lisäämällä kieleen turvallisuutta parantavia ominaisuuksia tai tehden kielestä helppokäyttöisemmän esimerkiksi automaattisella muistinhallinnalla. Tämä kuitenkin heikentää kielen tehokkuutta tai alustariippumattomuutta, mikä hankaloittaa kielten suoraa vertailua. Määrittelemällä absoluuttiset reunaehdot voidaan vertailla kieliä ehtojen puitteissa objektiivisesti. Jos yksikin näistä kriteereistä ei pidä, verrattava kieli ei ole aidosti C:tä parempi, vaan se häviää C:lle joissain osa-alueissa ja vastaavasti voi olla parempi joissakin toisissa.

Tutkielmassa vertaillaan kolmea osa-aluetta ohjelmointikielistä: suorituskykyä, muistinkäyttöä sekä yhteensopivuutta C:n ja muiden ohjelmointikielten kanssa, sillä nämä ominaisuudet nousevat esiin Kellin (2017) ja Chatleyn, Donaldsonin ja Mycroftin (2019) tutkimuksissa. Tutkielmassa käsitellään myös subjektiivisempia kielten ominaisuuksia, kuten kielen tiiviyttä, oikeellisuutta ja ylläpidettävyyttä, mutta näitä ominaisuuksia ei huomioida kielten paremmuusvertailussa. Nämä ovat kuitenkin olleet tärkeitä kriteerejä ohjelmointikielten valitsemisessa (Meyerovich & Rabkin, 2013), joten aiheiden käsitteleminen selittää, miksi näennäisesti tehottomammat ohjelmointikielet ovat suosittuja.

Ohjelmointikielellä kirjoitetun ohjelman tulee olla suoritusaikaisesti vähintään yhtä nopea kuin vastaava C:llä kirjoitettu ohjelma. Kieli siis ei saa vaatia ohjelmoijaa käyttämään mitään kielen ominaisuuksia, jotka voisivat hidastaa ohjelmien suoritusta C:hen verrattuna. Monet suoritusaikaiset turvallisuutta lisäävät ominaisuudet, kuten muistialueiden tarkistukset, hidastavat kielen suoritusaikaista nopeutta.

Ohjelmointikielellä toteutettu ohjelma ei myöskään saa käyttää enempää muistia niin suoritusaikaisesti kuin talletusvälineelläkään verrattuna vastaavaan C-ohjelmaan. Tämä koskee myös vakiokirjastoa (engl. *standard library*) – yksikin konkreettinen

vakiokirjaston funktio linkitettynä ohjelmaan kasvattaa ohjelman kokoa. Mikäli jokin vakiokirjasto toteutetaan, sen käyttäminen tulee olla ohjelmoijalle täysin vapaaehtoista. Yksi tapa toteuttaa tämä on liittää vain käytetyt funktiot osaksi ohjelmaa, jolloin käyttämättömät funktiot eivät kasvata ohjelman kokoa.

Jos samaa vakiokirjastoa käytetään useassa ohjelmassa, tilankäytön kannalta on edullisempaa säilöä vakiokirjasto jaettuna kirjastona, mutta tämä heikentää kääntäjän mahdollisuuksia käännösaikaiseen optimointiin. Jos vakiokirjastoa ei ole ladattu muistiin ohjelman käynnistyessä, ohjelman käynnistys voi kestää hieman kauemmin. C:n vakiokirjasto liitetään usein moderneissa käyttöjärjestelmissä ohjelmiin jaettuna kirjastona, sillä C:tä käytetään lähes jokaisessa käyttöjärjestelmän ohjelmassa. Näin jaetun kirjaston käyttäminen välttää osan jaetun kirjaston huonoista puolista: kirjasto löytyy vain yhtenä kopiona kiintolevyltä ja se on valmiina ladattuna välimuistiin.

Ohjelmointikielen tulee olla täysin yhteensopiva C:n suoritusympäristön kanssa. Tämä koskee C-koodin kutsumista C:n vierasfunktiorajapinnan läpi (engl. *Foreign function interface, FFI*) sekä kielen funktioiden kutsumista muiden ohjelmointikielten C-rajapinnan läpi. Kielen tulee näiden lisäksi toimia kaikissa ympäristöissä, joissa C toimii. Kielen pitää myös tukea C:n esikäsittelijää, jotta C:n käyttäminen verrattavan ohjelmointikielen kanssa olisi mahdollisimman saumatonta.

3.2 C:hen verrattavissa olevat ohjelmointikielet

Historian saatossa on tehty useita C:n kilpailijoita, jotka ovat yrittäneet parantaa C:tä joidenkin C:n hyvien puolien kustannuksella. Muutamat näistä ovat päätyneet hyvin suosituiksi ohjelmointikieliksi, kuten esimerkiksi C++ ja Go. Kielten suosion mittaamiseen on tehty useita projekteja, jotka vertailevat kieliä esimerkiksi hakutulosten tai projektien mukaan. Näitä ovat esimerkiksi TIOBEn ohjelmointikielten suosion indeksi (TIOBE software BV, 2020) ja GitHub-palvelun julkaisema Octoverse (GitHub, Inc., 2019).

Koska tutkimuskysymyksessä vertaillaan ohjelmointikieliä suorituskyvyn ja muistinkäytön suhteen, vertailuun kannattaa ottaa mukaan vain tehokkaita kieliä – korkeamman tason ohjelmointikielet on tarkoitettu ohjelmointinopeuden parantamiseen ja turvallisempien ohjelmistojen toteuttamiseen nopeiden ohjelmien sijaan. Tällöin kielen suorituskyky on heikompi. Yksi kattava suorituskykyä mittaava vertailu on Benchmarks Game (Gouy, 2020a), jossa pyritään kirjoittamaan mahdollisimman

nopea ohjelma pysyen silti kielelle idiomaattisessa lähdekoodissa[4].

TIOBEn listasta Ada, C++, D, Go ja Rust nousevat esiin verrattavina kielinä. Kaikki viisi kieltä ovat tehokkaita. Tämän lisäksi jokaisen kielten historiassa on ollut tavoitteena korvata C:n tai C++:n käyttö, kuten luvussa 4 kerrotaan.

Viime vuosina on tehty myös useita C:hen käännettäviä ohjelmointikieliä, jotka ovat jääneet pitkälti ilman mitään näkyvyyttä, kuten LISP/c (Baca, 2016), C-Mera (kiselgra, 2019), Carp (Svedäng & Heller, 2020) ja Nymph (Barber, 2020). LISP/c, C-Mera ja Carp ovat LISP-perheeseen kuuluvia C:ksi kääntyviä ohjelmointikieliä, jotka pyrkivät parantamaan C:n syntaksia korvaamalla sen LISP-perheen syntaksilla. Nymph taas on olio-ohjelmointikieli. Erityisesti Carp on tämän tutkielman kannalta kiintoisa ohjelmointikieli, sillä se on C:ksi kääntyvä ohjelmointikieli, joka on suunniteltu mahdollisimman suorituskykyiseksi.

3.3 Kielten suosioon vaikuttavat tekijät

Eräässä tutkimuksessa (Meyerovich & Rabkin, 2013) tutkittiin syitä ohjelmointikielten valintaan. Yhden tutkimuksen järjestämän kyselyn (s. 8, Slashdotissa julkaistu kysely, n=1679) perusteella kielen valintaan vaikuttaa avoimen lähdekoodin kirjastojen saatavuus, olemassa olevien ohjelmien jatkokehitys sekä kielen tunnettavuus ohjelmoijien keskuudessa – tutkimuksen mukaan ohjelmoijat siis suosivat jo käytettyjä ohjelmointikieliä uusien kielten sijaan. Saman kyselyn vastaajat arvioivat suorituskyvyn turvallisuutta tärkeämmäksi.

Saman tutkimuksen järjestämässä Slashdot-sivustolla julkaistussa kyselyssä noin 40% vastaajista arvioi tärkeäksi kriteeriksi työkalut. Kyselyn perusteella kielen olisi siis hyvä tarjota toimivat työkalut, kuten ohjelmistopakettien (engl. *software package*) hakemiseen paketinhallintajärjestelmän (engl. *package manager*), nopean ja käyttäjäystävällisen käännöstyökalun sekä valmiudet olemassa oleviin kehitysympäristöihin integroitumiselle. Olemassa olevien C-ohjelmistojen tukeminen on välttämätöntä mutta haastavaa johtuen C:n ekosysteemin monimuotoisuudesta, erityisesti lukuisista kääntämistyökaluista.

Tutkimuksessa selvitettiin myös suosittuja ominaisuuksia ohjelmointikieliltä (s. 13, SaaS MOOC -kurssin yhteydessä oleva kysely, n=415). Useita tutkimuksessa selvitetyistä suosituimmista ominaisuuksista ei ole mahdollista toteuttaa johtuen luvussa 3.1

[4]Hyvin monessa kielessä voi kirjoittaa C:hen verrattavissa olevaa matalan tason ohjelmointia, mutta Benchmarks Gamessa on tarkoituksena välttää tätä.

määritetyistä rajoitteista, kuten poikkeuksia ja rajapintoja. Useat muut tutkimuksessa esiin nousseet ominaisuudet, kuten suorituskyky, ovat taas suoraan rajoitteiden mukaisesti osa verrattavan ohjelmointikielen tavoitteita.

Tutkimuksessa myös verrattiin tiettyjen toteamuksien, kuten *"This language has a strong static type system"* keskinäistä korrelaatiota. Kielen tiiviys ("*This language is expressive*") korreloi eniten (korrelaatiokertoimella 0.76) kielestä pitämisen kanssa (s. 13, The Hammer Principle -sivustolla julkaistu kysely).

Tutkimuksen perusteella valmiiksi suosittuja kieliä käytetään enemmän myös uusissa projekteissa. Olemassa olevien kirjastojen tärkeyttä korostetaan useassa kohdassa tutkimusta. Täysin C:n kanssa yhteensopiva kieli voi käyttää C:lle tehtyjä kirjastoja, jolloin kielellä on käytettävissään laaja C:n ekosysteemi[5].

Uusia ohjelmointikieliä opetellessa ohjelmoijat turvautuvat aikaisemmista kielistä opittuihin käytäntöihin (Scholtz & Wiedenbeck, 1990). Uusien ohjelmointikielten käyttöönottoa helpottaa siis muiden vastaavien kielten osaaminen, sillä aikaisempi kokemus tukee uuden kielen opiskelua. Suunnittelemalla C:n korvaajan C:n kanssa samankaltaiseksi kieleksi voidaan pienentää uuden kielen opettelun kynnystä. Kielen eriävät ominaisuudet olisi siis hyvä toteuttaa siten, että ne ovat mahdollisimman helppoja oppia C:stä uuteen ohjelmointikieleen siirtyvälle ohjelmoijalle.

[5] Esimerkiksi GitHubista hakusanalla 'library' löytyy yli 26 000 C:llä kirjoitettua projektia.

4 Ohjelmointikielten vertailu

Tässä luvussa käydään läpi C:hen verrattavissa olevat ohjelmointikielet ja pohditaan mahdollisia syitä, miksi juuri C on laajassa käytössä muiden kielten sijaan. Kieliä käsitellään sekä analyyttisellä pohdinnalla tutkien kielen ominaisuuksia että Benchmarks Gamen suorituskykymittauksilla.

4.1 Yleisiä vertailtavien ohjelmointikielten ominaisuuksia

C:hen vertailtavissa ohjelmointikielissä on yleisesti useita ominaisuuksia, jotka hidastavat ohjelmien suoritusaikaista nopeutta, lisäävät muistinkäyttöä, vähentävät alustariippumattomuutta tai heikentävät yhteensopivuutta C:n kanssa.

Yleisin näistä on automaattinen muistinhallinta, joka muistin vapauttamisen automatisoimiseksi seuraa ohjelman käyttämää muistia. Lähes aina automaattinen muistinhallinta lisää kieleen ”roskien keräämisen” (engl. *garbage collection, GC*), jonka ajaksi ohjelman suoritus pysäytetään. Lisäksi roskien keräämiseen perustuva automaattinen muistinhallinta lisää muistinkäyttöä, sillä ohjelmointikieli joutuu suoritusaikaisesti seuraamaan käytössä olevia muistiosoitteita.

