Henri Holm

C#-pohjainen kääntäjän front-end

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

3.5.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Henri Holm C#-pohjainen kääntäjän front-end 56 sivua + 1 liite 3.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Lehtori Keijo Länsikunnas

Insinöörityön tarkoituksena oli tutkia ja vastata kysymyksiin, mitä ovat ohjelmointikielten kääntäjät ja tulkit, sekä tutkia prosessi, jonka ohjelmistojen lähdekoodit käyvät lävitse kääntäjien tehdessä niistä valitun suoritusalustan ymmärtämää binäärimuotoista käskysarjaa.

Työ jakautui kahteen pääosaan: raporttiosaan, jossa lähestyttiin kääntäjien toimintaa teoreettisemmalla tasolla ja projektiosioon, jossa toteutettiin, ja selvitettiin C#:lla Windows 10 pro -käyttöjärjestelmälle kirjoitetun kääntäjän front-endin toimintaa.

Työn haasteisiin kuului kääntäjien teorian ja toiminnan selvitys sekä erityisesti luetun teorian muunnos toimivaksi kääntäjän front-endiksi. Työssä perehdyttiin ensin teoriaan, mistä edettiin raporttiosan kirjoittamisen jälkeen ohjelmointityöhön. Ohjelmointi tehtiin yritys ja erehdys -metodilla lähdemateriaalin teoreettisuuden vuoksi.

Työn tuloksena oli laajennuskelpoinen kääntäjän front-end, jonka prosessoimaa ohjelmointikieltä on mahdollista laajentaa ja johon on myöhemmin mahdollista toteuttaa nykyisen C#-toteutuksen sijaan itsenäinen back-end optimointivaiheineen. Lisäksi työ auttoi ymmärtämään paremmin kääntäjien toiminnan ja käännösprosessin perusperiaatteet.

Tehdystä front-endistä voitiin myös päätellä, että se vaati paljon käsityötä ja voi johtaa raskaisiin ehtorakenteisiin, jotka ovat ohjelmiston ylläpidon ja laajennettavuuden kannalta epätoivottuja. Lisäksi havaittiin, ettei raporttiosassa esitelty kääntäjäarkkitehtuuri ole ainoa oikea tapa tehdä asioita, sillä tehdystä front-endistä on jätetty joitain raportissa esitettyjä vaiheita pois.

Avainsanat	Kääntäjä, tulkki, assembler

Author(s) Title Number of Pages Date	Henri Holm C#-based compiler front-end 56 pages + 1 appendice 3 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Keijo Länsikunnas, Senior Lecturer

The aim of this thesis was to examine and answer the question what are compilers and interpreters as well as study the process which software source codes go through when they are translated from text to a runnable binary for the selected platform.

Thesis project was divided in two parts: report section where compilers and their functions were studied on a more theoretical level and project section where a compiler front-end was written in C# for the Windows 10 operating system and where it was studied how this front-end worked.

Challenges met during the thesis project included investigating both theory and operating models and especially converting theory into practice. First, theory was studied, after which the report was written, and finally, programming was conducted. The rather theoretical source material caused programming to be done in trial and error method.

Results of this work were a compiler front-end that can be later expanded with fully independent back-end phases such as the optimization phase. It also processes a programming language that is expandable. Other results were a better understanding of compilers and the compilation process.

Conclusions from the implemented front-end were that it requires a substantial amount of manual labour and can lead into large conditional statements that are unwanted from the software engineering point of view. It was also noticed that the compiler architecture studied in the report section was not the only way to implement phases as the implemented front-end did not include all the phases introduced in the report.

Keywords	compiler, interpreter, assembler

Sisällys

Lyhenteet

1 Johdanto 1

2 Ohjelmointikielten kääntäjät 2

2.1 Kääntäjien perusteet 2
2.2 Kääntäjien historia 2
2.3 Kääntäjätyypit 5

3 Kääntäjän front-endin vaiheet 11

3.1 Syötteen lukeminen 11
3.2 Välitason koodigenerointi 16

4 Kääntäjän back-endin vaiheet 22

4.1 Suoritusaikainen ympäristö 22
4.2 Koodigenerointi 27
4.3 Kääntäjäoptimoinnit 30

5 Kääntäjäni 32

5.1 Perustiedot 32
5.2 Kohdealusta 32
5.3 Toiminnan kuvaus 33
5.4 Huomiot kääntäjän lähdekoodista 40
5.5 Tiedossa olevat virheet 46

6 Työprosessi ja johtopäätökset 46

6.1 Lähtökohdat 46
6.2 Suunnittelu 47
6.3 Toteutus 49
6.4 Insinöörityön toteutuksen jatkokehitys ja tulevaisuus 52
6.5 Johtopäätökset 53

Lähteet 57

Liitteet

Lyhenteet

CISC	Complex instruction set computing. Käskykanta, jossa yksi käsky voi suorittaa monta alemman tason toimintoa.
RISC	Reduced instruction set computing. Käskykanta, jossa yksittäiset käskyt on pyritty pitämään mahdollisimman yksinkertaisina ja tehokkaina suorittaa.
JIT	Just-in-time. Kääntäjätekniikka, joka pyrkii yhdistämään konekielelle kääntämisen ja tulkkauksen parhaat puolet.
AOT	Ahead-of-time. Kääntäjätekniikka, jossa korkean tason ohjelmakoodin käännös suoritettavaksi ohjelmakoodiksi tehdään ennen ohjelmasuoritusta.
EDSAC	Electronic Delay Storage Automatic Calculator. Maurice Wilkesin johtaman ryhmän rakentama tietokonemalli 1940-luvun lopulla.
SOAP	Symbolic Optimizer and Assembly Program. IBM:n vuonna 1953 julkaiseman IBM 650 -tietokoneen mukana tullut assembler.
COBOL	Common business-oriented language. Julkaistiin vuonna 1959 ja on täten eräs vanhimmista yhä käytössä olevista korkean tason ohjelmointikielistä.
LISP	List processor. Maailman toiseksi vanhin yhä käytössä oleva korkean tason ohjelmointikieli.

ALGOL	Algorithmic language. Eräs vaikutusvaltaisimmista varhaisista korkean tason ohjelmointikielistä, jonka mukaan on nimetty kokonainen ohjelmointikielten perhe.
AVX	Advanced Vector Extension. Intelin marraskuussa 2008 esittelemä käskykantalaajennus x86-arkkitehtuurille, joka sittemmin implementoitiin ensi kerran Intelin Sandy Bridge-arkkitehtuuriin vuoden 2011 alussa ja AMD:n Bulldozer-arkkitehtuuriin loppuvuodesta 2011.
IA64	Intel Architecture 64 bit. Intelin, vähintäänkin kaupallisesti, epäonnistunut 64-bittinen EPIC-käskykannan arkkitehtuuri, jonka oli tarkoitus korvata iäkäs x86-käskykanta.
VLIW	Very long instruction word. Käskykanta, jossa pyritään käskyjen suorittamiseen rinnakkain.
EPIC	Explicitly parallel instruction computing. VLIW-käskykantaan perustuva käskyjen rinnakkaissuoritukseen nojaava käskykanta, jossa yhtenä pyrkimyksenä on siirtää käskyjen rinnakkaisuuden ajoittaminen suorittimelta kääntäjille.
MASM	Microsoft Macro Assembler. Microsoftin Visual Studion mukana tuleva makroassembler.
BASIC	Beginner's all-purpose symbolic instruction code. 1960-luvulla opetustarkoitukseen kehitetty korkean tason ohjelmointikieli.
LL	Left to right Leftmost derivation. Kielioppi ja ylhäältä alas jäsennystekniikka, jossa jäsennyspuu rakennetaan aloittaen lauseen vasemmanpuoleisimmasta merkistä, josta tulee puun juuri.
LR	Left to right Rightmost derivation. Kielioppi ja alhaalta ylös jäsennystekniikka, jossa jäsennyspuu rakennetaan aloittaen lauseen oikeimmanpuoleisesti merkistä ja edeten silmujen kautta juureen.

CIL	Common Intermediate Language. Microsoftin .NET-ympäristön ja Monon käyttämä Javan tavukoodia vastaava välitason kieli, jota virtuaalikone kääntää käytössä olevan alustan konekieleksi.
GCC	GNU Compiler Collection. Alkujaan pelkästään C-kieltä varten luotu avoimen lähdekoodin kääntäjä, jota on sittemmin laajennettu tukemaan useampaa ohjelmointikieltä, muun muassa Fortrania ja Go:ta.
DAG	Directed acyclic graph. Suomeksi suunnattu syklitön verkko. Rakenteiden mallintamiseen käytetty esitys, jossa rakennetta läpi käytäessä kerran vierailtuun pisteeseen ei enää palata.
AVX2	Advanced Vector Extensions 2. Intelin luoma laajennus AVX-käskykantaan. Julkaistiin Haswell-arkkitehtuurin mukana vuonna 2013.
UTF	Unicode Transformation Format. Unicode.standardin merkkien koodaukseen käytetty koodaustapa. Standardi tukee UTF--, UTF-- ja UTF-32 koodaustapaa, jossa luku viittaa merkin maksimissaan käyttämään bittimäärään.

1 Johdanto

Insinöörityön tarkoituksena on perehtyä ohjelmointikielten kääntäjiin ja niiden toimintaan. Tavoitteena on selvittää, miten ihmisen luettavissa oleva teksti päätyy suoritinten ymmärtämään muotoon.

Kääntäjät ovat linkki suorittimien ja ihmisten välillä. Ihmiset puhuvat kieliä, joissa virkkeillä, lauseilla ja sanoilla voi olla monia merkityksiä asiayhteydestä ja ruumiinkielestä riippuen. Suorittimet eivät tätä kieltä ymmärrä, vaan ne noudattelevat (yleensä) ykkösiä ja nollia eli binäärejä, joilla määrätään sähkövirran kulkua transistorin läpi. Ihmisille tämä kieli on hankala ja ohjelmistokehityksen näkökulmasta työläs ja virhealtis. Ratkaisuna on siis tehdä ohjelma, joka osaa kääntää ihmisten paremmin ymmärtämän tekstin suoritinten ymmärtämäksi binäärimuotoiseksi komentosarjaksi. Näitä ohjelmia kutsutaan kääntäjiksi.

Työn projektiosan tavoitteena on työn rajoitteiden vuoksi kokonaisen kääntäjän sijaan kirjoittaa kääntäjän front-end. Front-end kääntäjissä tarkoittaa sitä käännösprosessin osaa, jossa luettua lähdekoodia käsitellään alustariippumattomalla tasolla. Työssä tehtävän kääntäjän front-endin on tarkoitus käsitellä VIC-20-tietokoneen BASICista (Beginner’s all-purpose symbolic instruction code) vaikutteita ottaneelle kielelle. Kääntäjän front-endin toimintaympäristö on x86-64-arkkitehtuurin alustalla toimiva Windows 10 pro -käyttöjärjestelmä.

Vaikka projektiosa jää pelkän front-endin tasolle, työn raporttiosassa on myös selvitystä siitä, mitä kääntäjien back-endissä tapahtuu. Tutkitussa mallissa kääntäjän back-end sisältää optimointi- ja koodigenerointivaiheet.

Työn motiivina on oma mielenkiinto aihetta kohtaan eikä sillä ole ulkopuolista tilaajaa. Ohjelmistoala toimii käytännössä täysin kääntäjien varassa, mutta niiden toimintaan ei tällä koulutustasolla ole tutustuttu laisinkaan. Siksi katsoin oman ymmärrykseni ja tietämykseni kasvattamiseksi aiheelliseksi tutkia työkalua, jota päivittäin töissäni käytän.