Monissa vertailtavissa kielissä on käytössä nimiruntelu (engl. *name mangling*), joka mahdollistaa useat näennäisesti samannimiset funktiot. Tämä kuitenkin aiheuttaa ohjelmointikielten välille yhteensopivuusongelmia, sillä toisesta kielestä kutsuttaessa pitää tietää kutsuttavan funktion todellinen nimi. Esimerkiksi `int`-tyyppisen arvon palauttava funktion `foo()` oikeaksi nimeksi voisi tulla `_Z3foov`, kuten G++-kääntäjä (Free Software Foundation, Inc, 2020) tekee.

Ohjelmointikielen ominaisuudet vaikuttavat siihen, minkälaisia ohjelmistoarkkitehtuureja kielellä tehdään (Gil & Lorenz, 1998). Moderneissa ohjelmointikielissä virheiden käsittely on yleensä toteutettu kahdella tavalla: toinen on poikkeavat paluuarvot ja toinen on poikkeuksien heittäminen. Yleisesti ottaen kaikki ohjelmointikielet tukevat ensimmäistä ja suurin osa toista tapaa. Poikkeusten käsittely on hieman hitaampaa ja aiheuttaa hieman suuremman muistinkäytön ja tehokkaaseen ohjelmakoodiin pyrkiessä yleensä vältetään poikkeusten käyttämistä (de Dinechin, 2000). Monet poikkeuksia tukevien ohjelmointikielten vakiokirjastot kuitenkin hallitsevat virhetilanteita poikkeuksilla, mikä pakottaa ohjelmoijan käyttämään poikkeuksia ohjelmoidessa.

4.2 Ada

Ada on Yhdysvaltain puolustusministeriön kehittämä ohjelmointikieli, joka suunniteltiin korvaamaan kaikki muut puolustusministeriön käyttämät ohjelmointikielet (Whitaker, 1996), muun muassa C:n. Toisin kuin monet muut ohjelmointikielet, Ada on suunniteltu monta vuotta kestäneen prosessin kautta (Tremblay, 1985, s. 121). Ada on hyvin moneen taipuva kieli, sillä se on suunniteltu hallitsemaan monia eri käyttötarkoituksia matalan tason bittitason ohjelmoinnista korkean tason arkkitehtuureihin.

Ohjelmoinnin helpottamiseksi Adassa on sekä poikkeukset että automaattinen muistinhallinta. Nämä kuitenkin hidastavat kieltä hieman aikaisemmin todetuista syistä. Lisäksi C-kielen kutsuminen on työlästä – jokainen C-funktio on yksitellen määritettävä kutsukonvention (engl. *calling convention*) kanssa (ISO/IEC, 2012, s. 471). Adan alustariippumaton C-tuki on kuitenkin äärimmäisen kattava, paikoitellen C:n omaa tukea kattavampi (C:n standardi ei kuvaile esimerkiksi kutsukonventioita[6], vaan ne on jätetty kunkin kääntäjätoteutuksen päätettäväksi). Ada on myös vertailun ainoa kieli, joka voi kutsua muilla ohjelmointikielillä kirjoitettuja kirjastorutiineja suoraan ilman C-rajapintojen käyttöä. Adassa on C:n lisäksi tuki C++:lle[7], Fortranille ja Cobolille (ISO/IEC, 2012, s. 585). Adassa ei kuitenkaan ole makrojärjestelmää, eikä Ada tue C:n makrojärjestelmää.

4.3 C++

C++ on Bjarne Stroustrupin 1980-luvulta eteenpäin kehittämä kieli, jonka yhtenä tarkoituksena on yhdistää Simula-kielen ominaisuudet ohjelman organisointiin yhteen C:n tehokkuuden ja joustavuuden kanssa (Stroustrup, 2007). C++ on nykypäivänä suosittu tehokkuutensa ja monipuolisuutensa takia monimutkaisissa ohjelmistoissa, kuten palvelinohjelmistoissa, kuvankäsittelyohjelmistoissa sekä peleissä (Stroustrup, 2014).

C++ on kehitetty C:n pohjalta ja C++:ssa onkin hyvä C-yhteensopivuus. Koska

[6] Esimerkiksi Windows-käyttöjärjestelmässä käytetään sekaisin kahta erilaista kutsukonventiota, sillä käyttöjärjestelmän rajapinnat käyttävät `stdcall`-kutsukonventiota ja ohjelmat yleensä `cdecl`-kutsukonventiota. Näin jokainen C-ohjelma joutuu käyttämään kunkin kääntäjän standardoimattomia ominaisuuksia.

[7] C++-tuki ei sisällä nimiruntelun tukemista, vaan kutsuttavista funktioista pitää määrittää runnellut nimet.

C++-funktiot nimirunnellaan eikä nimiruntelua ole määritelty tarkasti kielen standardissa, C++-koodia on hankalaa kutsua jopa samalla alustalla eri kääntäjien välillä – luvussa 4.1 on annettu esimerkki G++-kääntäjän nimiruntelemasta funktiosta. C-koodin otsikkotiedostoissa (engl. *header file*) on usein alussa C++-koodia, joka laittaa nimiruntelun pois päältä. Näin C++-ohjelmat voivat helposti kutsua C:llä kirjoitettujen kirjastojen funktioita, sillä C++-ohjelmat voivat käyttää C-kielen otsikkotiedostoja lähes aina ilman muita muokkauksia.

C++:n standardikirjaston virheidenkäsittely on toteutettu poikkeuksilla, jotka aiheuttavat pienen suoritusaikaisen hidastuksen. Monet vakiokirjaston funktioista ja metodeista voivat heittää poikkeuksen virhetilanteissa.

C++:ssa on mahdollista käyttää viitemäärälaskettua (engl. *reference counted*) muistinhallintaa (esimerkiksi vakiokirjaston `std::shared_ptr`), jolla voidaan käyttää suoritusaikaisesti varattua muistia ilman muistivuotoja. C++:n `std::shared_ptr` ei käytä erillistä roskien keräystä, vaan kun viimeinen viite olioon poistetaan, myös varattu muisti vapautetaan. Tällöin ohjelman suorituksen aikana ei tule roskienkeräystaukoja.

C++ tukee geneeristä ohjelmointia (engl. *generic programming*) luokkien yhteydessä malliohjelmoinnilla (engl. *template programming*). C++:n toteutuksessa jokaisesta uniikista mallin tyyppiparametrikombinaatiosta luodaan lopulliseen ohjelmaan kopio mallin ilmentymän (engl. *instance*) käytetyistä funktioista, joka kasvattaa ohjelmien kokoa. Tämä mahdollistaa jokaisen luokan ilmentymän erillisen optimoinnin, mutta yleisesti kasvattaa sekä kääntämisaikoja että ohjelmien kokoa.

C++ käyttää lähes samaa makrojärjestelmää kuin C. C++:n makrojärjestelmässä on 11 avainsanaa, joita ei voi määrittää uudelleen esikäsittelijässä (ISO/IEC, 2017b, luku 19.2)[8]. Tämä tarkoittaa sitä, että C++:n esikäsittelijä ei hyväksy joitakin C:n esikäsittelijän hyväksymiä makroja, tosin tämä ei tapahdu käytännössä koskaan, eli C++ voi käyttää suoraan C-ohjelmien otsikkotiedostoja. Koska makrojärjestelmä on muuten sama kuin C:n makrojärjestelmä, se on hyvin rajoittunut (ISO/IEC, 2017b, luku 19).

[8]Avainsanat ovat `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` sekä `xor_eq`.

4.4 D

D on 2000-luvun alussa Digital Mars -yrityksen julkaisema ohjelmointikieli, jonka tarkoituksena on mahdollistaa tehokkaiden ohjelmien kirjoittaminen helposti ja turvallisesti (D Language Foundation, 2020e). D:n kehityksessä tavoiteltiin C++:n mahdollistamaa tehokkuutta ja käännösaikaista varmennusta yksinkertaisemmassa muodossa (Alexandrescu, 2010, s. xix–xx). D on suunniteltu syntaksiltaan ja käytökseltään muistuttamaan C:tä ja C++:aa. Vaikka D-kielessä on olemassa automaattinen muistinhallinta, D:n *BetterC*-tila tekee kielestä ”paremman C:n” poistamalla suoritusaikaiset ominaisuudet, mukaan lukien automaattisen muistinhallinnan (D Language Foundation, 2020a). Tällöin kielestä poistuu useita ominaisuuksia, mutta esimerkiksi D:n käännösaikaista makrojärjestelmää voi käyttää.

C-koodin kutsuminen on melko helppoa, mutta ei aivan saumatonta, sillä jokainen kutsuttava funktio tulee määritellä erikseen – D ei ymmärrä C:n otsikkotiedostoja. Tämä kuitenkin onnistuu yhdellä rivillä jokaista C:n funktiota kohden, sillä D:n tyyppijärjestelmä on hyvin lähellä C:tä. D:lle on myös olemassa useita työkaluja otsikkotiedostojen automaattiseen muuntamiseen, kuten *Header to D*-työkalu (D Language Foundation, 2020d). Työkalut eivät kuitenkaan ole täydellisiä, eikä esimerkiksi C-makrojen käyttäminen ole ongelmatonta.

D ei sisällä tukea C:n makrojärjestelmälle. D:ssä ei myöskään ole C:n kaltaista esikäsittelijää, sillä sitä ei pidetty tarpeellisena (D Language Foundation, 2020b). D:ssä on kuitenkin käytettävissä C++:n kaltainen mallipohjainen makrojärjestelmä, jolla voidaan luoda geneerisiä funktioita. Kuten C++:ssä, D:llä kirjoitetuista geneerisistä funktioista luodaan kopiot jokaista tyyppikombinaatiota kohtaan.

4.5 Go

Go on Googlen kehittämä ja vuoden 2009 loppupuolella julkaisema ohjelmointikieli, jonka tarkoituksena on yhdistää käännösaikaisesti tyypitetyn ohjelmointikielen turvallisuus ja tehokkuus suoritusaikaisesti tyypitettyjen ohjelmointikielten helppokäyttöisyyteen (Pike, 2010). Toisin kuin monissa moderneissa C-perheen kielissä, Go-kielessä ei ole luokkia vaan pelkkiä tietueita ja rajapintoja. Go suunniteltiin erityisesti korvaamaan C++:n käyttö Googlella johtuen C++:n pitkistä käännösajoista.

Go-kielessä ei ole muista vertailtavista kielistä poiketen geneerisiä tyyppiparametreja. Tämä estää käännösaikaisesti tyyppitarkistetun geneerisen lähdekoodin kirjoittami-

sen. Ohjelmat voivat kuitenkin suoritusaikaisesti reflektion kautta tunnistaa muuttujien konkreettisen tyypin. Tämän mahdollistaminen kasvattaa ohjelmien suoritusaikaista muistinkäyttöä, sillä muuttujien mukana on säilytettävä tunniste muuttujan oikeasta tyypistä. Tämä kuitenkin yksinkertaistaa ohjelmien kirjoittamista, sillä ohjelmoijan ei tarvitse miettiä kirjoittamishetkellä monimutkaisia tyyppejä (esim. Pike, 2010, kalvo 8), kuitenkin mahdollistaen geneerisen lähdekoodin kirjoittamisen ilman käännösaikaisia geneerisiä malleja.

Vaikka Go-kielessä ei itsessään ole makroja, se sisältää `go generate` -työkalun, jota voidaan käyttää lähdekoodin generointiin (Pike, 2014). `go generate` mahdollistaa minkä tahansa komentorivikomennon suorittamisen ja on enemmänkin standardoitu tapa suorittaa tiettyjä komentorivikäskyjä osana ohjelman kääntämistä kuin tyypillinen makrojärjestelmä. `go generate` suoritetaan erillisenä komentona eikä esimerkiksi osana `go build` -komentoa.

Go-kielen virheidenkäsittely on toteutettu useissa kohdissa C:n tavoin; funktioista palautetaan virheellisissä tilanteissa virheellinen arvo. Tämä tosin tehdään usein palauttamalla erillinen `Error`-tyyppiä oleva arvo – Go mahdollistaa useamman kuin yhden paluuarvon. Go-kielessä on myös poikkeukset, joita suositellaan käytettävän vain poikkeuksellisissa tilanteissa, joita koodin kutsuja ei voi korjata suoritusaikaisesti (Google, Inc., 2020b). Tällainen tilanne voisi olla esimerkiksi väärä parametrin tyyppi. Poikkeuksen heittäminen voi myös olla perusteltua, mikäli kirjastorutiinin suoritus halutaan lopettaa syvällä yksityisissä funktioissa ja arvon halutaan menevän useiden funktioiden läpi kirjaston rajapintaan asti, jossa se muutetaan virhearvoksi.

C:n kutsuminen Go-kielestä ei ole aukotonta: koska Go on muistinkäytöltä turvallinen kieli, erityisesti muistin jakaminen C:n ja Go-kielen välillä on hankalaa. Lisäksi C:n funktio-osoittimia ei voi kutsua Go-kielen puolelta (Google, Inc., 2020a). Go mahdollistaa C-otsikkotiedostojen suoran käytön lähdekoodista, mikä helpottaa C-koodin kutsumista.

4.6 Rust

Rust on Mozilla Foundationin kehittämä ohjelmointikieli, joka on suunniteltu turvalliseksi, rinnakkaiseksi ja käytännölliseksi järjestelmäohjelmointikieleksi (Rust Project Developers, 2020b). Rust on vertailtavista kielistä yksilöllisin, sillä muut vertailtavat kielet eivät sisällä Rustin ominaisuuksia muuttujien omistuksesta.

Rustissa on monimutkainen tyyppijärjestelmä, jolla ohjelmat voivat todistaa esimer-

kiksi turvallisen rinnakkaisajon ilman, että ohjelmaan tulee suoritusaikaisia rajoitteita tai hidastuksia. Rust alkoi Graydon Hoaren henkilökohtaisena sivuprojektina, mutta on nyt käytössä esimerkiksi osana Gecko-selainmoottorin kehitystä C++:n ja JavaScriptin ohella.

Rustin tyyppijärjestelmä kannustaa kirjoittamaan turvallisia ohjelmia. Tietorakenteiden arvojen muuttaminen on tehty tietoisesti hankalaksi, sillä monimutkaisissa ohjelmissa holtittomasti muuttuva tila on usean vian syynä, jonka lisäksi muuttumaton tila tekee monisäikeistettyjen (engl. *multithreaded*) ohjelmien toteutuksesta huomattavasti helpompaa (Bloch, 2018, luku 4, kohta 17).

Rustin tyyppijärjestelmän ytimessä on käsitteet arvojen omistuksesta (engl. *ownership*), elinajoista (engl. *lifetime*) sekä muutettavuudesta (engl. *mutability*). Kun arvo sijoitetaan muuttujaan, sen elinajaksi asetetaan muuttujan elinaika, joka päättyy muuttujan poistuessa näkyvyysalueesta (engl. *scope*). Muuttujiin voidaan tehdä viitteitä, mutta viitteet eivät voi elää arvoja kauempaa, sillä kääntäjän elinaikatarkistaja (engl. *borrow checker*) pystyy seuraamaan arvojen elinaikoja. Kullakin muuttujalla voi olla joko rajaton määrä muuttumattomia viitteitä (engl. *immutable reference*) tai yksi muuttava viite (engl. *mutable reference*), mutta molempia ei voi käyttää yhtä aikaa.

```
fn main() {
    let s1 = "Hello".to_string();
    let s2 = s1;
    println!("{}, world!", s1);
}
```

Ohjelma 4.1: Tämä Rust-ohjelma aiheuttaa käännösvirheen rivillä neljä, sillä `s1`-muuttujan sisältö siirrettiin `s2`-muuttujaan rivillä 3.

Ohjelmassa 4.1 on yksinkertainen esimerkki tällaisesta käännösvirheestä. Muuttujaan `s1` asetetaan *omistettu merkkijono* 'Hello', jonka jälkeen muuttujan `s1` omistama merkkijono siirretään muuttujalle `s2`. Neljännellä rivillä yritetään käyttää `s1`-muuttujaa, mikä aiheuttaa käännösvirheen, sillä `s1`-muuttujan sisältö on siirretty, eikä `String`-tyyppisistä muuttujista voi luoda implisiittistä kopiota.

Ohjelmassa 4.2 on toinen vain Rustista löytyvä käännösvirhe: koska kolmannella rivillä otetaan muuttujaan `b` viite muuttujan `a` arvoon, muuttujan `a` arvon muokkaaminen estetään neljännellä rivillä.

```
fn main() {
    let mut a = "Foo".to_string();
    let b = &a;
    a += "bar";
    println!("{}", b);
}
```

Ohjelma 4.2: Tämäkin Rust-ohjelma aiheuttaa käännösvirheen, sillä muuttujan `a` arvoa yritetään muuttujaa, kun muuttuja `b` sisältää osoittimen muuttujan `a` arvoon.

Turvallisuudella on kuitenkin hintansa – Rust-ohjelmat vievät enemmän tilaa kuin vastaavat C-ohjelmat. Jos Rust-ohjelmista poistaa standardikirjaston ja käyttää suoraan C:n standardikirjastoa, ohjelmasta saa miltei samankokoisen kuin vastaavasta C-kielellä kirjoitetusta ohjelmasta (Rust Project Developers, 2018). Samalla tosin suurin osa Rustin ominaisuuksista jää pois. Rustin turvallisuus vaatii myös monimutkaisen tyyppijärjestelmän, joka on vaikeampi opetella kuin yksinkertaisemman kielen tyyppijärjestelmä.

Kuten Go-kielessä, Rustissa voi myös käyttää poikkeuksia. Rustin virheidenhallinta on muutenkin lähellä Go-kielen virhehallintaa – Rustin ohjekirja opastaa käyttämään mieluummin paluuarvoja kuin poikkeuksia (Rust Project Developers, 2020a).

Rust ei pysty suoraan käsittelemään C:n otsikkotiedostoja, mutta D:n tavoin Rustille on saatavilla työkaluja otsikkotiedostojen automaattiseen muuntamiseen (You, 2020). Rustin kehittäjät kuitenkin suosittelevat jokaisen kirjaston kohdalla kirjoittamaan käsin Rust-rajapinnan C-kirjastoille, sillä C:n tyyppimäärittelyt eivät tarjoa Rustin vaatimaa tarkkuutta funktioiden turvallisuudesta.

Rust sisältää kattavan makrojärjestelmän (Rust Project Developers, 2020d). Rust käsittelee makrot vasta ohjelman alkioanalyysin jälkeen, eli makroilla ei voi käsitellä mitä tahansa tekstiä. Rust-kääntäjän tulee tunnistaa kielen tekstialkiot ennen makrojen suorittamista, eli Rust ei mahdollista uusien operaattoreiden määrittämistä makrojen avulla. Rust vaatii myös erottimien (sulkeiden, lainausmerkkien ja heittomerkkien) olevan kielen syntaksin mukaisesti pareina. Rustin makroprosessori on sekä Turing-täydellinen että hygieeninen.

4.7 Yhteenveto

Yksikään vertailtavista kielistä ei täytä kaikkia luvussa 3.1 määriteltyjä rajoitteita. Yksikään kielistä ei täytä muistinkäytön rajoitteita, jonka lisäksi saumaton C:n käyttö onnistuu vain C++:n kanssa, mutta C++:kaan ei mahdollista täydellisen saumatonta C-yhteensopivuutta.

Taulukossa 4.1 on yhteenveto verrattavista ohjelmointikielistä sisältäen kunkin kielen nimiruntelun, muistinhallinnan, yhteensopivuuden C:n vierasfunktiorajapinnan kanssa sekä yhteensopivuuden muiden kielien vierasfunktiorajapintojen kanssa.

C++, D ja Rust mahdollistavat yhteensopivuuden parantamiseksi nimiruntelun poistamisen käytöstä, mutta tämä ei toimi esimerkiksi C++:n luokkien yhteydessä. Ada ja Go mahdollistavat funktioiden nimien valitsemisen linkittäjää varten, jolloin esimerkiksi `EsimerkkiFunktio`-nimistä funktiota voidaan kutsua `esimFunk`-nimellä C-ohjelmasta. Kaikki vertailtavat kielet tukevat vierasfunktiorajapintoja C:n mukaisesti, eli kielet voivat kutsua muita kieliä C:llä toteutettujen rajapintojen läpi.

C++ ja Rust ovat hyvin lähellä C:tä käännettyjen ohjelmien koossa, mutta häviävät C:lle laajojen ja paljon tilaa vievien vakiokirjastojen takia. Molemmat ovat myös hyvin monimutkaisia kieliä. Vain C++ ja Go voivat sisällyttää ohjelmiin C:n otsikkotiedostoja, kun taas Ada, D ja Rust vaativat jokaisen funktion määrittämistä. Adalle, D:lle ja Rustille on tosin olemassa työkaluja, joilla tämän määrittämisen voi automatisoida.

D, C++ ja Rust sisältävät mahdollisuuden käännösaikaisiin makroihin. C++:n makrojärjestelmä on melkein täysin yhteensopiva C:n makrojärjestelmän kanssa. Rustin makrojärjestelmä on taas huomattavasti ilmaisukykyisempi, muttei yhteensopiva C:n kanssa. Toisin kuin C++:n mallit, D:n mallit voivat ottaa parametreiksi vain tyyppejä.

Benchmarks Gamen tulokset myös heijastavat näitä tuloksia – C++ ja Rust ovat nopeudeltaan hyvin lähellä C:tä, kun taas Ada ja Go ovat huomattavasti hitaampia. Kuvassa 4.1 verrataan suorituskykyä, muistinkäyttöä ja ohjelmien lähdekoodin pituutta C:llä kirjoitettuun ohjelmaan. Lähdekoodimittauksessa mitataan ohjelman kokoa siten, että ohjelmasta poistetaan kommentit sekä ylimääräiset välimerkit. Tämän jälkeen ohjelma pakataan `gzip`-pakkausohjelmalla, kuten myös alkuperäisissä Benchmarks Gamen mittauksissa (Gouy, 2020b). Luvussa 6.1 kerrotaan tarkemmin

[9] Borrow Checkerin hallitsemana, joka mahdollistaa automaattisen muistinhallinnan ilman sen aiheuttamaa hidastusta. Rust sisältää myös viitemäärälasketun säiliön.

Kieli	Nimiruntelu	Muistinhallinta	Makrojärjestelmä
Ada	täysin hallittavissa	automaattinen	Ei ole
C++	saa osittain pois päältä	manuaalinen	Sama esikäsittelijä kuin C:ssä, erillinen kattava malliohjelmointi
D	saa pois päältä	molemmat	Tyyppipohjainen malliohjelmointi
Go	täysin hallittavissa	automaattinen	Ei ole
Rust	saa pois päältä	automaattinen[9]	Useita erilaisia makrojärjestelmiä