2 Ohjelmointikielten kääntäjät

2.1 Kääntäjien perusteet

Kääntäjien ja tulkkien tehtävä ohjelmointikielissä on sama kuin ihmiskielissä: kääntää viestin lähteenä oleva kieli vastaanottajan ymmärtämään muotoon. Koneiden tapauksissa tavoitteena on muuttaa ohjelmoijien helposti tulkattavissa oleva, usein tekstimuotoinen, komentosarja koneiden ymmärtämään binäärimuotoon.

Muita kääntäjiltä ja tulkeilta odotettuja toimenpiteitä ovat muun muassa semanttinen analyysi, välitason koodigenerointi ja optimoinnit. Kuten ihmisten puhumissa kielissä, ei ohjelmointikielissäkään ole usein suotavaa kääntää lähdekielen sanomaa suoraan sanasta sanaan vastaanottajalle, vaan viestistä pyritään optimoimaan irti viestin ydintarkoitus. Ihmiskielissä tämä voi tarkoittaa fraasin muuntamista viestin vastaanottajan ymmärtämään muotoon, kun vastaavasti ohjelmointikielissä se voi merkitä tuotettavan ohjelman koon, muistikäytön tai tehonkäytön minimointia ja suoritusnopeuden parannusta. (1.)

2.2 Kääntäjien historia

Ensimmäisten elektronisten tietokoneiden ilmestyttyä 1940-luvulla ei kääntäjiä ollut vielä olemassa, vaan koneita ohjelmoitiin suoraan niiden ymmärtämillä ykkösillä ja nollilla. Ohjelmointitapa oli altis virheille ja hidas. Aikaansaadut ohjelmakoodit olivat vaikeasti ymmärrettävissä, mikä osaltaan vaikeutti virheenkorjausta, ohjelmien muuntamista sekä siirtoa alustalta toiselle. (1)

Vuonna 1949 valmistui brittiläinen tietokone EDSAC (Electronic Delay Storage Automatic Calculator), joka ensimmäisenä tietokoneena maailmassa vastaanotti komentoja binäärien ohessa myös yksikirjaimisina muistikkaina. Muistikkaiden idea on piilottaa pitkät ja monimutkaiset binäärikomennot helpommin muistettavissa olevien muistisanojen taakse. Esimerkiksi x86/IA32-arkkitehtuurin komento 10110000 01100001 taipuu muotoon

```
;10110=MOV
;000=AL
;01100001=61h, h=heksadesimaali
;61h=97
MOV AL, 61h
```

Vuonna 1953 julkaistulle IBM 650 -tietokoneelle julkaistiin vuonna 1955 SOAP (Symbolic Optimizer and Assembly Program), jota voidaan pitää nykyisten assemblereiden esikuvana. Esimerkiksi siinä missä konekielellä oli aiemmin täytynyt aina käyttää absoluuttisia osoitteita, osasi SOAP antaa muistiosoitteelle symbolin, jota voitiin sitten myöhemmin ohjelmassa käyttää uudestaan. (2; 3; 4; 5.)

Ensimmäiset kääntäjät

1950-luvulla tietokoneiden ja ohjelmointikielten kehitys oli nopeaa. Pian ensimmäisten assemblereiden vanavedessä ilmestyivät ensimmäiset korkeamman tason ohjelmointikielet, jotka vaativat monimutkaisempia käännöstoimenpiteitä ollakseen koneiden ymmärrettävissä.

Ensimmäisiä kääntäjiä olivat A-0-systemille, Autocodelle ja Fortranille (Formula translator) tehdyt kääntäjät vuosien 1951 ja 1957 välillä. Näistä Fortranille tehty kääntäjä on yleisesti mainittu ensimmäiseksi oikeaksi kääntäjäksi.

Ensimmäisten kääntäjien ongelmana oli laadukkaan konekielisen koodin generointi. Aikakauden koneet olivat hitaita, joten sille kirjoitetun koodin tuli olla äärimmäisen optimoitua koneiden suorituskyvyn maksimoimiseksi. Tähän ongelmaan Fortranin kehittäjät vastasivat tekemällä maailman ensimmäisen optimoivan kääntäjän IBM 704 -tietokoneelle. Kääntäjän vallankumouksellisiin ominaisuuksiin kuului mm. kyky erotella laskuoperaattoreiden arvojärjestys, poistaa tarpeettomat muuttujat ja toistorakenteiden optimointi. (6.)

Muita mainitsemisen arvoisia kieliä aikakaudelta ovat LISP, nykyisin Lisp, (List processor), COBOL (Common business-oriented language) ja ALGOL (Algorithmic language). Lisp loi perustan nykyisille funktionaalisille ohjelmointikielille ja esitteli muun muassa. ehtolauseet sekä rekursion. COBOL vuorostaan oli ensimmäisiä kieliä, joiden intressit olivat akateemisen maailman sijaan talouden maailmassa. Kielen tavoite oli saada aikaan säästöjä ohjelmistotuotantoon ja ohjelmistojen kääntämisen alustalta toiselle avoimuudella ja yhteensopivuudella eri alustojen välillä. Listan kolmas kieli, ALGOL, oli

vuorostaan yhdysvaltalais-eurooppalaisen yhteistyön tuotos. Siinä missä tietokoneita oli suhteellisen laajalti kaupallisesti saatavilla Yhdysvalloissa 1950-luvun lopussa, ne olivat Euroopassa vielä harvinaisia. Euroopassa rakennetuille tietokoneille kaivattiin jotain yhteistä kieltä, ja niin sai syntynsä ALGOL. ALGOLin edistyksellisiin ominaisuuksiin kuului mm. dynaaminen muistinhallinta ja syntaksin ohjaama käännösvaihe. Vaikka kieli ei jäänyt eloon kovin pitkäksi aikaa, se vaikutti kuitenkin moniin myöhäisempiin kieliin ja synnytti kokonaisen kieliperheen. (7; 8; 9; 10.)

Nykyajan kääntäjät

Ohjelmointikielten ja tietokoneiden kehittyminen on johtanut yhä suurempiin ja monimutkaisempiin kääntäjiin ja tulkkeihin. Kehityssuunta on mennyt alati korkeamman tason ohjelmointikieliin, joiden pyrkimys on nopeuttaa itse ohjelmistojen kirjoittamista ja minimoida ohjelmoijien tekemien virheiden määrää muun muassa automatisoimalla muistinhallinta, raja-arvojen tarkistuksilla ja piilottamalla osoittimet. Kääntäjien on nykyään myös osattava ottaa huomioon ohjelman käskyjen rinnakkaisajo järjestelmissä, joissa on monta suoritinydintä tai käskykannat, jotka suunnittelunsa puolesta nojaavat vahvasti parallelismiin. Esimerkkinä rinnakkaisuuteen perustuvasta alustasta on totaalisen epäonnistunut IA-64 (Intel Architecture 64 Bit), jolla Intel pyrki korvaamaan vanhan ja ajastaan jälkeenjääneeksi katsotun x86-käskykannan. IA-64 on lähteestä riippuen VLIW (Very Long Instruction Word)- tai sen siihen perustuvan EPIC (explicitly parallel instruction-set computing) -käskykannan arkkitehtuuri, jota mainostettiin suurella rahalla tulevaisuuden tärkeimmäksi prosessoriarkkitehtuuriksi, mutta joka reilujen myöhästelyiden jälkeen paljastui täydelliseksi pettymykseksi suorituskyvyn osalta. Osa heikosta suorituskyvystä selittyy sillä, että arkkitehtuuri ”suunniteltiin kääntäjille, joita oli mahdoton tuottaa”. (1; 11; 12; 13.)

2.3 Kääntäjätyypit

JIT-kääntäjät

JIT (Just-in-time) on 1960-luvulla Lispin mukana esitelty käännöstapa, joka pyrkii yhdistämään konekielelle kääntämisen ja tulkkaamisen parhaat puolet. JIT-käännöksessä lähde- tai väliesityskoodin kääntäminen ja optimointi suoritusalustan ymmärtämään muotoon suoritetaan ohjelman ajon aikana. JIT-kääntäjistä tunnetuin lienee Javan virtuaalikoneen kääntäjä, joka pyrkii tekemään Javac-kääntäjän luomasta tavukoodista mahdollisimman tehokasta konekieltä. (14.)

JIT-kääntäjä yhdistämään konekielelle kääntämisen ja tulkkaamisen parhaat puolet. Ideaalitilanteessa JIT-kääntäjän muodostama konekieli olisi yhtä nopeaa kuin sellaisen kääntäjän, joka tuottaa konekielen ennen ohjelman ajoa, ja veisi yhtä vähän tilaa kuin tulkattu kieli. Käytännössä tämä ei tietenkään onnistu esimerkiksi siitä syystä, että kääntämiseen menevä aika ei saisi näkyä käyttäjälle. Tämä vuorostaan tarkoittaa sitä, ettei JIT-kääntäjällä ole yhtä paljon aikaa analysoida ja optimoida koodia kuin ennen ajoa käännöksensä suorittavalla kääntäjällä. (14.)

JIT-kääntäjiin liittyy varsin tiiviisti tavukoodi. Tavukoodi on virtuaalikoneeksi kutsutun teoreettisen suorittimen natiivia konekieltä. Koska tavukoodi on reaalimaailman alustoista riippumatonta, voidaan samaa tavukoodia suorittaa käyttöjärjestelmästä ja käskykannoista riippumatta ideaalitilanteessa sellaisenaan. Ainoa vaatimus on, että tavukoodia suorittava virtuaalikone toimii valitulla alustalla. Tavukoodiksi kääntyviä ja virtuaalikoneen suorittamia tunnettuja kieliä ovat muun muassa. Java ja C#.

JIT-kääntäjien toimintatapoihin vaikutti suuresti Gilbert Joseph Hansenin vuonna 1974 julkaistu työ Adaptive systems for the dynamic run-time optimization of programs. Työssään hän otti kantaa kolmeen tärkeään kysymykseen:

1) Mitä koodia tulisi optimoida?

2) Milloin koodia tulisi optimoida?

3) Miten koodia tulisi optimoida?

Työssään Hansen toteutti yksinkertaisen laskurin, joka tarkkaili, kuinka monta kertaa kussakin koodiblokissa vierailtiin. Laskurin ylitettyä tietyn raja-arvon yleni siihen liitetty koodiblokki optimoitavien koodiblokkien joukkoon. Lopulta optimoitavaksi päätyneet koodiblokit kävivät lävitse perinteisen joukon alustariippuvaisia ja -riippumattomia optimointiprosesseja. Optimoinnit kohdennettiin vielä siten, että mitä useammin optimointivaihe vieraili samassa koodiblokissa, sitä paremmin kyseinen blokki optimoitiin. Vastaavaa mallia on sittemmin käytettyä muun muassa. Javan JIT-kääntäjissä, mutta ei .NETin implementaatiossa, jossa suoritettava tavukoodi käännetään kerran ajon aikana ja tuotettu käännös sijoitetaan muistiin myöhempää käyttöä varten. (14; 15; 16; 17.)

Vaikka JIT-kääntäjien tuottama konekieli ei pääasiassa ole yhtä hyvää kuin käännöksensä ennen ohjelmien ajoa suorittavien kääntäjien, on niiden kuitenkin mahdollista tuottaa optimointeja, joihin perinteiset kääntäjät eivät kykene.