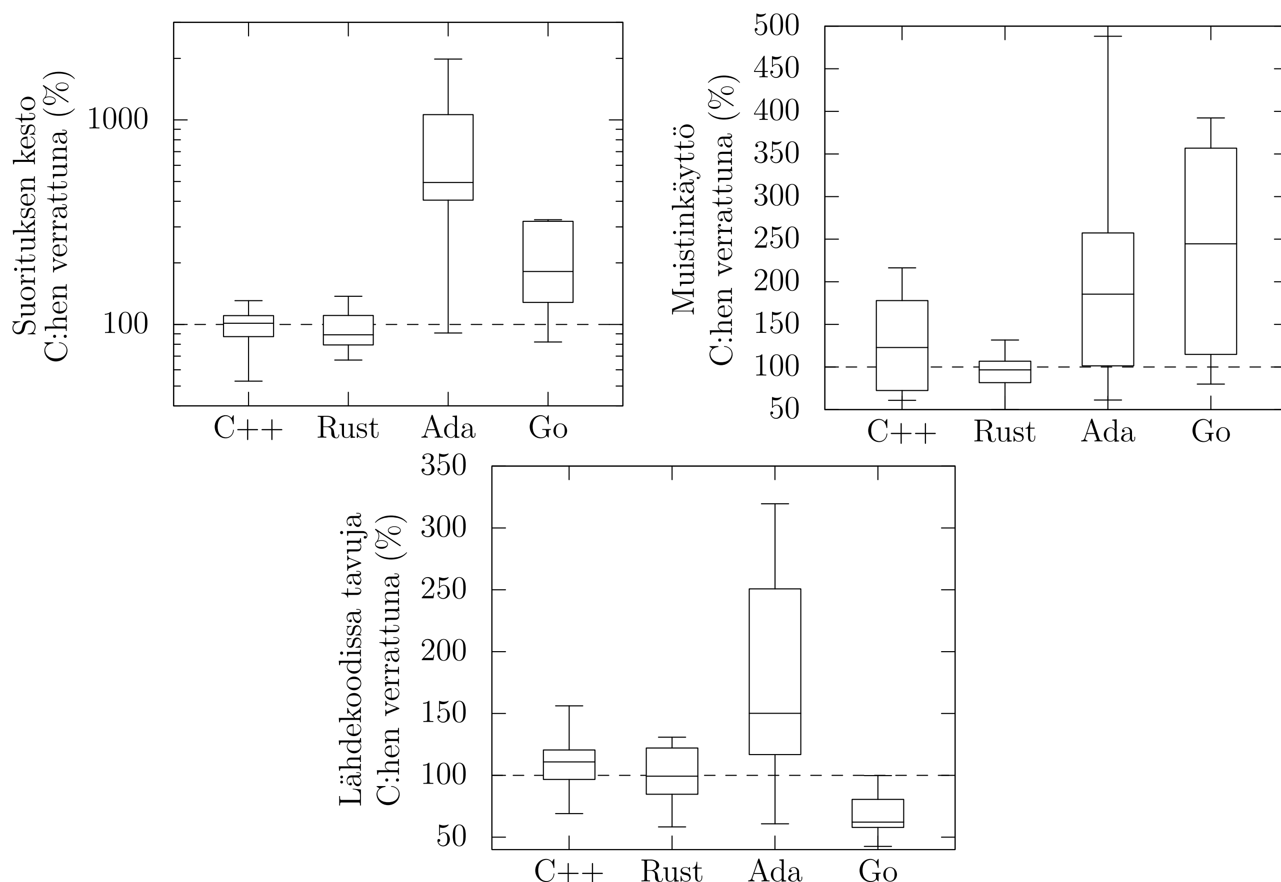
Kieli	Yhteensopivuus C:n kanssa	Yhteensopivuus muiden kielten kanssa
Ada	Kattava	C, C++, Fortran, Cobol
C++	Lähes saumaton	Vain C
D	Kattava	Vain C
Go	Yhteensopivuusongelmia	Vain C
Rust	Kattava	Vain C

Taulukko 4.1: Kielten ominaisuuksien yhteenveto.

mittauksista. Tarkat mittausarvot löytyvät liitteestä 1. Toisin kuin Benchmarks Gamessa, ennen lähdekoodin pakkaamista lähdekoodista on poistettu kaikki tyhjämerkit, jotta lähdekoodin asettelu ei vaikuta mittauksiin. Kuvaajat perustuvat nopeimman kielellä kirjoitetun ohjelman tuloksiin – lähes jokaisella verrattavalla kielellä on jokaisessa suorituskykymittauksissa useampi ohjelma.

C ja C++ ovat hyvin lähellä toisiaan suoritusnopeudessa. Rust on jonkin verran hitaampi, kun taas Ada ja Go ovat huomattavasti hitaampia. Kaikki verrattavat kielet käyttävät enemmän muistia kuin C. C++ ja Rust ovat yksittäisissä suorituskykymittauksissa C:tä nopeampia. Vertailtavilla kielillä kirjoitetut ohjelmat ovat C:llä kirjoitettuihin ohjelmiin keskimäärin hieman hitaampia ja vievät enemmän muistia, mutta eivät tarjoa merkittäviä säästöjä lähdekoodin määrään.

Benchmarks Game ei sisällä D:tä, eli Benchmarks Gamen tuloksia ei voi käyttää D:n vertaamiseen muiden kielten kanssa. Muissa Benchmarks Gamen kaltaisissa suorituskykymittauksissa (kuten kostya, 2020, jossa D on merkitty kääntäjätoteutuksesta riippuen joko DMD, LDC tai GDC) D on ollut suorituskyvyltään C:n ja Go-kielen välissä.



Kuva 4.1: Benchmarks Gamen (Gouy, 2020a) tuloksiin perustuvat kuvaajat ohjelmointikielten suorituskyvystä, muistinkäytöstä ja ohjelmien koosta verrattuna C:llä kirjoitettujen ohjelmien tuloksiin. Benchmarks Game ei sisällä verrattavista kielistä D:tä.

Nopeimpien ohjelmien lähdekoodeja lukiessa paljastuu, että useat ohjelmat ovat pyrkineet nopeampiin ohjelmiin luettavuuden kustannuksella. Tämä ilmenee kaikissa vertailtavissa kielissä: C-, C++- ja Ada-ohjelmissa käytetään GCC-laajennoksia, joita ei käytettäisi kääntäjäriippumattomuuteen, luettavuuteen ja ylläpidettävyyteen pyrkiessä. Go-ohjelmissa taas käytetään säännöllisten lausekkeiden käyttämiseen standardikirjaston toteutuksen sijaan C:llä kirjoitettua pcre-kirjastoa (Hazel, 2020).

5 Purkka-ohjelmointikieli

Jotta uudesta ohjelmointikielestä saisi tutkielman vertailukriteerien mukaisesti C:tä paremman kielen, uuden kielen suunnittelussa tulee seurata tarkasti vertailukriteerejä. Erityisesti C-yhteensopivuus on tärkeä ominaisuus, jota muissa ohjelmointikielissä ei ole pidetty tärkeänä ohjelmointikielen suunnittelussa. Tässä luvussa ensin käsitellään periaatteet, joilla Purkasta voisi tulla C:tä parempi ohjelmointikieli tutkielman kontekstissa, jonka jälkeen käsitellään tarkemmin Purkan tyypitystä, syntaksia sekä C-yhteensopivuutta.

5.1 Purkan suunnitteluperiaatteet

Luvut 2.1, 2.2 ja 2.3 nostavat tärkeinä C:n ominaisuuksina kielen yksinkertaisuuden ja tehokkuuden, jotka ovat mahdollistaneet kielen leviämisen järjestelmästä toiseen. Luvussa 2.3 nostetaan tämän lisäksi esiin vaatimus C-ohjelmien kirjoittamisesta funktio kerrallaan uudella kielellä, jotta kielestä toiseen siirtyminen olisi ylipäätään mahdollista ilman kohtuutonta investointia. Luvussa 2.4 esitellään lukuisia C:n syntaktisia ongelmia, jotka voi muuttaa tehden kielestä helppolukuisemman, kuten tyyppipäättelyn lisääminen ohjelmointikieleen.

Käännösaikaista varmennettavuutta voi parantaa lisäämällä esimerkiksi kieleen summatyypit, jotka löytyvät esimerkiksi Rustista. Yhteensopivuus nousee myös esiin luvussa 3.3, jossa kirjastojen saatavuus näytetään tärkeäksi ohjelmointikielen valintakriteeriksi.

Uuden kielen määrittelyssä tulee pitää luvun 3.1 kriteerit, jotta ominaisuuksia päättäessä ei muodostu esimerkiksi yhteensopivuusongelmia C:n kanssa. Tämä rajoittaakin suurta osaa luvussa 4 esiteltyjä ominaisuuksia, kuten automaattista muistinhallintaa ja poikkeuksia. Monet ominaisuudet myös monimutkaistaisivat kieltä tarjoamatta kuitenkaan tehokkuusparannuksia.

Yksittäisiä ominaisuuksia kuitenkin pystyy lisäämään, kuten luvuissa 4.5 ja 4.6 esiintyvät useat paluuarvot sekä luvussa 4.6 esitellyt summatyypit, sillä näiden sisällyttäminen kieleen mahdollistaa paremman käännösaikaisen todentamisen ilman suoritusaikaisia haittoja. Summatyypit voivat myös parantaa kääntäjäoptimointia, jos kääntäjä pystyy poistamaan ohjelmakoodia päättelemällä tietyt summatyypin arvot mahdottomiksi.

Myös epätyhjiä osoittimia varten voi lisätä tyypin, jonka voi käännösaikaisesti

muuntaa mahdollisesti tyhjäksi osoittimeksi. Tyhjiä osoittimia on syytetty miljardin dollarin virheeksi (Hoare, 2009). Epätyhjien osoittimien käyttö parantaa käännösaikaista varmennusta. Esimerkiksi Kotlin-ohjelmointikielessä (JetBrains, 2020b) tyhjien osoittimien turvaton käsittely on estetty (JetBrains, 2020a)

Uusi kieli tulisi kääntää C:ksi, jotta sen käyttäminen on mahdollista kaikissa järjestelmissä, joissa C:tä käytetään. C:ksi kääntäminen myös mahdollistaa C-yhteensopivien kirjastojen käyttämisen ilman erillistä yhteensopivuuskerrosta. Kielen tulee myös ymmärtää C:llä kirjoitettuja ohjelmia sisältäen myös C:llä kirjoitetut makrot ja otsikkotiedostot. Esimerkiksi POSIX-C:n `errno`-muuttuja voi olla määritelty makrona ja `errno`-muuttujan käyttäminen ilman esikäsittelijää esimerkiksi viittaamalla suoraan `errno`-nimiseen muuttujaan on määrittelemätöntä toimintaa (ISO/IEC, 2017a, s. 234), joten `errno`-arvon lukeminen vaatii tuen C:n esikäsittelijälle.

5.2 Tyypit

Jotta yhteensopivuus C:n kanssa olisi joustavaa, Purkan tyyppijärjestelmä muodostetaan mahdollisimman paljon C:n kaltaiseksi. C:n tyyppisyntaksi sisältää useita erilaisia tapoja ilmaista samaa pohjatyyppiä – esimerkiksi `long`, `signed long`, `long int` ja `signed long int` ilmaisevat kokonaislukutyyppiä, joka pystyy sisältämään ainakin luvut $[-(2^{31}-1), 2^{31}-1]$[10] (ISO/IEC, 2018, luku 5.2.4.2.1). Tyypit kuten `long` ja `signed long` ovat kuitenkin standardien mukaisessa C:ssä keskenään täysin vaihdettavissa, eli Purkan ei tarvitse pystyä kääntymään jokaiseen mahdolliseen vaihtoehtoon, vaan tyyppijärjestelmässä voi olla yksi tyyppi joka vastaa kaikkia `long`-tyypin vaihtoehtoisia kirjoitusasuja.

C:n kokonaislukutyypeistä (`char`, `short`, `int`, `long`) `char`-tyyppi on ainoa, jolla `char`, `signed char` ja `unsigned char` ovat kolme erillistä tyyppiä (ISO/IEC, 2018), kun muilla alkeistyypillä esimerkiksi `signed int` on sama tyyppi kuin `int`. Merkkijonot koostuvat `char`-tyyppisistä alkioista, kun taas `signed char` ja `unsigned char` vastaavat tavun kokoista kokonaislukua ja epänegatiivista kokonaislukua. Purkka-kielessä C:n `char`-tyyppiä vastaa `char`, kun taas `signed char` ja `unsigned char` ovat `i8` ja `u8`.

Tyyppien kääntäminen C:stä Purkaksi ja takaisin on yksinkertaista: jokaista C-tyyppiä vastaa täsmälleen yksi Purkka-tyyppi. Jokaiselle Purkan alkeistyypille taas

[10] Käytännössä modernit toteutukset käyttävät kahden komplementtia kokonaislukujen ilmaisemiseen, jolloin 32-bittinen kokonaislukumuuttuja voi sisältää luvut $[-2^{31}, 2^{31}-1]$.

on määritelty yksi C-tyyppi, johon kyseinen Purkka-tyyppi käännetään. Liitteessä 2 on taulukko C:n ja Purkan tyypeistä.

C:n alkeistyyppien, osoittimien, taulukoiden ja yhdistetyyppien lisäksi määritellään ohjelmoinnin tehostamiseksi sekä käännösaikaisen optimoinnin ja oikeellisuuden parantamiseksi yksittäisiä lisätyyppejä. Nämä tyypit ovat epätyhjät osoittimet sekä summatyypit.