Esimerkiksi perinteinen C-kääntäjä joutuu kääntämään lähdekoodinsa kehittäjän valitsemalle prosessorikannalle, joka voi mahdollisesti olla vuosikymmenen vanha ja on siten estynyt käyttämään uusimpia mahdollisia optimointikeinoja, kuten vaikkapa uudempien sukupolvien mukanaan tuomia käskykantoja. Toisin kuin AOT (Ahead-of-time) tyyppisesti käännetty sovellus, ei JIT-käännetyn sovelluksen kehittäjän tarvitse ottaa kantaa suoritusympäristön olosuhteisiin, minkä seurauksena esimerkiksi x86-alustalle C-koodia kääntävä kääntäjä voi joutua suorittamaan käännöksen sillä periaatteella, että sen täytyy tukea Core 2 Duo aikakauden suorittimia eikä se voi siten käyttää optimoinneissaan AVX (Advanced Vector Extensions) käskykantaa. Sen sijaan tavukoodia kääntävä JIT-kääntäjä voi todeta suorittavansa käännöstään tällä hetkellä Sandy Bridge–sarjan suorittimelle ja käyttää edellä mainittua käskykantaa ongelmitta. Tämä johtaa siihen, että tietyissä tilanteissa Javalla kirjoitettu ohjelma on nopeampaa kuin C:llä tai C++:lla kirjoitettu, vaikka keskimäärin jääkin suorituskyvyssä toiseksi. Toisaalta täytyy ottaa myös huomioon ajon aikana tapahtuneeseen käännökseen kulunut aika ja sen vaatimat resurssit, mikä voi johtaa siihen, että saavutettu etu katoaa. (18; 19.)

Muita JIT-kääntäjien huonoja puolia perinteisiin kääntäjiin nähden ovat muun muassa kääntäjän tuottaman konekielen ennustettavuuden hankaluus ja ohjelmien hitaampi käynnistysaika. Tuotetun konekielen ennustettavuutta vaikeuttaa se, ettei konekieleksi kääntyviä osuuksia voida tarkkaan määritellä ennen ohjelman suoritusta. Vaikea ennustettavuus vuorostaan aiheuttaa sen, etteivät JIT-kääntäjään tukeutuvat ratkaisut sovellu erityisen hyvin reaaliaikajärjestelmiin, joissa ennustettavuus on tärkeässä ase-

massa. Hitaat käynnistysajat vuorostaan ovat suoraan seurausta, ainakin Oraclen Java-toteutuksessa, siitä että virtuaalikoneen pitää ennen ohjelman suorituksen aloittamista ladata tavukoodit muistiinsa ja suorittaa niille käännöstoimenpiteitä. (19)

Perinteiset kääntäjät

Perinteiset kääntäjät ovat terminä mitä epämääräisin, mutta tässä yhteydessä niillä tarkoitetaan sitä kääntäjäryhmää, joka suorittaa käännöksensä lähdekoodista valitun alustan konekielelle ennen ohjelman suoritusta. Vaikka koneiden suorituskyky on aikojen saatossa parantunut, ja tulkkien optimoinnit parantuneet suurin harppauksin, on tämä edelleenkin de facto tapa tehdä alustaoptimoitua ohjelmistoa, jossa suorituskyvyllä on suuri rooli.

Ensimmäiset korkeamman tason ohjelmointikielten kääntäjät ilmestyivät 1950-luvun aikana. Ensimmäiset niistä tosin muistuttivat toiminnaltaan enemmän linkittäjiä kuin nykyaikaisia kääntäjiä. Yhtä kaikki, yhtenä näiden kääntäjien suurimmista ongelmista oli nopeus. Vaikka korkeamman tason kielten edut ovat ohjelmistotuotannon näkökulmasta kiistattomat, olivat aikakauden koneet suorituskyvyltään rajoittuneita. Fortranin ensimmäistä kääntäjää kehittäessään kehittäjät ajattelivat jo nopeuden puoliintumisen käsinkirjoitettuun assembly-koodiin verrattuna johtavan projektin epäonnistumiseen. Tilannetta voi verrata nykyaikaan, jossa käsin kirjoitettu assembly-koodi on harvinaisuus ja ohjelmatkin ovat alkaneet siirtyä paikallisesti suoritettavista natiiviohjelmista voimakkaasti JavaScriptin varaan rakennettujen verkkopalveluiden puolelle. Parhaatkin JavaScript-tulkit ovat edelleen useita kertoja hitaampia kuin perinteisten kääntäjien tuottama konekieli. (6; 20.)

Tulkkeihin ja JIT-kääntäjiin nähden perinteisten kääntäjien suurin etu on niiden tuottaman ohjelmiston nopeus. Koska käännös tapahtuu ennen ohjelman suoritusta, on kääntäjällä mahdollisuus tutkia ja optimoida koodia enemmän kuin tulkeilla ja JIT-kääntäjillä. Toisaalta käännökset ovat lopullisia, ja toisin kuin JIT-kääntäjät ja tulkit, eivät perinteiset kääntäjät kykene tekemään ohjelmalle enää mitään sen suorituksen aikana. Tämä johtaa usein siihen, etteivät perinteiset kääntäjät kykene käyttämään kaikkia mahdollisia optimointikeinoja, kuten vaikkapa uusimpia ja nopeimpia käskyjä, konekieltä generoidessaan. Lisäksi kääntäjien tuottama binäärikoodi on keskimäärin suurempaa kuin JIT-kääntäjien tai tulkkien tuottama, ja ohjelmien siirrettävyys on heikompaa, sillä alkuperäinen koodi on käännettävä erikseen jokaiselle alustalle. (21.)

Assemblerit

Assemblerit ovat ensimmäisiä ja yksinkertaisimpia kääntäjiä. Yksinkertaisimmillaan niiden pitää vain sitoa muistisana konekieliseen käskyyn, kuten luvun 2.2 esimerkissä. Toisaalta kuten tulkeista, JIT-kääntäjistä ja perinteisistä kääntäjistä, löytyy assemblereistakin useita eri tyyppejä.

Ensimmäiset assemblerit kirjoitettiin 1940- ja 1950-lukujen taitteessa. Niiden tarkoituksena oli nopeuttaa ja helpottaa ohjelmointia vaihtamalla suorat konekieliset binäärikomennot muistisanoiksi tai muistikkaiksi. Pian assemblereita monimutkaistettiin lisäämällä niihin abstraktiota muun muassa piilottamalla muistiosoitteet symboleihin ja lisäämällä makroja. Lisäksi siinä missä ensimmäiset assemblerit vain latasivat käännetyn ohjelman muistiin, on kehittyneemmistä versioista kyky luoda käännettyjä uudelleensiirrettäviä objektitiedostoja, jotka sisältävät käännetyn ohjelman tiedot. (2.)

Assemblereiden historian tärkein keksintö lienee symbolitaulun keksiminen. Symbolitaulujen idea on sitoa yhteen muistisanat ja niiden konekieliset vastineet ja tarjota kohtalaista abstraktiota menettämättä liikaa puhtaan konekielen suorituskykyä. Symbolitaulut tukevat vain lisäystä ja hakua, mutta eivät poistoa mikä osaltaan vaikuttaa siihen, miten ne on järkevintä toteuttaa. Koska pienissäkin ohjelmissa assembler lukee symbolitaulua satoja kertoja, on lisäysten ja hakujen nopeus tärkeää. Symbolitaulut luodaan liki aina lineaarisena taulukkona, binäärihaun implementoivana järjestettynä taulukkona, nippulajiteltuna linkattuna listana, binääripuuna tai hajautustaulukkona. (3; 2.)

Nykyisellään on muun muassa niin mikro- kuin makroassemblereita, kertaalleen tai kahdesti (one- tai two-pass) lähdekoodinsa läpikäyviä assemblereita, cross-assemblereita ja korkean tason assemblereita. Edellä mainittujen ominaisuuksien yhdistäminen yhteen assembleriin ei ole millään tapaa harvinaista.

Kahdesti lähdekoodinsa läpikäyvät assemblerit lukevat ensimmäisellä kerralla lähdekoodista vain osoitelaput, niiden tyypit ja niihin sijoitetut arvot ja sijoittavat ne sitten symbolitauluun. Ensimmäisellä lukukierroksella tarkistetaan myös käskyjen koko toista kierrosta varten, jolloin käskyt lopulta kootaan.

Kertaalleen lähdekoodinsa lukevat assemblerit nimensä mukaisesti lukevat lähdekoodin vain kerran. Niiden suurin ero kahdesti lähdekoodinsa lukeviin on niiden rajoitettu

kyky luoda objektitiedosto. Siinä missä kahdesti lähdekoodinsa lukevat kykenevät tekemään monikäyttöisen objektitiedoston, ovat kertaalleen lähdekoodinsa lukevat käytännössä rajattuja tekemään absoluuttisia objektitiedostoja. Ne luodaan kirjoittamalla muistissa oleva käskysarja tiedostoon, jonka sisältämät muistiosoitteet ovat kiinteitä, eikä niitä pysty enää käännöksen jälkeen helposti muuttamaan. Tämänkaltaisen objektitiedoston käyttökelpoisuutta voidaan pohtia esimerkiksi perinteisten pöytäkoneiden tai kannettavien tietokoneiden näkökulmasta, joissa suoritetaan samanaikaisesti useita eri ohjelmia ennalta tietämättömissä muistipaikoissa.

Kertaalleen lähdekoodinsa lukevien assemblereiden kyvyttömyys muodostaa monikäyttöinen objektitiedosto on seurasta siitä, että kun ohjelma suoritetaan tietokoneessa, ovat sen käsittelemä tieto ja käskyt vain objektitiedoston binäärejä muistissa eikä käskyjen mahdollisia uudelleensijoitusbittejä ladata erikseen muistiin. Tietokone odottaa löytävänsä suoritettavasta ohjelmasta vain yhden objektitiedoston muistista, joten mahdolliset uudelleensijoitusbitit olisivat vain käsittelykelvotonta binäärimuotoista tietoa oikean tiedon seassa. Komennon uudelleensijoitettavuus vuorostaan selviää vain alkuperäistä komentoa tutkimalla, mihin kertaalleen lähdekoodinsa lukeva assembler ei kykene. Sen on ladattava saamansa komento välittömästi muistiin, sillä toista tarkistelukertaa kyseiselle lähdekoodiriville ei tule. (2.)

Makroassemblerit ovat assemblereita, joihin on implementoitu varattujen sanojen takana tapahtuvia valmiita, usein ylemmän tason ohjelmointikielistä tuttuja, toimintoja. Esimerkiksi MASM (Microsoft Macro Assembler) tukee ylemmistä kielistä tuttuja avainsanoja, kuten if, else if, else, for ja goto, jotka jollain muulla assemblerilla jouduttaisiin mahdollisesti toteuttamaan alemman tason käskyillä. Käännösvaiheessa makroassemblerit tosin korvaavat edeltävät käskyt alemman tason vastikkeillaan ennen kääntöä konekieleksi. Makroassemblerien idea on siis helpottaa ohjelmoijan työtä vaihtamalla monen käskyn sarjat yhdeksi lyhyeksi käskyksi, joka yliajetaan käännösvaiheessa.