Epätyhjä osoitin on osoitin, joka ei salli arvokseen C:n `NULL`-arvoa. Tyyppi kääntyy Purkasta standardien mukaiseksi C:ksi yksinkertaisesti vastaavaan C:n mahdollisesti tyhjään osoitintyyppiin. Epätyhjien osoittimien käyttö parantaa käännösaikaista varmennusta. Epätyhjä osoitin myös mahdollistaa kääntäjäoptimointeja, jotka eivät muuten olisi turvallisesti mahdollisia, kuten turhien tarkistusten poistamisen. GCC-kääntäjä mahdollistaa epätyhjät osoittimet `__attribute__((nonnull))`-määreellä, jota Purkka-kääntäjä voi käyttää ohjelmoijan niin pyytäessä.

```
enum Puu {
    Lehti(i32),
    Haara(&Puu, &Puu)
}
```

Ohjelma 5.1: Summatyyppi Purkka-kielessä.

```
struct Puu;

enum _Puu_d { _PUU_LEHTI, _PUU_HAARA };

struct _Puu_Lehti { int _1; };

struct _Puu_Haara { struct Puu *_1; struct Puu *_2; };

struct Puu {
    enum _Puu_d v;
    union { struct _Puu_Lehti Lehti; struct _Puu_Haara Haara; };
};
```

Ohjelma 5.2: Ohjelman 5.1 summatyyppi suoraviivaisesti käännettynä C-kielelle.

```
struct Puu;

struct _Puu_Lehti { struct Puu *_d; int _1; };

struct _Puu_Haara { struct Puu *_1; struct Puu *_2; };

union Puu { struct _Puu_Lehti Lehti; struct _Puu_Haara Haara; };
```

Ohjelma 5.3: Ohjelman 5.2 optimoitu C-versio.

Summatyyppi on yhdistelmä C:n `struct`-, `union`- ja `enum`-tyypeistä. Summatyypit mahdollistavat `union`-tyyppien käytön käännösaikaisella varmennuksella. Summatyyppi kääntyy `struct`-tyypiksi, jossa on `union`-tyyppi, joka sisältää kaikki summatyypin variantit sekä `enum`-tyyppinen muuttuja summatyypin varianttien erottimena eli diskriminanttina.

Ohjelmassa 5.1 on Purkka-kielellä kirjoitettu summatyyppi `Puu`, jossa on `Lehti`- ja `Haara`-variantit. `Lehti`-variantti sisältää 32-bittisen kokonaisluvun, kun taas `Haara`-variantti sisältää osoittimet kahteen alipuuhun. Kääntäessä tietorakenne muutetaan ohjelman 5.2 kaltaiseksi lähdekoodiksi. Tässä erikoistapauksessa kääntäjä voisi optimoida tietorakennetta ohjelman 5.3 mukaisesti säilömällä `_Puu_d` -tyypin tiedon `_Puu_Lehti`- ja `_Puu_Haara`-tyyppien ensimmäiseen alkioon: jos osoitin on tyhjä, on kyseessä `Lehti`-variantti, muulloin `Haara`-variantti.

5.3 Syntaksi

Purkan syntaksi on hyvin samankaltainen C:n syntaksiin verrattuna. Tämä helpottaa kielen oppimista aikaisemman tutkimuksen mukaisesti (Scholtz & Wiedenbeck, 1990). Samankaltainen syntaksi myös helpottaa nykyisten ohjelmien uudelleenkirjoitusta, sillä ohjelmien rakennetta ei tarvitse muuntaa. Suurimmat erot C:hen liittyvät tyyppien kirjoittamiseen, jossa syntaksia on muutettu modernien ohjelmointikielten syntaksin mukaiseksi.

Funktiomäärittelyt alkavat `fun`-avainsanalla. Tämä noudattaa usean muun modernin ohjelmointikielen tapaa aloittaa funktiomäärittely avainsanalla. Vertailun vuoksi Rust käyttää avainsanaa `fn`, Go käyttää avainsanaa `func` ja Kotlin käyttää avainsanaa `fun`. Avainsanan käyttäminen yksinkertaistaa kielen kielioppia ja jäsentäjän toteutusta.

Purkka-kieli sisältää tyyppipäättelyn, joka helpottaa koodin kirjoittamista, kun kääntäjä pystyy päättelemään muuttujien tyypit. C-kielen mahdollisuuksia automaattiseen tyyppipäättelyä on tutkittu aikaisemminkin (mm. Melo, Ribeiro, de Araújo, & Pereira, 2017). Koska Purkan syntaksi on yksiselitteinen, monet artikkelissa esitetyt C:n ongelmat eivät ole Purkassa haittana. Toisaalta esimerkiksi yhteenlasku toimii samoin kuin C:ssä, joten sen tyyppipäättely ei ole triviaalia: jos C:ssä tai Purkassa toinen yhteenlaskettavista on osoitin, summan tyyppi on osoitin, muutoin tyyppi muodostuu C:n ”tavallisten” lukujen muunnossääntöjen mukaisesti.

Koska tyyppipäättely toteutetaan puhtaasti käännösaikaisesti, se ei voi aiheuttaa suoritusaikaisia haittoja. Tyyppisyntaksi itsessään muistuttaa paljon Rustin tyyppisyntaksia.

C:n `static`-avainsana on jaettu käytön mukaisesti kahteen avainsanaan. Funktiomäärittelyissä ja globaaleissa muuttujissa C:n `static`-avainsanaa vastaa Purkan `pub`-avainsana (”public” eli julkinen), mutta käänteisellä merkityksellä: jos `pub`-avainsanaa ei ole käytetty, koodi käyttäytyy kuin siihen olisi lisätty C:n `static`-avainsana. Funktioiden sisällä Purkka tukee myös `static`-avainsanaa, joka toimii samalla tavalla kuin C:n `static` käytettynä funktioiden sisällä.

C käsittelee tietueiden ja taulukoiden alustamista identtisellä syntaksilla. Purkka erottaa nämä kaksi syntaksia erikseen Rust-ohjelmointikielen syntaksin mukaisesti. Tietueen tyypin nimen pitäminen mukana tietueliteraaleissa pitää tyypin selkeästi näkyvissä myös tyyppipäättelyä käytettäessä. Yksinkertaisemmissa tapauksissa tämä ei vaikuta lähdekoodin pituuteen, mutta monimutkaisten tietueiden alustuksessa Purkka-koodi on hieman pidempää kuin vastaava C-koodi.

Suurin osa C:n lauseista on Purkassa lausekkeita. Esimerkiksi `if-else` lausepari on Purkassa lauseke, joka mahdollistaa selkeämmän koodin kirjoittamisen. C:ssä vastaavia lausekkeita voi kirjoittaa käyttämällä välimuuttujaa tai yksinkertaisissa tapauksissa `?:`-operaattorilla.

5.4 C-yhteensopivuus

Koska Purkan tulee olla myös makrojen osalta C-yhteensopiva, Purkka osaa laajentaa C-makroja. Purkka-kääntäjä pystyy muuntamaan makrokutsun C-lähdekoodiksi, laajentamaan sen C-esikäsittelijällä ja muuntamaan makron laajennetun muodon takaisin Purkka-koodiksi.

Purkka ei sisällä yhtään suoritusaikaista ominaisuutta, joita ei ole C:ssä. Tämä pitää kielen yksinkertaisena ja C-yhteensopivana. Tämä toisaalta tekee kielellä ohjelmoinnista työläämpää verrattuna muihin moderneihin ohjelmointikieliin.

Useat C-toteutukset sisältävät erilaisia laajennoksia, joiden tarkoituksena on mahdollistaa erilaisten alustariippuvaisten ominaisuuksien käyttö. Benchmarks Gamessa yksi käytetyimmistä laajennoksista on SIMD-tyypit, joita varten esimerkiksi GCC-kääntäjällä on oma syntaksinsa. GCC:llä SIMD-tyypit voidaan määritellä käyttämällä `__attribute__((vector_size))` -määrettä. Purkka tukee useita GCC:n laajennoksia, muun muassa vektorityyppejä.

Syntaksi sisältää myös `pragma`-avainsanan, jota voidaan käyttää vastaavasti kuin C:n esikäsittelijän `pragma`-direktiiviä erilaisten laajennosten käyttämiseen. Esimerkiksi OpenMP-laajennosta käytetään pragmojen läpi.

Jotta Purkkaa voisi käyttää mahdollisimman helposti nykyisten järjestelmien kanssa, Purkka käännetään C-koodiksi. Tämä mahdollistaa olemassa olevien C-kääntäjien käyttöä Purkan kääntämiseen kaikille mahdollisille alustoille, joille on olemassa standardien mukainen C-kääntäjä. Esimerkiksi `make`-työkalua käyttäville projekteille riittää yksi Makefile-sääntö Purkka-ohjelmien kääntäminen C-ohjelmiksi. Tämä sääntö on esitetty ohjelmassa 5.4. Muissa käännösautomaatiosovelluksissa Purkan integrointi osa kääntämistä on todennäköisesti yhtä helppoa.

```
%.c : %.prk
	purkka $< -o $@
```

Ohjelma 5.4: Kaksi riviä Makefile-syntaksia riittää Purkan integroimiseen Make-ohjelmaa käyttäviin projekteihin.

6 Uuden ohjelmointikielen vertaaminen C:hen

Tässä luvussa verrataan uuden ohjelmointikielen suorituskykymittauksia C:hen sekä muihin tutkielmassa käsiteltyihin kieliin. Suorituskykymittaukset näyttävät, että Purkan toteutus ei hidasta C:tä. Tämä taas johtuu siitä, että Purkassa ei ole yhtään ominaisuutta, joita C:ssä ei ole. Tässä luvussa myös tutkielman oikeellisuutta ja pohditaan jatkotutkimuskohteita.

6.1 Vertailun tulokset

Tätä tutkielmaa varten on toteutettu kääntäjä, joka kääntää yksinkertaiset ohjelmat Purkka-kielestä C-kieleen (Hannikainen, 2020). Kääntäjään ei ole toteutettu kaikkia luvussa 5 esitellyistä ominaisuuksia, mutta kääntäjä tukee esimerkiksi C:n esikäsittelijää.

Nopeimmat Benchmarks Gamen C-toteutukset on käännetty Purkka-kielelle. Nämä ohjelmat on sitten käännetty Purkka-kääntäjällä takaisin C:ksi ja käännetty sitten GCC-kääntäjällä vastaavilla asetuksilla kuin verrattavat C-ohjelmat. Kuvassa 6.2 verrataan Purkka-toteutuksia muihin verrattaviin kieliin ja kuvassa 6.3 verrataan Purkka-toteutuksia nopeimpaan C-ohjelmaan. Koska Purkka-toteutukset on muunnettu nopeimmasta C-toteutuksesta ja käännetty takaisin lähes identtiseen muotoon, suoritusajan ja muistinkäytön tulisi olla samat verrattuna C-toteutukseen.

Suorituskykymittaukset mittaavat ainoastaan prosessoriin ja muistinkäyttöön liittyviä mittauksia, eivätkä esimerkiksi näytönohjaimella suoritettua laskentaa. Mittaukset tehdään vain yksittäisillä kääntäjäversioilla, joten mittaukset eivät välttämättä päde muilla saman ohjelmointikielen kääntäjillä.

Suorituskykymittaukset on suoritettu Ubuntu 19.10 -käyttöjärjestelmällä Linux 5.3.0 -kernelillä. Mittaustietokoneessa on neliytiminen Intel i7-8550U -prosessori Hyper-Threading-tuella (4.0 GHz) ja 32 gigatavua DDR3-muistia. C- ja C++-ohjelmat on käännetty GCC-kääntäjän versiolla 9.2.1, Ada-ohjelmat GNAT-kääntäjän versiolla 8.3.0, Go-ohjelmat Go-kääntäjän versiolla 1.12.10 ja Rust-ohjelmat Rust-kääntäjän versiolla 1.41.0.