Muista edellä mainituista assemblereista voidaan lyhyesti mainita cross-assemblerit, korkean tason assemblerit ja inline-assemblerit. Cross-assemblereilla on kyky tuottaa konekieltä eri alustalle, kuin jossa lähdekoodi on kirjoitettu, mikä mahdollistaa assemblyn kirjoittamisen laitteelle, jonka oma suorituskyky ei riitä käännöksen tekoon. Korkean tason assemblerit sisältävät suhteellisen monimuotoisen abstraktion, kuten tietorakenteet, unionit sekä luokat. Inline-assemblerit vuorostaan mahdollistavat assemblykoodin kirjoittamisen muun muassa C- ja C++-koodin sisälle. Yleensä tällä tavoitellaan

suorituskykylisää ohjelman suorituskriittisissä kohdissa tai suoraa laitteistotason kontrollointia, joka ei korkeamman tason kielillä ole helppoa tai mahdollista. (22; 23 s. 1–7; 24.)

Tulkit

Tulkit ovat kääntäjiä, jotka suorittavat käännöksensä ohjelman suorituksen aikana dynaamisesti suorittaen käskyjä sitä mukaa, kuin ne lähdekoodissa tulevat vastaan. Tämä on selkeä vastakohta perinteisille kääntäjille, jotka eivät ole millään tapaa mukana ohjelman suorituksen aikana. Toisin kuin perinteiset kääntäjät ja JIT-kääntäjät, eivät tulkit missään vaiheessa käännä käskyjä suorittavan alustan konekielelle, vaan kutsuvat kirjastofunktioita, joissa on tieto mitä kunkin käskyn kuuluu suorittaa. Tulkkauksen etuihin kuuluvat olematon käännökseen menevä aika ja alustariippumattomuus. Haittapuolena on ohjelman suorituksen huomattava hitaus, joka on seurausta funktiokutsuihin menevästä ajasta. Tulkkien tulkkaamia kieliä kutsutaan usein skriptikieliksi tai komentosarjakieliksi. (25.)

Ensimmäinen tulkki kirjoitettiin vuonna 1958 LISP:lle IBM 704 -tietokoneelle. Ennen tätä ohjelmat käännettiin joko ennen suoritusta tai kirjoitettiin suoraan konekielellä. Alkujaan tulkin sijaan kielelle oli tarkoitus kirjoittaa kääntäjä, mutta se osoittautui hankalaksi tehtäväksi ja niinpä kääntäjän sijaan kielen kehittäjät kirjoittivat assemblyllä funktioita, jotka tarjosivat jonkinlaisen suoritusympäristön LISP:lle. (26.)

Kuten JIT-kääntäjiin, myös tulkkeihin liittyy tiiviisti tavukoodin käsite. Suosituista tulkatuista kielistä muun muassa Python, Ruby ja PHP toimivat virtuaalikoneella, joka suorittaa lähdekoodista käännettyä tavukoodia. Tosin kuten niin usein ohjelmoinnissa, ei tämä ole aivan absoluuttinen totuus. Jokaiselle edellä mainituista kielistä on useamman tyyppisiä tulkkeja ja kääntäjiä, joten prosessi voi erota valitusta ympäristöstä riippuen.

Tulkit ja tulkatut kielet ovat nykypäivänä huomattavan suosittuja. Ne tarjoavat nopean ja helpon ohjelmistokehityksen usein ”riittävällä” suorituskyvyllä. Marraskuussa 2015 TIOBEn mittauksen mukaan kymmenestä suosituimmasta ohjelmointikielestä neljä (Python, PHP, Ruby ja Perl) ovat oletuksena tulkattuja kieliä. Tosin TIOBEn käyttämiä mittaustekniikoita vastaan on esitetty kritiikkiä. (27; 28; 29.)

3 Kääntäjän front-endin vaiheet

Tässä luvussa käsitellään käännösprosessin suoritusalustasta riippumattomia toimenpiteitä perinteisten kääntäjien näkökulmasta, koska insinöörityöni työosa muistuttaa toiminnaltaan perinteisen kääntäjän front-endiä. Assemblereissa, JIT-kääntäjissä ja tulkeissa prosessi voi erota joiltain osin tulevien sivujen selvityksestä. Huomionarvoista on myös se, että tulevissa luvuissa oleva selvitys ei ole ainoa oikea reitti kääntäjän implementoimiseen, vaan perustuu vahvasti Appelin (30) ja Ahon, Lamin, Setthin ja Ullmanin (1) teoksiin.

3.1 Syötteen lukeminen

Valitusta kääntäjätyypistä riippumatta ohjelman kääntäminen alkaa aina lähdekoodin lukemisesta. Ensimmäiseksi kääntäjä suorittaa lähdekoodille leksikaalisen, eli sanastollisen analyysin, jossa lähdekoodi pilkotaan yksittäisiksi sanoiksi, joilla on jokin kollektiivinen merkitys.

Leksikaalinen analyysi

Leksikaalisessa analyysissä lähdekoodi järjestetään yksittäisiksi sanoiksi, lekseemeiksi, jotka ovat ohjelmakoodin pienimpiä merkityksellisiä osia eli loppusymboleita. Lekseemit voivat olla esimerkiksi literaaleja (1, 1.5, ”Vilma”), varattuja sanoja (int, char) tai operaattoreita (+, -, *). Leksikaalisen analyysin idea on siis kerätä alkuperäisestä lähdekoodista irti kaikki merkityksellinen ja mahdollisesti ”poistaa” asiat, jotka eivät vaikuta ohjelman suorituslogiikkaan millään tapaa, kuten kommentit ja välilyönnit (joskin välilyönnit ja sarkainmerkit ovat merkityksellisiä merkkejä joissain ohjelmointikielissä, esimerkiksi Pythonissa). Poistolla tässä tapauksessa tarkoitetaan sitä, että leksikaaliselta analysoijalta eteenpäin menevä tieto ei sisällä tietoa edellä mainituista asioista, eikä suinkaan niiden kadottamista alkuperäisestä koodista. Tämä tosin voidaan jättää myös parserin vastuulle, jos niin halutaan, mutta se on työläs ratkaisu. (1; 30.)

Leksikaalista analysointia voi edeltää makroesiprosessointi, jossa osa lähdekoodista korvataan makrojen sisällöllä. C ja C++ ovat malliesimerkkejä tästä. Esimerkiksi seuraava C- ja C++-ohjelmoijille tuttu lyhyt rivi

```
#include <stdio.h>
```

korvautuu esikääntäjän toimesta stdio.h-tiedoston sisällöllä, jossa on esimerkiksi yleisesti käytetty printf()-funktion esittely. Esikääntäjä voidaan myös integroida leksikaalisen analysoijaan. (31.)

Leksikaalinen analysoija ei ole kovin monimutkainen, mutta se sisältää monia yleiskäyttöisiä ominaisuuksia, joista on hyötyä myöhemmin käännöksen aikana ja myös käännökseen liittymättömissä asioissa. Näitä ovat säännölliset lausekkeet ja äärelliset automaatit. Huomioitavaa toki on, että leksikaalinen analysoija voidaan toteuttaa myös ilman edellä mainittuja. (32.)

Säännölliset lausekkeet ovat tehokas tapa etsiä ja erotella lähdekoodista varattuja sanoja, muuttujanimiä ja operaattoreita. Säännöllisillä lauseilla voidaan myös löytää virheitä lähdekoodista, sillä jokaisen lähdekoodin sanan tulisi löytyä säännöllisellä lausekkeella. Jokainen käsittelemättä jätetty sana tai merkki on viite syntaksivirheestä. Yksittäisten säännöllisten lausekkeiden tuloksista saadaan toimiva leksikaalinen analyysi muodostamalla äärellinen automaatti tai sitä simuloiva prosessi, joka vastaanottaa säännöllisten lausekkeiden löytämiä merkkijonoja ja prosessoi ne. (30, s. 18–30.)

Äärelliset automaatit ovat pohjimmiltaan äärellisen tilakoneen siirtymädiagrammeja, joiden tehtävänä on vastata jokaiseen mahdolliseen syötteeseen joko kyllä tai ei. Äärellisiä automaatteja on niin deterministisinä kuin epädeterministisinä. Nimensä mukaisesti äärellisissä automaateissa mahdollisten tilojen lukumäärä on rajallinen.

Äärellisessä deterministisessä automaatissa jokaista syötteen merkkiä kohden on yksi ja vain yksi mahdollinen tilasiirtymä. Toisin sanoen, jos alkupisteessä A saadaan mikä tahansa syötteen merkki tai symboli, voi tämä merkki tai symboli johtaa vain yhteen sitä seuraavaan tilaan B. Esimerkiksi C-kielisestä lähdekoodista voidaan lukea alkupisteessä A merkki ”i”, joka johtaa vain yhteen tilaan B. Tilassa B luetaan seuraava merkki, joka johtaa tilaan C. Oletetaan, että tilassa B luetaan seuraavaksi merkki ”e”. Tämä johtaa tilaan C, jossa voidaan jo päätellä käsittelyssä olevan jokin käyttäjän määrittelemää tunnus muuttujalle, funktiolle tai tietorakenteelle johtuen siitä, ettei C-kielessä ole mitään ”ie”-alkuista varattua sanaa. Sen sijaan, jos tilassa B olisi luettu merkki ”f”, oltaisiin tilassa C, jossa käsitellään joko varattua sanaa ”if” tai ”if”-alkuista käyttäjän määrit-

telemää tunnusta. Tässä tapauksessa olisi olemassa vielä hypoteettinen tila D, joka realisoituu syötteen seuraavaa merkkiä lukiessa. Tila C olisi lopullinen tila, mikäli seuraava merkki olisi mikä tahansa muu kuin muuttujan nimessä hyväksytty merkki (a–z, A–Z, _, tai 0–9) ja kaikissa muissa kieliopillisesti korrekteissa tapauksissa edettäisiin tilaan D, joka olisi tällä erää automaatin lopullinen tila. Kieliopillisesti virheellinen merkki johtaisi joko määrittelemättömään tilaan tai tilaan D, jossa virhe on sallittu lopputila. Deterministisessä automaatissa syötteen jokaiselle merkille on siis olemassa valmiiksi määritelty polku, jossa jokaista merkkiä vastaava tilamuutos on etukäteen tiedossa.

Äärellisessä epädeterministisessä automaatissa sen sijaan yksi merkki voi johtaa tilasta A kahteen toisistaan eriävään tilaan B. Lisäksi tällä automaattityypillä on olemassa tiloja, joista voidaan siirtyä toiseen tilaan lukematta syötteestä yhtään merkkiä. Epädeterministisessä automaatissa ei siis ole ennalta tiedossa olevaa yksiselitteistä polkua tilasta toiseen syötteen merkkien perusteella.

Yleensä säännöllinen lauseke halutaan muuntaa äärelliseksi epädeterministiseksi automaatiksi, joka vuorostaan muunnetaan äärelliseksi deterministiseksi automaatiksi. Säännöllinen lauseke voidaan muuntaa suoraan äärelliseksi deterministiseksi automaatiksi, mutta muuntamisoperaation työmäärä on käytännössä sama kuin äärellisen epädeterministisen automaatin toteutus. Syy, miksi säännölliset lausekkeet halutaan muuntaa äärelliseksi deterministiseksi automaatiksi, on suorituskyky. Vaikka eri automaattien rakentamiseen kuluu aluksi aikaa, on äärellisen deterministisen automaatin merkkijonoa kohti kuluva suoritusaika luetun merkkijonon kokoon perustuva vakio. Äärellisen epädeterministisen automaatin suoritukseen kuluva aika on vuorostaan merkkijonon koko * säännöllisen lausekkeen koko. (1.)