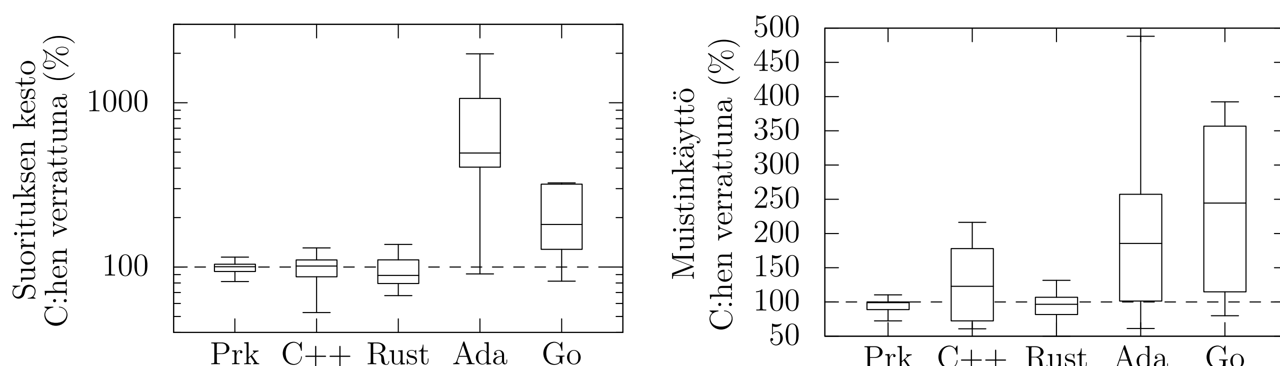
Mittaukset on toteutettu suorittamalla ohjelma suurimmalla syötteellä viisi kertaa ja mittaamalla kunkin suorituksen viemä aika ja muistinkäyttö. Muistinkäyttö on mitattu 200 millisekunnin välein `libgtop2`-kirjastolla ja kunkin suorituksen muistinkäytöksi on merkitty suurin mitattu muistikäyttö.

Kuvaajista nähdään, että Benchmarks Gamen tarjoamissa esimerkeissä Purkka on täsmälleen yhtä tehokas kieli kuin C, eikä se vie enempää muistia. Pienet eroavaisuudet suuntaan ja toiseen on selitettävissä testauksen aiheuttamalla luonnollisella vaihtelulla. Purkalla toteutettu `fannkuchredux`-ohjelma on kuitenkin noin 50% nopeampi kuin C:llä toteutettu lähes identtinen versio ja `nbody`-ohjelma vie vain 80% muistia verrattuna C-toteutukseen.

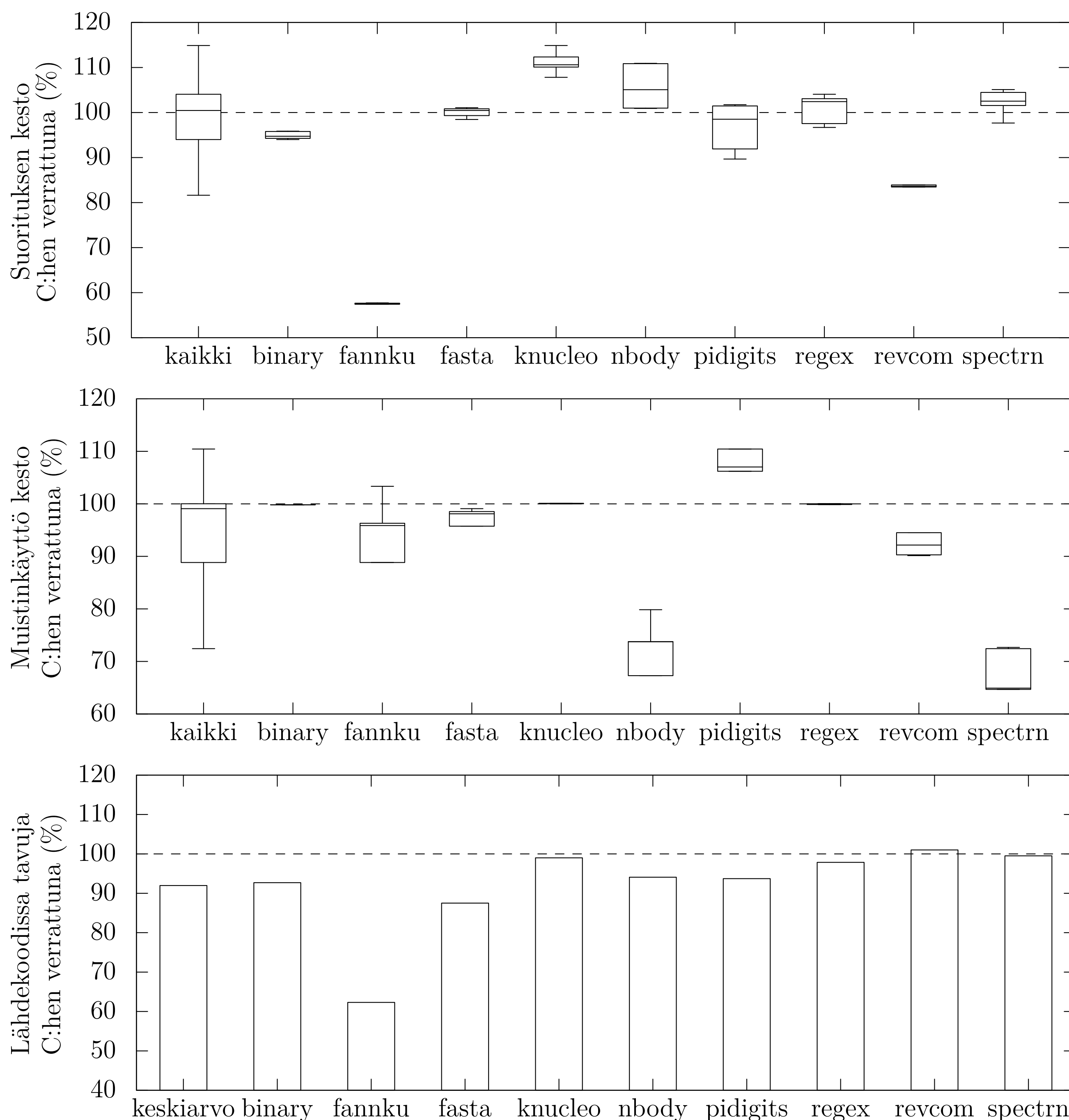
Purkka-tiedostot ovat keskimäärin 6% pienempiä kuin vastaavat C-tiedostot. Suurimmat kokoerot tiedostoissa tulevat muuttujien määrittelyistä sekä tyyppipäättelystä. Kuvassa 6.1 on yksittäisiä tyyppimäärittelyjä, jotka ovat Purkassa hieman yksinkertaisempia kuin vastaavat C-määrittelyt.

Purkka-määrittely ja C-määrittely	Selite
`let a: u32;` `unsigned int a;`	Epänegatiivinen 32-bittinen kokonaisluku
`let a, b: [&i8;5];` `signed char *a[5], *b[5];`	Taulukko viidestä osoitinmuuttujasta 8-bittiseen kokonaislukuun
`const a = ["1", "2", "3"];` `const char * const a[] = {"1", "2", "3"};`	Kolmen merkkijonon taulukon vakiomuuttuja
`let a = fun (a, b) => a + b;` `int a(int a, int b) { return a + b; }`	Funktio, joka laskee argumenttiensa summan

Kuva 6.1: Purkan muuttujien määrittelyt verrattuna C:n vastaaviin määrittelyihin.



Kuva 6.2: Benchmarks Gamen ohjelmiin perustuvat kuvaajat Purkalla kirjoitettujen ohjelmien suorituskyvystä, muistinkäytöstä ja ohjelmien koosta verrattuna muilla kielillä kirjoitettujen ohjelmien tuloksiin.



Kuva 6.3: Benchmarks Gamen ohjelmiin perustuvat kuvaajat Purkalla kirjoitettujen ohjelmien suorituskyvystä, muistinkäytöstä ja ohjelmien koosta verrattuna C:llä kirjoitettujen ohjelmien tuloksiin.

6.2 Johtopäätökset ja vertailun arviointi

Purkka on vertailun ainoa kieli, joka on yhtä tehokas kuin C. Tämä johtunee pitkälti siitä, että Purkka käännetään lähes identtisenä C:ksi, eikä tarjoa suoritusaikaisia ominaisuuksia, jotka hidastaisivat kieltä. Purkka ei kuitenkaan päihitä C:tä tutkimuksen määrittelyn mukaisesti, vaan on suorituskyvyltään täsmälleen yhtä tehokas kuin C.

Kaikkien vertailtavien kielten suunnittelutavoitteissa on mainittu C:n lisäksi myös C++:n käytön korvaaminen. Tämä on todennäköisesti vaikuttanut kielen suunnitteluun monimutkaistaen kieliä, jotta kieli pystyisi korvaamaan C++:n monimutkaiset malliohjelmointirakenteet. Muista kielistä poiketen Purkan tavoitteena on korvata ainoastaan C, mikä on ohjannut suunnittelua tutkielman määrittelyjen mukaisesti.

Suorituskykymittaukset eivät kuitenkaan vastaa todellista maailmaa, sillä ne usein mittaavat vain yksittäistä pientä osa-aluetta, kuten yksittäisten operaatioiden nopeutta monimutkaisten ohjelmistojen sijaan. Lisäksi suorituskykymittaukset ovat usein epätarkkoja, sillä monimutkaiset moniajoympäristöt eivät mahdollista deterministisiä mittauksia.

Mittaukset koskevat vain yksittäisiä ohjelmia ajettuna tietyssä ympäristössä tietyllä kääntäjällä. Ne eivät siis anna kattavaa kuvaa ohjelmointikielistä, vaan mittaustuloksia yksittäisen kääntäjätoteutuksen kääntämistä ohjelmista. Kääntäjät voivat käyttää jopa yksittäisten versioiden välillä erilaisia optimointeja, jotka voivat nopeuttaa tai hidastaa ohjelmaa juuri mitatulla syötteellä, mikä pienentää tulosten vertailukelpoisuutta. Kääntäjät voivat myös toimia paremmin tai huonommin erilaisilla tietokonearkkitehtuureilla, jolloin esimerkiksi prosessorin valinta vaikuttaa mittaustuloksiin.

Purkkaa ei todennäköisesti oteta laajaan käyttöön, sillä se ei tarjoa merkittäviä parannuksia C:hen, vaan mahdollistaa lähinnä hieman lyhyemmän lähdekoodin. Ohjelmointikielten valintaan liittyy usein monimutkaisia syitä, kuten aikaisempi kokemus tai ohjelmoijan henkilökohtainen mieltymys johonkin kieleen, eikä projekteissa käytettyjä ohjelmointikieliä valita pelkästään kielen ominaisuuksien perusteella. Tämän lisäksi Purkan toteuttaja on yksittäinen opiskelija, kun taas esimerkiksi Go-kielen ja Rustin toteuttajat ovat kokeneita ohjelmointikielten suunnittelijoita, jonka lisäksi kieliä ylläpidetään aktiivisesti tunnettujen organisaatioiden toimesta. Tämä on todennäköisesti myös syy sille, miksi esimerkiksi LISP/c (Baca, 2016), C-Mera (kiselgra, 2019), Carp (Svedäng & Heller, 2020) ja Nymph (Barber, 2020) ovat jääneet

ilman laajempaa huomiota.

Tutkimuksesta voidaan kuitenkin päätellä, että C:tä parempi kieli on todennäköisesti toteutettavissa. Käännösaikaista turvallisuutta voi parantaa tiukemmalla tyyppijärjestelmällä ilman suoritusaikaisia haittoja. Tarkemmilla tyyppimäärittelyillä voi myös tehdä optimointeja, esimerkiksi vaatimalla osoitinargumentit aina ei-tyhjiksi osoittimiksi.

Osan Purkan ominaisuuksista voisi ottaa käyttöön jopa uusissa C:n versioissa. Tyyppipäättelyn lisääminen esimerkiksi korvaamalla käyttämättömän `auto`-avainsanan[11] tarkoittamaan pääteltyä tyyppiä voisi auttaa moderneja C-kääntäjiä käyttävien projektien kirjoittamista. GCC-kääntäjä tukee tyyppipäättelyä `__auto_type`-määreellä. Summatyyppien lisääminen laajennoksena `enum`- tai `union`-tyyppisyntaksiin ei vaikuta tämänhetkisiin standardien mukaisiin ohjelmiin, mutta mahdollistaisi summatyyppien käytön. Suurempia syntaktisia muutoksia, kuten tyyppisyntaksin uudelleenkirjoitusta tuskin pystyy toteuttamaan säilyttäen yhä tuen nykyiselle C:lle.

Purkan jatkokehityksen voisi aloittaa omalla makrojärjestelmällä, sillä Purkassa ei ole määriteltynä omaa makrojärjestelmää. Purkka tukee C:n makrojärjestelmää, mutta C:n makrojärjestelmää kattavamman makrojärjestelmä voisi helpottaa monimutkaisten käännösaikaisten laskujen laskelmista.