Oli leksikaalisen analysoijan toteutustapa mikä tahansa, sen tulee aina noudattaa sääntöä, jonka mukaan merkkijonon pisin kieliopillisesti hyväksyttävä alimerkkijono muodostaa lekseemin. Leksikaalinen analysoija ei saa välittömästi lähettää eteenpäin merkkijonoa ”if”, vaikka tämä olisi sinällään validi lekseemi, vaan sen tulee tarkistaa, onko tuleva merkki sellainen, että merkkijono ”if” olisikin vain osa pidempää lekseemiä. (30.)

Jäsennys

Jäsentäjän tehtävänä on lukea leksikaaliselta analysoijalta tulevia lekseemejä, vahvistaa niiden syntaksinen korrektius ja rakentaa jäsennyspuu joko implisiittisesti tai suoraan, mitä kääntäjän mahdolliset myöhemmät vaiheet prosessoivat. Jos mahdolliset myöhemmät front-endin (alustariippumattomat syötteenkäsittelyn) vaiheet on toteutettu jäsentäjään, ei jäsennyspuun rakentaminen ole pakollista. Jäsentäjältä voidaan odottaa ymmärrettävää viestiä virhetilanteista ja kykyä ratkoa jäsennysvaiheen aikana esille tulleita virheitä, kunnes lähdekoodi on saatu prosessoitua loppuun.

Yleisimmät kääntäjien jäsennyspuun muodostamiseen käyttämät metodit ovat ylhäältä alas ja alhaalta ylös. Kummatkin tavat lukevat leksikaaliselta analysoijalta tulevia lekseemejä vasemmalta oikealle, mutta ylhäältä alas tekniikka luo ensin jäsennyspuun juuren johon liitetään solmuja, kun taas alhaalta ylös tekniikka tekee solmujen kautta juureen. Vaikka tekniikat eivät sovellu kaikkien kielten jäsentämiseen, ovat ne riittävät useimmille nykyaikaisille ohjelmointikielille, jotka perustuvat LL (Left to right Leftmost derivation)- tai LR (Left to right Rightmost derivation) -kielioppiin.

LL-jäsentäjät ovat tekstiä vasemmalta oikealla lukeva ylhäältä alas jäsennyspuunsa rakentava jäsentäjätyyppi. Niiden alaluokka on LL(k)-jäsentäjät, jossa k ilmaisee, kuinka monta merkkiä (merkin tarkoittaessa tässä kohtaa lekseemin luokitusta) jäsentäjä lukee pinosta löytyvän tämänhetkisen merkin lisäksi, ennen kuin tekee päätöksen jäsennyspuun seuraavan solmun luonnista. Nämä jäsentäjät ovat siten myös ennustavia, sillä niillä ei juuri ole tietoa tulevista merkeistä ja niiden vaikutuksesta jäsennyspuuhun. Kiinnostavin tapaus on LL(1), koska sen kielioppi on vääjäämättä sellainen, että siitä muodostettu ennustava jäsennystaulukko ei sisällä kahta yhtäläistä merkintää.

LR-jäsentäjät ovat tekstiä vasemmalta oikealla lukeva alhaalta ylös jäsennyspuunsa rakentava jäsentäjätyyppi. Kuten LL-jäsentäjillä, on LR-jäsentäjilläkin alaluokka LR(k)-jäsentäjät, joissa k ilmaisee täsmälleen samaa asiaa kuin LK(k)-jäsentäjillä. Näistä kahdesta jäsentäjätyypistä LR-jäsentäjät ovat tehokkaampia, sillä ne kykenevät muun muassa jäsentämään liki kaikki kontekstittoman kieliopin mukaiset kielet eikä niiden tarvitse perustaa silmujen muodostamista arvaukseen ja ne kykenevät löytämään syntaksivirheet nopeammin. (1; 30.)

Jäsennyspuiden lisäksi hyvä jäsentäjä kykenee palauttamaan tietoja mahdollisista syntaksivirheistä. Syntaksivirheitä ovat esimerkiksi ylimääräiset tai puuttuvat lohkomerkit, kuten C:n ”{” ja ”}”. Sekä LL- että LR-jäsentäjät kykenevät palauttaman virheitä heti kun mahdollista, eli tilanteessa, jossa leksikaaliselta analysoijalta tulevia merkkejä ei voida enää analysoida käännettävän kielen kieliopin mukaisesti.

Jäsentäjä voi reagoida löytämiinsä virheisiin useilla tapaa, joista ehkä yksinkertaisin on lopettaa lähdekoodin jäsennys heti virheen löydyttyä ja palauttaa siitä viesti ohjelmoijalle. Toisaalta jäsentäjä voi yrittää selvitä virhetilanteesta ja jatkaa lähdekoodin jäsentämistä keräten tiedot myös myöhemmin mahdollisesti löytyvistä syntaksivirheistä ja palauttaen kerralla kaikki löytämänsä virheet. Mahdollisia selviytymiskeinoja ovat esimerkiksi paniikkitila, jossa jäsentäjä lukee leksikaaliselta analysoijalta tulevia merkkejä, kunnes se löytää synkronoivan merkin, jotka ovat usein rajaavia merkkejä kuten puolipisteitä tai lohkon lopetusmerkkejä. Vaikka tila jättää suuren määrän lähdekoodia jäsentelemättä, se on toiminnaltaan yksinkertainen. (1.)

Semanttinen analyysi

Semanttinen analyysi eli merkitysanalyysi yhdistää muuttujien määritelmät niiden käyttötapauksiin, tarkistaa muuttujien ja niihin sidottujen operandien tyypit ja muokkaa lähdekoodin (yleensä) abstraktin kielen sellaiseen yksinkertaiseen muotoon, että siitä on helpompi muodostaa konekieltä. Semanttisen analysoijan tehtäviin kuuluu myös virheenkäsittely ja semanttisista virheistä ilmoittaminen. Esimerkkinä semanttisesta virheestä voidaan mainita taulukon indeksointi liukuluvulla, joka on useassa kielessä kiellettyä. Semanttinen analysoija voi käyttää analysoinnissaan hyväksi jäsentäjän tekemää jäsennyspuuta. (30; 1.)

Semanttisessa analyysissä symbolitaulut ja niiden indeksointinopeudet ovat tärkeässä asemassa. Joka kerta, kun lähdekoodista havaitaan jokin muuttujan käyttötapaus (ts. mikä tahansa muu kuin kyseisen muuttujan määrittely), täytyy sen sijainti ja tyyppi tarkistaa symbolitaulusta. Symbolitauluja voi myös olla useita. Näin saadaan pidettyä kirjaa kaikista aktiivisista ympäristöistä, kuten luokista ja niiden muuttujien arvoista. Tämä myös mahdollistaa saman muuttujanimen käytön eri näkyvyysalueissa. Tehokkuusvaatimus on seurausta siitä, että suuressa imperatiivisella ohjelmointikielellä (esim. C, Java) toteutetussa ohjelmassa voi olla tuhansia yksittäisiä muuttujamääritelmiä.

Symbolitaulut voidaan toteuttaa imperatiivisesti tai funktionaalisesti riippumatta siitä onko käännettävä kieli imperatiivinen vai funktionaalinen. Imperatiiviset toteutukset nojaavat yleensä niputettuihin hajautustauluihin, joiden ominaisuuksiin kuuluu nopea indeksointi ja ylikirjoittaja päivitys eli päivitystilanteessa indeksillä löytyvä aikaisempi tieto poistetaan ja korvataan uudella arvolla. Funktionaalisissa toteutuksissa aikaisempaa tietoa ei poisteta vaan päivityksessä luodaan uusi taulu, johon uusi arvo sijoitetaan. Koska uuden hajautustaulun luonti on resursseja vievä toimenpide, on funktionaalisessa toteutuksessa binääripuu hajautustaulua parempi toteutusratkaisu. (30, s. 106–107; 1.)

3.2 Välitason koodigenerointi

Kun alkuperäinen lähdekoodi on lopulta saatu jäsenneltyä ja sen suorituslogiikka on selvillä, siitä on aika muodostaa välitason koodia. Välitason koodin idea on olla sellainen esitys alkuperäisestä koodista, että siitä on helppo muodostaa suoritettava konekielinen koodi menettämättä kuitenkaan alkuperäisen koodin logiikkaa. Vaihe on siis siirtymä abstraktista syntaksista abstraktiseen konekieleen. Abstrakti syntaksi voidaan kääntää suoraan konekielelle, mutta tällöin menetetään välitason koodigeneroinnin suoma siirrettävyys. Ilman välitason koodigenerointia X määrä ohjelmointikieliä Y määrälle suoritusalustoja vaatii X * Y määrän toteutuksia. Välikoodigeneroinnilla korkean tason kielestä voidaan luoda sellainen abstraktimpi toteutus, että edellinen X * Y muuttuu muotoon X + Y. Tämä on seurausta siitä, että ilman välikoodigenerointia kääntäjän front-endin on luotava alustariippuvaiselle back-endille sopivaa koodia käytettäväksi ja eriävä suoritusympäristö vaatii erilaisen suoritettavan koodin. Siispä 2 ohjelmointikieltä, joita on tarkoitus suorittaa 3:lla eri alustalla vaatisi 3 kääntäjätoteutusta ohjelmointikieltä kohden, mistä kukin olisi eri suoritusalustaa varten tehty, mikä luo tilanteen 2 * 3 eli yhteensä 6 kääntäjätoteutusta. Sen sijaan tilanteessa, jossa front-end luo back-endille yhdenlaista käännettävää välitason koodia, voidaan kahdelle kielelle luoda yksi front-end kieltä kohden ja jokaiselle alustalle oma back-endinsä, mikä tarkoittaa kolmea back-endiä. Koska back-endin saama koodi on nyt lähtökielestä riippumatonta välitason esitystä, on toteutusten määrä nyt 2 + 3 eli 5. Työssä käytettyjen kirjallisuuslähteiden noudattamassa arkkitehtuurissa välitason koodigenerointi on kääntäjän front-endin viimeinen vaihe, mikäli kääntäjään sellainen toteutetaan. (1, s. 357; 30, s. 150–151)

Välitason koodin voi toteuttaa monella eri tapaa. Se voi olla ”oikea” kieli, tai se voi olla kokoelma tietorakenteita. Esimerkiksi ensimmäinen C++-kääntäjä käytti välikielenään C:tä, sillä kääntäjän back-end oli C-kielen kääntäjä. Myös moni nykyaikainen korkean tason kieli käyttää C:tä välitason kielenään johtuen sen suhteellisesta läheisyydestä konekielen kanssa ja siitä, että C-kielelle on kääntäjiä liki kaikille alustoille. Muita suosittuja välitason kieliä ovat Javan virtuaalikoneen tavukoodi ja Microsoftin CIL (Common Intermediate Language), jota .NET-ympäristö ja avoimen lähdekoodin Mono käyttävät. Usean suositun Linux-jakelun mukana tuleva GCC (GNU Compiler Collection) vuorostaan käyttää välitason esityksenä abstraktia syntaksipuuta. GCC:n front-endit sisältävät jäsentäjät jokaiselle kääntäjän tukemalle ohjelmointikielelle. Tämän vuoksi jokainen suoritusympäristö vaatii vain yhden back-endin tuettua ohjelmointikieltä kohden. Kun otetaan huomioon pelkästään pääjakelun tukemat 6 kieltä (avoimen lähdekoodin vuoksi on olemassa jakeluja, joissa tuettujen kielten lukumäärä voi olla eri) ja 48 eri käskykantaa, voidaan säästetyn työmäärän katsoa olevan huomattava (ideaalitilanteessa 288 toteutusta voidaan korvata 54 toteutuksella). (1, s. 358; 33; 34; 35.)