Purkkaa varten ei ole kääntäjän lisäksi kehitetty yhtään kehitystyökalua, vaikka luvussa 3.3 valmiit työkalut nostetaan tärkeäksi ominaisuudeksi. Esimerkiksi hyvälaatuinen automaattinen käännösjärjestelmä helpottaisi kielen käyttämissä uusissa ohjelmissa. Tämän voisi nimetä ohjelmointikielen nimen mukaisella teemalla Jesariksi, sillä se sitoo käännettävät ohjelman palaset toisiinsa yhdistettynä mahdollisiin riippuvuuksiin.

[11] Avainsanaa käytetään tarkkaan ottaen `static`-avainsanan vastinparina, eli ei-staattisten muuttujien luomiseen. Koska kaikki muuttujat ilman `static`-määrettä ovat ei-staattisia, `auto`-avainsana on turha.

7 Yhteenveto

Tutkielmassa selvitettiin mahdollisuuksia parantaa C-ohjelmointikieltä. Tätä tarkoitusta varten luotiin uusi ohjelmointikieli, Purkka. Tutkimuksessa verrattiin useita erilaisia ohjelmointikieliä C:hen suorituskyvyn sekä muistinkäytön perusteella, mutta yksikään verrattavista ohjelmointikielistä ei ollut C:tä nopeampi kuin yksittäisissä mittauksissa. Purkka tarjoaa pienen syntaktisen parannuksen C:hen ilman suoritusaikaisia haittoja. C:n parantaminen on siis mahdollista ainakin syntaksin osalta.

Tutkielmassa selvitettiin tärkeimpiä C:n ominaisuuksia, joiden takia C on yhä käytössä. Ominaisuuksista nousi esiin kielen yksinkertaisuus, tehokkuus ja alustariippumattomuus. Mahdollisten korvaavien kielten tulisi ymmärtää C:n esikäsittelijää varten kirjoitettuja otsikkotiedostoja, jotta kielestä toiseen siirtyminen olisi mahdollisimman yksinkertaista. Kielen tulisi myös mahdollistaa yksittäisten funktioiden muuntaminen kerrallaan uuden ohjelmointikielen mukaisiksi, jotta kielestä toiseen siirtyminen ei vaadi huomattavia investointeja.

Jotta kielestä toiseen siirtyminen olisi kannattavaa, uuden kielen tulisi tarjota huomattavia parannuksia C:hen verrattuna. Tutkielmassa löydettiin mahdollisiksi ominaisuuksiksi makrojärjestelmän ja syntaksin parantaminen sekä summatyypit. Nämä eivät kuitenkaan välttämättä ole yksinään tarpeeksi suuri tekijä, jotta C:stä luopuminen olisi järkevää.

Verrattavat ohjelmointikielet häviävät C:lle eniten yhteensopivuuden osalta. Vain C++ osaa käsitellä C:n otsikkotiedostoja, kun muissa verrattavissa kielissä on epäkäytännöllistä kutsua jopa yksittäisiä C-funktioita.

Tutkielmassa määritellystä ohjelmointikielestä voisi ottaa yksittäisiä ominaisuuksia uusiin C-standardeihin vaikuttamatta yhteensopivuuteen vanhempien standardien mukaisiin ohjelmiin. Tämä onnistuu ainakin summatyypeille, jotka voisi lisätä C-standardiin tavallisen `enum`- tai `union`-tyyppien yhteyteen.

Lähteet

Alexandrescu, A. (2010). *The D Programming Language* (1st painos). Addison-Wesley Professional.

Baca, J. (2016). *eratosthenesia/lispc: "Lispsy" Lisp(ish) to C Converter (designed for CLISP).* `https://github.com/eratosthenesia/lispc`.

Barber, B. (2020). *roecrew/nymph: A slightly different version of C.* `https://github.com/roecrew/nymph`.

Bloch, J. (2018). *Effective Java* (3. painos). Boston, MA: Addison-Wesley.

Chatley, R., Donaldson, A., & Mycroft, A. (2019). The next 7000 programming languages. Teoksessa B. Steffen & G. Woeginger (toim.), *Computing and software science: State of the art and perspectives* (s. 250–282). Cham: Springer International Publishing. doi: 10.1007/978-3-319-91908-9$_1$5

D Language Foundation. (2020a). *Better C - D Programming Language.* `https://dlang.org/spec/betterc.html`.

D Language Foundation. (2020b). *The C Preprocessor vs D - D Programming Language.* `https://dlang.org/articles/pretod.html`.

D Language Foundation. (2020c). *D Programming Language.* `http://dlang.org/spec/spec.html`.

D Language Foundation. (2020d). *htod - D Programming Language.* `https://dlang.org/htod.html`.

D Language Foundation. (2020e). *Overview - D Programming Language.* `https://dlang.org/overview.html`.

de Dinechin, C. (2000). C++ exception handling. *IEEE Concurrency*, *8*(4), 72-79. doi: 10.1109/4434.895109

Free Software Foundation, Inc. (2020). *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF).* `https://gcc.gnu.org/`.

Gil, J., & Lorenz, D. H. (1998). Design patterns and language design. *Computer*, *31*(3), 118-120. doi: 10.1109/2.660196

GitHub, Inc. (2019). *The State of the Octoverse | The State of the Octoverse reflects on 2018 so far, teamwork across time zones, and 1.1 billion contributions.* `https://octoverse.github.com/`.

Google, Inc. (2020a). *cgo - The Go Programming Language.* `https://golang.org/cmd/cgo/`.

Google, Inc. (2020b). *Effective Go - The Go Programming Language.* `https://golang.org/doc/effective_go.html#panic`.

Google, Inc. (2020c). *The Go Programming Language.* https://golang.org/ref/spec.

Gouy, I. (2020a). *The Computer Language Benchmarks Game.* https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

Gouy, I. (2020b). *How programs are measured | Computer Language Benchmarks Game.* https://benchmarksgame-team.pages.debian.net/benchmarksgame/how-programs-are-measured.html.

Hannikainen, J. (2020). *Purkka.* https://github.com/jgke/purkka.

Hazel, P. (2020). *PCRE - Perl Compatible Regular Expressions.* https://www.pcre.org/.

Heathcote, J. (2014). *C Pre-Processor Magic.* http://jhnet.co.uk/articles/cpp_magic.

Hoare, T. (2009). *Null References: The Billion Dollar Mistake.* https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

ISO/IEC 1003.1:2012 – IEEE Standard for Information Technology – Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7 (Standard). (2017a). Geneva, CH: International Organization for Standardization.

ISO/IEC 14882:2017 – Information technology – Programming languages – C++ (Standard). (2017b). Geneva, CH: International Organization for Standardization.

ISO/IEC 8652:2012 – Information technology – Programming languages – Ada (Standard). (2012). Geneva, CH: International Organization for Standardization.

ISO/IEC 9899:1990 – Information technology – Programming languages – C (Standard). (1990). Geneva, CH: International Organization for Standardization.

ISO/IEC 9899:1999 – Information technology – Programming languages – C (Standard). (1999). Geneva, CH: International Organization for Standardization.

ISO/IEC 9899:2011 – Information technology – Programming languages – C (Standard). (2011). Geneva, CH: International Organization for Standardization.

ISO/IEC 9899:2018 – Information technology – Programming languages – C (Standard). (2018). Geneva, CH: International Organization for Standardization.

JetBrains. (2020a). *Calling Java from Kotlin - Kotlin Programming Language.* https://kotlinlang.org/docs/reference/java-interop.html#null-safety-and-platform-types.

JetBrains. (2020b). *Kotlin Programming Language.* https://kotlinlang.org/.

Kell, S. (2017). Some Were Meant for C: The Endurance of an Unmanageable Language. Teoksessa *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (s. 229–245). New York, NY, USA: ACM. doi: 10.1145/3133850.3133867

Kernighan, B. W., & Ritchie, D. M. (1978). *The C Programming Language* (ensimmäinen painos). Prentice Hall.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (toinen painos). Prentice Hall.

kiselgra. (2019). *kiselgra/c-mera.* https://github.com/kiselgra/c-mera.

kostya. (2020). *kostya/benchmarks: Some benchmarks of different languages.* https://github.com/kostya/benchmarks/.

Melo, L. T. C., Ribeiro, R. G., de Araújo, M. R., & Pereira, F. M. Q. (2017, joulukuuta). Inference of Static Semantics for Incomplete C Programs. *Proc. ACM Program. Lang.*, *2*(POPL). doi: 10.1145/3158117

Meyerovich, L. A., & Rabkin, A. S. (2013). Empirical Analysis of Programming Language Adoption. Teoksessa *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (s. 1–18). New York, NY, USA: ACM. doi: 10.1145/2509136.2509515

Microsoft Corporation. (2020). *TypeScript - JavaScript that scales.* http://www.typescriptlang.org/.

Nethercote, N., & Seward, J. (2007). Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Teoksessa *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (s. 89–100). New York, NY, USA: ACM. doi: 10.1145/1250734.1250746

OpenMP Application Programming Interface Version 5.0 (Standard). (2018). OpenMP Architecture Review Board.

Pike, R. (2010). *Another Go at Language Design.* https://web.stanford.edu/class/ee380/Abstracts/100428.html.

Pike, R. (2014). *Generating code - The Go Blog.* https://blog.golang.org/generate.

Python Software Foundation. (2020). *python/cpython: The Python programming language.* https://github.com/python/cpython.

Red Hat, Inc. (2019). *Security flaws caused by compiler optimizations.* https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations.

Ritchie, D. (1993). The Development of the C Language. Teoksessa *The Second ACM SIGPLAN Conference on History of Programming Languages* (s. 201–208). New York, NY, USA: ACM. doi: 10.1145/154766.155580

Rust Project Developers. (2018). *Frequently Asked Questions - The Rust Programming Language.* `https://prev.rust-lang.org/en-US/faq.html#why-do-rust-programs-have-larger-binary-sizes-than-C-programs`.

Rust Project Developers. (2020a). *Error Handling - The Rust Programming Language.* `https://doc.rust-lang.org/book/first-edition/error-handling.html`.

Rust Project Developers. (2020b). *Frequently Asked Questions - The Rust Programming Language.* `https://www.rust-lang.org/en-US/faq.html`.

Rust Project Developers. (2020c). *Introduction - The Rust Reference.* `https://doc.rust-lang.org/reference/`.

Rust Project Developers. (2020d). *Macros - The Rust Programming Language.* `https://doc.rust-lang.org/1.2.0/book/macros.html`.

Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, *2*(1), 51-72. doi: 10.1080/10447319009525970

Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A Fast Address Sanity Checker. Teoksessa *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (s. 309–318). Boston, MA: USENIX. Saatavilla `https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany`

Stroustrup, B. (2007). Evolving a Language in and for the Real World: C++ 1991-2006. Teoksessa *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (s. 4-1–4-59). New York, NY, USA: ACM. doi: 10.1145/1238844.1238848

Stroustrup, B. (2014). *C++ Applications.* `http://www.stroustrup.com/applications.html`.

Svedäng, E., & Heller, V. (2020). *carp-lang/Carp: A statically typed lisp, without a GC, for high performance applications.* `https://github.com/carp-lang/Carp`.

TIOBE software BV. (2020). *TIOBE index.* `https://www.tiobe.com/tiobe-index`.

Tremblay, J.-P. (1985). *The Theory and Practice of Compiler Writing.* McGraw-Hill Book Company.

Whitaker, W. A. (1996). History of Programming languages—II. Teoksessa T. J. Bergin Jr. & R. G. Gibson Jr. (toim.), (s. 173–232). New York, NY, USA: ACM. doi: 10.1145/234286.1057816

You, J.-Y. (2020). *rust-lang-nursery/rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries.* https://github.com/rust-lang-nursery/rust-bindgen.