Oli välitason koodi missä muodossa tahansa, sillä on kaksi tärkeää vaatimusta:

1) Käännettävän lähdekoodin merkitys ei saa muuttua.

2) Generoitu koodi ei saa käyttää mielin määrin alustan resursseja.

Kolmiosoitekoodi

Eräs suosittu välitason koodi on kolmiosoitekoodi, joka perustuu operandeihin (jotka siis ovat osoitteita) ja operaattoreihin. Kuten nimestä voi mahdollisesti päätellä, on kolmiosoitekoodissa enimmillään kolme operandia suoritettavaa riviä kohden. Operandeja kolmiosoitekoodissa ovat

1) muuttujanimet, joiden tiedot korvataan implementaatiovaiheessa symbolitauluun osoittavalla osoittimella, joka on osoitin siihen, mitä tietoja muuttujanimen takana oikeasti on

2) vakiot eli ohjelmasuorituksen aikana tilansa ja arvonsa säilyttävä tieto

3) kääntäjän generoimat väliaikaismuuttujat, joita erityisesti optimoivat kääntäjät tarvitsevat toimintaansa.

Operaattoreita vuorostaan ovat esimerkiksi binääriset ja unaariset laskentaoperaattorit, kopiointioperaattorit, ehdottomat ja ehdolliset hyppykäskyt sekä aliohjelmien kutsuihin liittyvät käskyt. Välitason koodia esittämään valittujen operaattoreiden lukumäärään vaikuttavat seikat ovat vaatimus siitä, että operaattoreita on riittävästi toteuttamaan käännettävän kielen matemaattiset esitykset mutta toisaalta niin vähän, ettei lopullisen konekielen generointi hankaloidu liiaksi.

Kolmiosoitekoodin esitys voidaan toteuttaa nelikkona tai kolmikkona. Nelikot ja kolmikot ovat tietueita, joissa on tilaa joko kaikille kolmelle operandille ja yhdelle operaattorille (nelikot) tai kahdelle operandille ja yhdelle operaattorille (kolmikot). Kolmikossa kolmas operandi, tulos, on korvattu viittaamalla itse käskyyn. Tällä vältetään ylimääräisten muuttujien luonti, mutta toisaalta käskyjen siirtäminen on vaikeampaa kuin nelikkoesityksessä, mitä optimoivat kääntäjät käyttävät hyödykseen.

Kolmikkoesitys lausekkeelle A+B*C voisi olla seuraava:

OP	arg1	arg2
*	B	C
+	A	(0)

jossa (0) on viittaus edelliseen sijaintiin.

Vastaavan tapauksen nelikkoesitys olisi vastaavasti muotoa

OP	arg1	arg2	tulos
*	B	C	t1
+	A	t1	t2

Kolmikkoesityksen vahvuuteen kuuluu sen yhteensopivuus suunnattujen syklittömien verkkojen (DAG, directed acyclic graph) kanssa. Kun kolmikkoesityksestä karsitaan toisto, on sen puurakenne suunnattu syklitön verkko. (1, s. 363–370.)

Muuttujatyypit ja tyyppimuunnokset

Välikoodigeneroinnissa voidaan määritellä muuttujan tyypin perusteella, paljonko se tulee tarvitsemaan muistia ohjelman ajon aikana, ja sitoa muuttuja muistipaikkaan sen mukaan. Muuttujalle varattu muistipaikka tallennetaan symbolitauluun myöhempää käyttöä varten. Suorituksen aikaista dynaamista muistinhallintaa vaativat muuttujat, kuten muuttuvamittaiset taulukot, merkkijonot ja vastaavat, saavat pelkästään tilanvarauksen muuttujaan osoittavaa osoitinta varten. (1, s. 373) Dynaaminen muisti sijaitsee yleensä kekomuistissa. Kekomuistiin ja dynaamiseen muistinhallintaan palataan luvussa 4.1.

Muistia käsitellään yleensä vähintään tavun kokoisessa yksikössä. Yksi tavu vuorostaan koostuu kahdeksasta bitistä. Jokaisella muuttujatyypillä on leveys, joka ilmaisee kokonaisluvulla, kuinka monta tavua muuttujatyyppi tarvitsee. Usean tavun kokoiset tietotyypit ja muuttujat tallennetaan peräkkäisille tavuille ja osoitteeksi sille annetaan kyseessä olevalle instanssille varatun ensimmäisen tavun osoite. (1, s. 374)

Muuttujia voidaan myös (kielioppisääntöjen sen salliessa) käyttää sekaisin keskenään. Esimerkiksi tapauksessa A + B muuttuja A voi olla mielivaltaisen kokoinen kokonaisluku ja muuttuja B liukulukutyyppiä. Javan kieliopissa on määritelty tyyppimuunnoksia varten leventävät ja kaventavat muunnokset, joita käytetään sen mukaan, ollaanko muuttujaa muuntamassa muistinkäytön tai lukualueen näkökulmasta pienemmäksi vai suuremmaksi. Javan kielioppi sallii esimerkiksi muuttujatyypin char leventää muuttujatyypiksi int, mutta ei muuttujatyypiksi short. Tyyppimuunnokset voivat olla kääntäjän pakottamia eli implisiittisiä tai ohjelmoijan tekemiä eli eksplisiittisiä. (1, s. 388–389.)

Kontrollirakenteet

Kontrollirakenteiden, joita ovat esimerkiksi toisto- ja ehtorakenteet, käännös on sidoksissa boolean-lausekkeiden käännöksiin. Boolean-lausekkeilla ohjataan esimerkiksi toistorakenteiden päättymistä ja ohjelmasuorituksen haarautumista ehtorakenteissa.

Boolean-lausekkeet koostuvat boolean-operaattoreista, jotka ovat JA, TAI sekä EI. (1, s. 399.)

Koska konekieli ei ole strukturaalinen ohjelmointikieli, joudutaan ylemmän tason ohjelmointikielten ehtorakenteet muuntamaan hyppykäskyiksi osoitelappuihin. Välikoodigeneroinnissa kääntäjä luo jokaiselle koodihaarautumalle uuden osoitelapun, jolle generoidaan oma kolmiosoitekoodinsa. Toistorakenteet muodostetaan vastaavaan tapaan luomalla uusi osoitelappu, johon palataan hyppykäskyllä takaisin niin kauan, kunnes ohjelman suoritus on saavuttanut halutun tilan. (1, s. 403.)

Helpoin tapa kontrollirakenteiden tuottamiseen on laskea niiden tila johonkin väliaikaismuuttujaan ja suorittaa hyppy tutkimalla tätä arvoa. Parempilaatuista koodia saa hyppäämällä suoraan haluttuun paikkaan, mutta tämä vaatii perittyjen attribuuttien käyttämistä. (1, s. 402.)

Kontrollirakenteiden muuntaminen osoitelapuiksi ja hyppykäskyiksi sellaisenaan on kuitenkin ongelmallista, sillä suora käännös johtaa massiiviseen määrään tarpeettomia hyppykäskyjä ja osoitelappuja. Ongelma voidaan poistaa joko optimointivaiheessa tai välikoodigeneroinnissa backpatchingilla. Backpatchingissa vielä tuntemattomia hyppyosoitteita varten luodaan lista. Myöhemmin käännöksen aikana listaan käydään asettamassa osoitteita sitä mukaan, kun niihin tullaan vastaan. (1, s. 410.)

Oman lisänsä kontrollirakenteisiin tuovat switch-caset. Ne ovat kontrollirakenteita, joissa tarkistetaan muuttujan tila T ja etsitään sitä vastaava arvo A, jossa suoritetaan arvoon liitettyjä komentoja. Mikäli T:lle ei löydy arvoa A, siirrytään suorittamaan oletusarvoon liitettyjä komentoja. T:tä vastaavan arvon A etsintä on haarautuma, jossa on n määrä haarautumia.

Mikäli etsittävien arvojen lukumäärä on pieni, voidaan switch-case muuntaa perinteiseksi vertailurakenteeksi, joissa syötettä verrataan jokaiseen koodaajan määrittelemään arvoon A, kunnes löydetään haluttu. Tällainen switch-case voidaan muodostaa tekemällä taulu, jossa on arvo ja osoitelappu koodiin, jota arvolla halutaan suorittaa. Suurempi arvojoukko on tehokkaampaa käydä läpi hajautustaululla. (1, s. 419.)

Erityistapauksen muodostavat arvojoukot, joiden arvot lähellä toisiaan. Tällaiset switch-caset voidaan n-määräisten haarautumien sijaan korvata niputetulla lajittelulla, jossa

pienemmät osa-arvojoukot ovat omassa nipussaan. Mikäli muuttujan arvo ei vastaa minkään nipun raja-arvoja, voidaan hypätä suoraan suorittamaan oletusarvoon liitettyä koodia. (1, s. 419)

Aliohjelmat

Aliohjelmat, eli tutummin kielestä riippuen funktiot tai metodit, ovat hyvän ohjelmointitavan mukainen tapa jakaa ohjelman toimintalogiikka pienempiin toimintayksiköihin. Täten lähdekoodista saadaan luettavampaa ja ylläpidettävämpää sekä vältetään mahdollista koodin toistumista. Ei ole esimerkiksi kovin mielekästä määritellä joka kerta erikseen koodia, joka lukee syötteen ohjelmakäyttäjänä näppäimistöltä.

Jokaisella aliohjelmalla on oma pinomuistialue, johon sijoitetaan aliohjelman suoritukseen tarvitsema tieto, kuten aliohjelman paikalliset muuttujat. Aliohjelman suorituksen alussa sen paikalliset muuttujat laitetaan pinon huipulle ja suorituksen loputtua pino tyhjennetään. Pinomuistin toimintaan palataan luvussa 4.1.

Aliohjelmat ottavat usein vastaan 1–n, jossa n on mielivaltainen mutta mielellään mahdollisimman pieni kokonaisluku, parametria. Parametrit välitetään yleensä

1) arvona, jolloin lausekkeen arvo lasketaan ennen sen välittämistä aliohjelmalle tai muuttujan tilasta tehdään kopio ja välitetään se aliohjelmalle; tällöin lähetetyn muuttujan arvo ei (yleensä, poikkeuksia mm. taulukot C:ssä) muutu kutsuvassa funktiossa, vaikka sille kutsuttavassa funktiossa jotain tehtäisiinkin.

2) referenssinä, jolloin aliohjelma saa muuttujan tai ilmaisun arvon sijaan viittauksen muuttujan tai ilmaisun arvon muistipaikkaan. Parametrin välittyessä kutsuvasta funktiosta kutsuttuun funktioon referenssinä, muuttuu sen arvo kutsuvassa samaksi, kuin mihin se on kutsutussa päättynyt, sillä kumpikin käsittelevät samaa muistialuetta; sen sijaan lausekkeessa käytettyä tietoa ei referenssinä pystytä muokkaamaan, sillä osoitin on saatu vain lausekkeen tulokseen. (1, s. 34.)

Lisäksi aliohjelmilla on usein muuttujien tapaan tyyppi. Aliohjelmien tyyppi kertoo, minkä tyyppistä tietoa aliohjelma tulee palauttamaan. Aliohjelma, joka ei palauta mitään, on yleensä tyypiltään void (eli tyypitön). Void voidaan myös merkitä sellaisen aliohjel-

man parametriksi, joka ei vastaanota minkäänlaisia parametreja. Näin toimitaan esimerkiksi C:ssä.

Välikoodigeneroinnissa jokainen aliohjelma saa oman symbolitaulun sitä mukaa, kuin aliohjelmamäärityksiin törmätään lähdekoodia läpi käydessä. Aliohjelmakutsujen lausekemuotoiset parametrit kannattaa muuntaa kolmiosoitekoodiksi, jotta ne voidaan muuntaa arvoiksi ja sitä kautta osoitteiksi. (1, s. 423.)

4 Kääntäjän back-endin vaiheet

4.1 Suoritusaikainen ympäristö

Kääntäjän tärkein tehtävä on käännöksessä implementoida tarkasti korkean tason ohjelmointikielen abstraktit komennot suoritusympäristön ymmärtämään muotoon menettämättä tai muuntamatta ylemmän tason loogista tarkoitusta. Näitä abstraktioita ovat näkyvyysalueet, nimet, tietotyypit, operaattorit ja liki kaikki, mitä ohjelmoija arjessaan käyttää.

Jotta nämä abstraktiot voitaisiin esittää koneen ymmärtämässä muodossa, joutuvat kääntäjät muodostamaan suoritusaikaisen ympäristön, jossa kääntäjän kääntämä ohjelmaa suoritetaan. (1, s. 427.)

Muistinhallinta

Kääntäjän näkökulmasta sen tuottama ohjelmat toimivat käyttöjärjestelmän, tietokoneen ja kääntäjän jakamassa loogisessa muistiavaruudessa. Tämän muistiavaruuden kartoitus ja muuntaminen fyysisiksi osoitteiksi on käyttöjärjestelmän tehtävä. (1, s. 407.)

Kääntäjän tehtävä vuorostaan on varata ohjelman lähdekoodin muuttujatyyppien perusteella riittävästi tilaa ohjelman käyttämille muuttujille. Kääntäjän muistinvaraus on suoritusympäristö riippuvaista, mutta alustoilla, joilla muistista ei ole pulaa, kääntäjä voi varata tarvetta enemmän muistia muuttujille, jotta ne saataisiin sijoitettua osoitteita tasaamalla muistihakujen näkökulmasta nopeaksi. Nopeus perustuu siihen, että prosessori (alustasta riippuen) hakee kerrallaan usean tavun kokoisia muistialueita, kuten

vaikka konekielisen sanan. Jos tietoa ei ole sijoitettu sanan rajojen sisälle, joutuu prosessori suorittamaan useamman muistihaun ohjelmasuorituksen aikana. (36; 37.)

Kääntäjän tuottaman konekielisen koodin koko on käännösaikana vakio, joten kääntäjä voi sijoittaa tuottamansa konekielisen koodin koodisegmenttiin, joka on usein muistiavaruuden alapäässä ja sisältää ohjelmasuoritukselle tarpeelliset käskyt. Tiedossa on myös osa ohjelmasuorituksen käsittelemään tietoon liittyvästä koodista, kuten globaalit vakioarvot. Ne sijoitetaan konekielisen koodin staattiseen segmenttiin. Yksi syy mahdollisimman suuren tietomäärän staattisen varaamiseen on, että staattiseen segmenttiin sijoitetun tiedon osoitteet ovat käytettävissä käännösaikana ja näitä osoitteita voidaan hyväksikäyttää käännösprosessissa sijoittamalla osoitteet suoritettavaan ohjelmakoodiin. (1, s. 429.)

Jäljelle jäävät keko- ja pinomuistit ovat muistiavaruuden vastakkaisissa päissä. Kummatkin ovat tyypiltään dynaamisia muisteja, eli niiden koko voi muuttua ohjelmasuorituksen aikana. Nämä muistialueet kasvavat toisiaan kohti. Pinomuistiin varataan yleensä aliohjelmien pienet muuttujat sekä tietorakenteet. Kekomuisti on muistialueista suurempi ja se sisältää suuremmat ja/tai pidempiaikaiset muuttujat. Yleensä muistit on järjestetty siten, että kekomuisti sijaitsee muistiavaruuden alapäässä kasvaen (osoiteavaruudessa) ylös ja pinomuisti on ylhäällä kasvaen alas. (1, s. 429.)

Pinomuistin hallinta

Liki kaikki aliohjelmia tukevien ohjelmointikielten kääntäjät säilyttävät pinomuistissa ainakin osaa aliohjelmien ajoaikana tarvitsemista muuttujista. Aliohjelmaa kutsuttaessa sen käyttämät muuttujat, sille välitettävät parametrit ja paluuosoite säilytetään pinoon. Aliohjelmasta poistuttaessa sen käyttämä pino tyhjennetään. Tällä saavutetaan se, että tietoa voidaan jakaa aliohjelmakutsujen välillä, mitä ei suoriteta ajallisesti päällekkäin, ja muistien suhteelliset osoitteet ovat aina samat riippumatta aliohjelmasuoritusten järjestyksestä. (1, s. 430.)

Jotta pinomuistin käyttämisessä olisi järkeä, tulee aliohjelmakutsujen olla ajallisesti erillään toisistaan. Jos aliohjelman A aktivointi aktivoi aliohjelman B, tulee B:n aktivoinnin olla ohi, ennen kuin A:n aktivointi voidaan päättää. Lisäksi B:n aktivoinnin päättyessä tulee ohjelmasuorituksen jatkua siitä, mihin A:ssa jäätiin. Jos B:n aktivoinnissa tai suorituksessa tapahtuu jokin virhe, voidaan A:n suoritus lopettaa samanaikaisesti B:n

kanssa tai virhe voidaan ottaa kiinni ja suoritusta jatkaa jossain muualla. Aliohjelmien kutsut voidaan esittää kutsupuuna. (1, s. 430–431.)

Aliohjelmien kutsut ja paluut hoidetaan usein kontrollipinolla. Jokaisella aktiivisella aliohjelmalla on oma pinokehyksensä kontrollipinossa. Pinokehys sisältää tiedot kutsupuusta kutsupuun juuren ollessa pinon pohjalla. Myöhemmät kutsupuun solut eli aliohjelmakutsut lisätään pinoon päälle siten, että uusin kutsu on aina pinon päällimmäisin. Lisäksi pinokehyksessä on tiedot väliaikaisista arvoista, joita ei voida säilyttää rekistereissä, aliohjelman paikallisesta tiedosta, koneen tilasta juuri ennen aliohjelmakutsua sisältäen paluuosoitteen, johon aliohjelmasta pitää palata, linkit toisiin pinokehyksiin, joiden tietoa saatetaan mahdollisesti tarvita aliohjelman suoritukseen ja linkki kutsuvan ohjelman pinokehykseen. Lisäksi pinokehyksessä on mahdollisesti tilaa aliohjelman mahdolliselle aliohjelmakutsuille ja niiden paluuarvoille sekä aliohjelman käyttämille parametreille, joskin nämä yleensä halutaan tehokkuussyistä rekistereihin säilöön. (1, s. 434.)

Aliohjelmakutsuja varten tarvitaan suoritettavaa koodia, joka varaa aliohjelmalle riittävän kokoisen pinokehyksen ja siirtää aliohjelman tarvitsemat tiedot sinne. Vastaavanlaista koodia tarvitaan myös aliohjelmasta paluuta varten, jotta kutsuva (ali)ohjelma voi jatkaa toimintaansa. Tämän koodin tehtävien jako on implementaatiosta riippuvainen, eikä ole oikeaa vastausta siihen, mikä osa suoritettavasta koodista kuuluu kutsuttavaan ja mikä kutsuttuun aliohjelmaan. Suuremman tehtävämäärän asettaminen kutsutulle funktiolle on koodin määrän puolesta parempi, sillä aliohjelmaa voidaan kutsua mielivaltaisen useasta lähdekoodin sijainnista, mutta sillä on vain yksi implementaatiopaikka. Toisaalta kutsuttu aliohjelma ei tiedä kaikkea tarvittavaa, joten sille ei voida antaa kaikkia pinokehyksen varaamiseen tarvittavia tehtäviä. (1, s. 436) Aho, Lam, Sethi ja Ullman suosittelevat (1, s. 436), että pinokehystä ja kutsuja luottaessa toimitaan seuraavasti:

1) Aliohjelmien keskenään käyttämät arvot sijoitetaan kutsujan pinokehyksen päälle, jotta ne olisivat mahdollisimman lähellä kutsuttavan aliohjelman pinokehystä. Näin kutsuttavan ei tarvitse kopioida kutsujan koko pinokehystä arvoja hakiessaan, ja se myös mahdollistaa vaihtelevan parametrimäärän vastaanottavat aliohjelmat, kuten C:n printf-funktion.

2) Kooltaan staattiset muuttujat sijoitetaan pinokehyksen keskiväliin. Näitä muuttujia ovat muun muassa. koneen tila ja pinokehysten väliset linkit.

3) Kooltaan dynaamiset muuttujat sijoitetaan pinokehyksen pohjalle.

4) Pinon huippu sijoitetaan prosessorin rekisteriin, mikäli mahdollista. Aliohjelma käyttää pinoa yleensä pino-osoittimen kautta, joka voidaan laittaa osoittamaan pinokehyksen kooltaan kiinteiden muuttujien kenttään, jotta niitä voidaan hakea kiinteän kokoisella siirroksella, joka on tiedossa välikoodigeneroinnissa. Tällöin kooltaan dynaamiset muuttujat jäävät pinokehyksen yläpuolelle (muistiosoitteissa) ja niiden siirrokset saatetaan joutua laskemaan ajonaikana. Vaihtoehtoisesti niihin päästään käsiksi pinon huipun osoittimen avulla. (1, s. 436; 30, s. 213.)

Näillä periaatteilla aliohjelman kutsujan vastuulle jäävät seuraavat asiat:

1) välitettävien lausekemuotoisten parametrien arvojen laskenta

2) paluuosoitteen ja kutsujan pinokehyksen huipun osoittimen tallennus kutsuttavan aliohjelman pinokehykseen

3) aliohjelman pinokehyksen huipun osoittimen osoitteenlaskenta ja sen arvon välittäminen aliohjelmalle.

Tällöin aliohjelman tehtäväksi jää

1) rekisteriarvojen tallennus ja paikallisen tiedon alustus

2) paluuarvojen tallennus

3) kutsuvan ohjelman pinokehyksen huipun osoitteen palauttaminen (ks. edellisen listan kohdan 2. jälkimmäinen tehtävä)

4) paluu paluuosoitteeseen.

(1, s. 436–438)

Kekomuistin hallinta

Siinä, missä pinomuistissa säilötään yleensä lyhytikäistä ja pienikokoista tietoa, jonka elinikä on etukäteen tiedossa (esim. paikalliset muuttujat, jotka ovat hengissä vain aliohjelman suorituksen ajan), on kekomuisti säiliönä suuremmalle tiedolle, jonka elinikä ei ole etukäteen määritelty tai joka katoaa vasta, kun ohjelmassa niin käsketään. Näitä ovat esimerkiksi C:ssä mallocilla tilavarauksensa saaneet muuttujat. (1, s. 452.)