Liite 1. Mittaukset

Ohjelmien selitteet

binarytrees

Ohjelma on moniosainen: ensin luodaan binääripuu ja poistetaan se. Tämän jälkeen luodaan binääripuu, joka poistetaan vasta ohjelman loputtua. Tämän jälkeen luodaan useita binääripuita, joista lasketaan solmujen määrä.

fannkuchredux

Ohjelma laskee jokaiselle joukon $1, ..., n$ permutaatioista tarkistussumman, joka lasketaan seuraavalla kaavalla: permutaatiosta otetaan ensimmäinen luku x. Permutaation ensimmäisen x alkion järjestys muutetaan päinvastaiseksi. Näitä kahta operaatiota toistetaan, kunnes ensimmäinen alkio on 1. Jos permutaatio on pariton, tarkistussumma on kierrosten määrä, muuten tämän negaatio. Tarkistussummat lasketaan yhteen.

fasta

Ohjelmassa luodaan suuri määrä DNA-sekvenssejä joko kopioiden annettua sekvenssiä tai painotetulla satunnaisella valinnalla kahdesta aakkostosta.

knucleotide

Ohjelma lukee *fasta*-ohjelman luoman tiedoston ja laskee siinä esiintyvien nukleotidisarjojen määrän 1, 2, 3, 4, 6, 12 ja 18 nukleotidin pituisille sarjoille.

nbody

Ohjelmassa simuloidaan symplektisellä integroinnilla planeettojen sijaintia.

pidigits

Ohjelmassa lasketaan π:n desimaaleja tietyllä algoritmilla.

regexredux

Ohjelma lukee *fasta*-ohjelman muodostaman tiedoston ja ajaa ennalta määritettyjä säännöllisiä lausekkeita tiedoston sisältöön.

spectralnorm

Ohjelma laskee annetun matriisin spektraalisäteen.

revcomp

Ohjelma lukee *fasta*-ohjelman muodostaman tiedoston ja muodostaa tiedoston DNA-sekvenssien komplementin.

Suoritusaika(s)

binarytrees

Kieli	Mittaukset
C	1.395, 1.539, 1.535, 1.520, 1.527
C++	1.495, 1.536, 1.537, 1.528, 1.524
Rust	1.173, 1.196, 1.196, 1.194, 1.205
Go	13.949, 14.266, 14.716, 14.822, 14.926
Ada	10.434, 10.889, 11.461, 12.631, 12.470
Purkka	1.441, 1.424, 1.417, 1.413, 1.440

fannkuchredux

Kieli	Mittaukset
C	12.365, 12.924, 13.453, 13.399, 13.461
C++	11.023, 11.534, 11.588, 11.444, 11.511
Rust	10.982, 11.311, 11.339, 11.293, 11.299
Go	10.769, 11.142, 11.442, 12.184, 11.885
Ada	23.972, 25.183, 25.506, 25.518, 25.255
Purkka	7.426, 7.533, 7.573, 7.555, 7.565

fasta

Kieli	Mittaukset
C	0.815, 0.770, 0.800, 0.821, 0.801
C++	0.889, 0.863, 0.876, 0.887, 0.881
Rust	0.954, 0.938, 0.939, 0.956, 0.944
Go	1.051, 1.045, 1.018, 1.027, 1.045
Ada	11.281, 8.604, 7.894, 8.559, 7.926
Purkka	0.789, 0.796, 0.805, 0.808, 0.810

knucleotide

Kieli	Mittaukset
C	3.701, 3.937, 4.010, 4.046, 3.830
C++	1.920, 2.063, 1.946, 1.923, 1.980
Rust	3.054, 3.031, 2.894, 3.019, 3.060
Go	12.467, 7.278, 7.089, 7.298, 7.069
Ada	18.550, 17.729, 17.170, 17.172, 17.469
Purkka	4.210, 4.387, 4.486, 4.319, 4.299

nbody

Kieli	Mittaukset
C	2.862, 3.038, 2.703, 3.134, 3.039
C++	3.033, 3.049, 2.623, 2.751, 2.632
Rust	1.980, 2.096, 2.104, 2.113, 2.103
Go	4.506, 4.025, 4.047, 4.061, 4.071
Ada	14.593, 13.937, 14.659, 14.611, 15.869
Purkka	3.105, 3.278, 2.984, 3.276, 2.982

pidigits

Kieli	Mittaukset
C	0.735, 0.731, 0.733, 0.729, 0.635
C++	0.725, 0.673, 0.679, 0.727, 0.680
Rust	0.628, 0.701, 0.719, 0.718, 0.720
Go	0.896, 0.763, 0.845, 0.794, 0.766
Ada	0.745, 0.647, 0.741, 0.742, 0.742
Purkka	0.639, 0.725, 0.702, 0.723, 0.655

regexredux

Kieli	Mittaukset
C	0.912, 0.902, 0.928, 0.896, 0.975
C++	1.206, 1.031, 1.027, 1.019, 1.032
Rust	1.123, 1.267, 1.261, 1.111, 1.120
Go	14.957, 15.317, 15.074, 15.309, 14.803
Ada	3.744, 3.739, 3.778, 3.996, 3.709
Purkka	0.945, 0.892, 0.960, 0.951, 0.900

spectralnorm

Kieli	Mittaukset
C	0.301, 0.316, 0.308, 0.307, 0.314
C++	0.243, 0.245, 0.248, 0.250, 0.256
Rust	0.332, 0.336, 0.336, 0.342, 0.336
Go	0.570, 0.582, 0.561, 0.575, 0.591
Ada	3.231, 3.283, 3.301, 3.321, 3.330
Purkka	0.302, 0.314, 0.325, 0.323, 0.317

revcomp

Kieli	Mittaukset
C	1.033, 0.556, 0.565, 0.556, 0.561
C++	1.437, 1.176, 1.171, 1.190, 1.210
Rust	1.077, 0.606, 0.574, 0.591, 0.582
Go	2.127, 1.603, 1.680, 1.740, 1.803
Ada	27.073, 12.978, 14.099, 13.592, 13.613
Purkka	0.935, 0.546, 0.549, 0.534, 0.547

Muistinkäyttö(KB)

Muistinkäyttö on mitattu 200 millisekunnin välein `libgtop2`-kirjastolla ja kunkin suorituksen muistinkäytöksi on merkitty suurin mitattu muistikäyttö.

binarytrees

Kieli	Mittaukset
C	177364, 177452, 177508, 177440, 177484
C++	113040, 113100, 113200, 113100, 113280
Rust	144964, 83448, 112028, 129812, 153144
Go	405024, 433900, 419308, 451448, 434564
Ada	527856, 527776, 527876, 527776, 527692
Purkka	177100, 177136, 177140, 177120, 177148

fannkuchredux

Kieli	Mittaukset
C	888, 884, 884, 944, 948
C++	1960, 1820, 1896, 1960, 1968
Rust	1000, 1000, 928, 936, 932
Go	10428, 10656, 10172, 11188, 10688
Ada	4352, 4440, 4392, 4412, 4408
Purkka	940, 876, 808, 872, 808

fasta

Kieli	Mittaukset
C	2904, 2956, 2764, 2852, 2916
C++	2200, 2124, 1752, 1824, 1816
Rust	3140, 3000, 3076, 3080, 3076
Go	11292, 11076, 11056, 10776, 10780
Ada	1824, 1820, 1764, 1824, 1880
Purkka	2836, 2756, 2756, 2824, 2852

knucleotide

Kieli	Mittaukset
C	129912, 130004, 130032, 129952, 129924
C++	156288, 156000, 156268, 156112, 156036
Rust	133488, 133568, 133476, 133648, 133528
Go	143264, 142600, 142832, 143000, 142640
Ada	258288, 258236, 258500, 258240, 258440
Purkka	130372, 129948, 130076, 129992, 130068

nbody

Kieli	Mittaukset
C	1100, 1044, 1032, 1040, 1044
C++	1788, 1652, 1652, 1724, 1788
Rust	800, 800, 872, 860, 800
Go	1448, 1448, 1444, 1448, 1444
Ada	1960, 1828, 1964, 1760, 1828
Purkka	708, 776, 776, 708, 840

pidigits

Kieli	Mittaukset
C	2372, 2456, 2428, 2440, 2564
C++	4360, 4364, 4344, 4364, 4348
Rust	3228, 2748, 2940, 2828, 2756
Go	8748, 8628, 8784, 8628, 8708
Ada	4144, 4548, 4356, 4324, 4196
Purkka	2708, 2604, 2708, 2604, 2624

regexredux

Kieli	Mittaukset
C	148780, 148684, 149040, 148628, 148680
C++	275616, 275868, 275624, 275756, 275752
Rust	143260, 134692, 134896, 144196, 143716
Go	438800, 435636, 439108, 435628, 438820
Ada	154284, 154400, 150948, 150744, 150772
Purkka	148552, 148588, 148824, 148780, 148752

spectralnorm

Kieli	Mittaukset
C	1264, 2252, 1068, 1200, 2472
C++	1196, 1196, 1196, 1068, 1264
Rust	940, 868, 936, 872, 864
Go	1384, 1324, 1896, 1792, 1320
Ada	4248, 4144, 4364, 4184, 4248
Purkka	1196, 1068, 1200, 1068, 1072

revcomp

Kieli	Mittaukset
C	994536, 609924, 599112, 607524, 602784
C++	980804, 826356, 838760, 821688, 980752
Rust	994956, 615996, 638436, 603876, 635056
Go	824688, 775320, 824336, 799820, 775264
Ada	491652, 491672, 491668, 491728, 491700
Purkka	994564, 629192, 616500, 645284, 615496

Lähdekoodin koko (B)

Lähdekoodin koko on mitattu poistamalla lähdekoodista kaikki kommentit sekä tyhjämerkit, jotta lähdekoodin asettelu ei vaikuta mittauksiin. Tämän jälkeen ohjelma ajetaan `gzip`-pakkausohjelman läpi ja mitataan pakatun lähdekoodin koko.

binarytrees

Kieli	Koko
C	753
C++	728
Rust	664
Go	585
Ada	1154
Purkka	698

fannkuchredux

Kieli	Koko
C	1377
C++	951
Rust	1055
Go	797
Ada	1915
Purkka	858

fasta

Kieli	Koko
C	2082
C++	2441
Rust	1764
Go	1295
Ada	1267
Purkka	1822

knucleotide

Kieli	Koko
C	1399
C++	1489
Rust	1389
Go	1396
Ada	4470
Purkka	1385

nbody

Kieli	Koko
C	1370
C++	1651
Rust	1673
Go	1104
Ada	1600
Purkka	1289

pidigits

Kieli	Koko
C	414
C++	459
Rust	1180
Go	552
Ada	1038
Purkka	388

regexredux

Kieli	Koko
C	1300
C++	2629
Rust	758
Go	757
Ada	3295
Purkka	1272

spectralnorm

Kieli	Koko
C	1025
C++	932
Rust	1018
Go	484
Ada	1539
Purkka	1020

revcomp

Kieli	Koko
C	1301
C++	2032
Rust	1702
Go	554
Ada	844
Purkka	1314

Liite 2. Purkka-kielen tyypit

Tässä liitteessä on taulukko Purkka-kielen tyypeistä sekä Purkka-tyyppejä vastaavista C-tyypeistä. Taulukko ei sisällä keskenään identtisiä C-tyyppejä, kuten `int` ja `signed int`.

Purkka-tyyppi	Purkka-tyyppiä vastaava C-tyyppi tai sen hahmotelma
`void`	`void`
`char`	`char`
`i8`, `i16`, `i32`, `i64`	`int8_t`, `int16_t`, `int32_t`, `int64_t`
`u8`, `u16`, `u32`, `u64`	`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
`float`	`float`
`double`	`double`
`cbyte`, `cshort`, `cint`, `clong`, `clonglong`	`signed char`, `short`, `int`, `long`, `long long`
`cubyte`, `cushort`, `cuint`, `culong`, `culonglong`	`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`
`&T, &?T`	`T *`
`[T]`	$T_{\text{declaration specifiers}}$`[]`$T_{\text{direct abstract declarator}}$
`[T;expr]`	$T_{\text{declaration specifiers}}$`[expr]`$T_{\text{direct abstract declarator}}$
`fun t: (T ..) -> R`	$R_{\text{declaration specifiers}}$ `t(T ..)`$R_{\text{direct abstract declarator}}$
`fun (T ..) -> R`	$R_{\text{declaration specifiers}}$ `(*)(T ..)`$R_{\text{direct abstract declarator}}$
`struct { k: T ..}`	`struct { T k ..}`
`enum { A [= expr] ..}`	`enum { A [= expr] ..}`
`union { k: T ..}`	`union { T k ..}`
`(T1 ..)`	`struct { T1 e1; ..}`
`enum { T(T1 ..) .. }`	`struct { union {T1 t1 ..} v; enum {T ..} e;}`
GCC-laajennokset	
`attribute(T, ...))`	`T __attribute__((...))`