Kekomuistia hallitsee muistiohjain, jolla on kaksi perustoimintoa:

1) Varaus. Kun ohjelma pyytää muuttujalle tilaa, yrittää muistinhallinta varata sille vierekkäisiä muistipaikkoja vaaditun tilan verran. Jos keossa ei ole riittävästi vapaata tilaa, yrittää muistinhallinta pyytää virtuaalimuistia käyttöjärjestelmältä, mikäli suoritusympäristöstä sellainen on. Jos tämäkään ei onnistu tai virtuaalimuistia ympäristöstä ei ole, välittää muistinhallinta tiedon epäonnistumisesta ohjelmalle.

2) Vapauttaminen. Muistinhallinta vapauttaa varatun muistin muistialtaaseen, josta se voidaan tarvittaessa taas varata muiden ohjelmien käyttöön. (1, s. 453.)

Ideaalitilanteessa muistia varattaisiin aina samankokoisille lohkoille, joiden elinkaari olisi ennustettavissa, mutta kuten jo luvun alusta voi arvata, näin ei käytännössä koskaan käy. Siksi muistinhallinnan on oltava tehokas käyttämänsä tilan kanssa, eli se ei saa varata liikaa muistia ohjelmalle ja varatun muistin tulisi olla eheää. Lisäksi muistinhallinnan tulisi sijoittaa ohjelman käsittelemä tieto siten, että se on nopeasti saatavilla eli sen pitäisi ymmärtää muistihierarkia. Hakuajat ovat lyhimmillään rekistereissä, joten se on ideaalipaikka muuttujille, mutta rekistereitä on vähän eikä niihin mahdu paljoa tietoa. Kiintolevyllä tai vastaavalla massamedialaitteella tilaa on yleensä kertaluokkia enemmän, mutta hakuajat ovat myös monta kertaa pidemmät. Muistinhallinnan itsensä tulisi myös toimia nopeasti, eikä se saisi vaikuttaa huomattavasti ohjelmasuorituksen nopeuteen. (1, s. 454.)

Koska suurin osa ohjelman suoritukseen kuluvasta ajasta menee pienen koodinalueen suorittamiseen, voidaan suurin osa koodista sijoittaa hitaampaan muistiin ja pieni mutta suoritusnopeuden kannalta merkittävä osa sijoittaa nopeampaan välimuistiin. (1, s. 455–456.)

Ohjelmasuorituksen alkaessa kekomuisti on eheä vapaan muistin alue, mutta ohjelmasuorituksen aikana kekoon ilmestyy muistivarausten ja -vapautusten vuoksi varattuja ja varaamattomia alueita. Jokaisella varauskerralla muistihallinta pyrkii varaamaan vapaasta muistista vähintään pyydetyn kokoisen jatkuvan muistin alueen ohjelman käyttöön. Mikäli vapaata muistia ei löydy täsmälleen pyydettyä määrää, jaetaan vapaan muistin alue pienempiin vapaisiin alueisiin. Vapautettaessa muistialueet palautetaan muistialtaaseen uutta käyttöä varten. Mikäli varauksia ja vapautuksia tehdään umpimähkään, käytettävissä oleva muistiallas hajoaa pieniin vapaisiin alueisiin eikä tietoa saada enää säilytettyä perättäisten vapaiden muistipaikkojen puutteen vuoksi. (1, s. 457–458.)

Muistin pirstaloituminen voidaan minimoida sijoittamalla käsiteltävä tieto pienimpään sopivaan muistialueeseen, jolloin suurten muistialueiden pilkkoutuvat jollain tapaa järkevästi, tai vaihtoehtoisesti ensimmäiseen sopivaan muistialueeseen, mikä minimoi muistialueen etsintään kuluvan ajan. Näistä ensimmäistä voidaan edelleen tehostaa niputtamalla vapaat muistialueet eri kokoluokkiin, jolloin käsiteltävälle tiedolle voidaan etsiä sopivasta nipusta sopiva vapaa muistialue, mikä osaltaan kutistaa vapaan alueen etsintään kuluvaa aikaa. (1, s. 558–560.)

4.2 Koodigenerointi

Koodigenerointi on insinöörityössä esitetyn kääntäjämallin suorittaman käännösvaiheen viimeinen toimenpide (1, s. 505). Siinä kääntäjä viimein luo välitason esityksestä (tai suoraan lähdekoodista, mikäli välitasoa ei ole toteutettu) ohjelman, jota valitun käskykannan tulee kyetä suorittamaan.

Käskykannat

Valitulla käskykannalla on huomattava vaikutus kääntäjän toimintaan. Yleisimpiä käskykantoja ovat CISC (Complex instruction set computing) ja RISC (Reduced instruction set computing). Ensiksi mainittua edustavat pöytäkoneista ja kannettavista tietokoneista useimmiten löytyvät x86-arkkitehtuuriin perustuvat suorittimet ja jälkimmäistä älypuhelimista ja taulutietokoneista yleisesti löytyvät ARM-arkkitehtuuriin pohjautuvat suorittimet. Tosin kumpikaan arkkitehtuuri ei ole puhdas käskykantansa edustaja. (38.)

CISC-käskykanta syntyi vastaukseksi 1960- ja 1970-luvulla esitettyihin kauhukuviin, joiden mukaan ohjelmistojen kasvava kompleksisuus johtaisi ohjelmistotuotannon kustannusten räjähdysmäiseen kasvuun. CISC:n ideana oli implementoida osa ohjelmistojen monimutkaisuudesta suoraan laitteistotasolle, mikä helpottaisi ohjelmoijien työtä ja kääntäjien kirjoittamista. Tuloksena on vähemmän koodia, sillä esimerkiksi lukujen hakeminen muistista rekisteriin, kertominen keskenään ja tuloksen säilyttäminen muistiin voidaan esittää yhdellä käskyllä sen sijaan, että ohjelmoija kirjoittaisi kaikki käskyt erikseen. Käskyjen vähäinen määrä johtaa myös muistinkäytön minimointiin, mikä erityisesti CISC-käskykannan syntyaikakauden koneilla oli haluttua, sillä muisti oli hyvin hidasta ja kallista. (39.)

RISC-käskykanta syntyi 1980-luvulla havainnosta, jonka mukaan suurinta osaa CISC-käskykanta monimutkaisista käskyistä ei käytetty. Kääntäjätkin pilkkoivat monimutkaiset käskyt pienempiin osiin ja käyttivät niiden suorittamiseen käskykannan yksinkertaisempia käskyjä. RISC:ä varten suurin osa CISC:n käskyistä pudotettiin pois, ja tavoitteeksi asetettiin käskyn ajaminen joka kellojaksolla. Lisääntynyt muistinkäyttö kompensoitiin pudottamalla tarpeettomaksi käynyt mikro-ohjelma suorittimelta pois ja tallentamalla nyt yksinkertaisiksi käyneet käskyt tilalle. RISC:ssä komennot ovat siis yksinkertaisia, rekistereitä on paljon ja koodia on paljon. CISC:ssä komennot ovat monimutkaisia ja rekistereitä sekä suoritettavaa ohjelmakoodia on vähemmän. Lisäksi CISC:ssä on useampia tapoja viitata muistiin kuin RISC:ssä ja CISC:n suorittamien käskyjen viemien kellojaksojen lukumäärä voi vaihdella. (39.)

Käskyjen valitseminen

Konekielisten käskyjen valinta on riippuvainen välitason koodiesityksestä, valitusta suoritusympäristöstä ja valittujen käskyjen laatuvaatimuksista (1, s. 508). Käskyjen valinnan vaikeus on riippuvainen valitusta suoristusympäristöstä. Mikäli valittu suoritusympäristö ei käsittele tietotyyppejä yhdenmukaisella tavalla, vaatii jokainen poikkeus oman käsittelynsä. Lisäksi valittavien käskyjen tehokkuus on sidoksissa valittuun käskykantaan. (1, s. 509.)

Jos tehokkuus ei ole yksi vaatimuksista, voidaan kolmiesityskoodi kääntää liki sellaisenaan valitulle käskykannalle, mutta tämän seurauksena on yleensä turhia muistitallennuksia ja -lukuja. Niiden välttämiseksi koodigeneroinnissa on suotavaa ottaa selville, löytyykö käskykannasta käskyjä, joilla turhat tallennukset ja lukemiset voitaisiin välttää

esimerkiksi suorittamalla asia yhdellä käskyllä (1, s. 509). Toisaalta koodigeneroinnin täytyy ottaa huomioon valitun käskyn hinta. Voi olla, että jonkin asian suorittaminen monella yksinkertaisella käskyllä on nopeampaa kuin yhdellä mutta hitaalla käskyllä.

Rekisteriallokointi

Rekisterit ovat suorittavan alustan nopeimpia laskennallisia yksiköitä, mutta niitä on yleensä varsin vähän eivätkä ne käytännössä riitä kaikelle ohjelmasuorituksessa tarvittavalle tiedolle. Niinpä eräs kääntäjän tärkeimmistä tehtävistä on päättää, mikä käsiteltävästä tiedosta sijoitetaan rekistereihin ja mikä toisaalle muistiin. Rekisteriallokointi voidaan jakaa kahteen osaan:

1) Rekisteriallokointi. Tässä vaiheessa päätetään, mitkä muuttujat sijoitetaan rekistereihin missäkin vaiheessa ohjelman suorituksen aikana.

2) Rekistereihin sijoittaminen. Tässä vaiheessa muuttujalle valitaan rekisteri, johon se sijoitetaan.

Kuten liki kaikessa kääntäjien toimintaan liittyvässä, ei rekisteriallokoinnin ongelmiin ole optimaalista vastausta eli ei ole täydellistä vastausta siihen, mikä muuttuja sijoitetaan mihin rekisteriin minä ajankohtana. (1, s. 510.)

Yksi tapa allokoida rekistereitä on varata tietty rekisteriryhmä tietyille arvoille, eli esimerkiksi antaa aritmeettisille operaatioille tietty joukko rekistereitä käytettäväksi ja pinon huipulle tietyn ryhmän rekistereitä ja niin edelleen. Tämän etuna on koodigeneroinnin yksinkertaistaminen, mutta toisaalta hankaluutena on se, että osa rekistereistä jää käyttämättä ohjelmasuorituksen aikana mahdollisesti kokonaan tai pitkiksi ajoiksi. (1, s. 553.)

Koska suurin osa ohjelmasuoritukseen kuluvasta ajasta menee toistorakenteissa, on perusteltua sijoittaa toistorakenteissa käytettävät muuttujat rekistereihin. Mikäli toistorakenteella on useita tasoja, tulee suurin osa toistorakenteeseen kuluvasta ajasta menemään sisimpään toistorakenteeseen, joten sen muuttujille voidaan antaa prioriteetti rekisteriallokointia tehdessä. Mikäli kaikki rekisterit ovat jo varattuja, pitää jonkin rekisterin sisältö ottaa talteen muistiin ja vapauttaa uutta käyttöä varten. Rekisterien arvojen talteenotto ja vapauttaminen uutta käyttöä varten on suorituskyvyn näkökulmasta epä-

toivottu toimenpide, ja tämän toimenpideparin käyttöä tulisi pitää mahdollisimman vähäisenä. (1, s. 554.)

x86-64-arkkitehtuuria koskevat erityishuomiot

CISC-käskykantaan perustuvana x86-64-arkkitehtuuri tarjoaa useita keinoja muistiosoituksiin, rikkaan komentokannan, kaksiosoitteisia käskyjä ja vähän rekistereitä, jotka on jaoteltu käyttötapausten mukaan. Lisäksi osa käskyistä on omistettu tiettyjen rekisterityyppien käytettäväksi.

CISC-käskykannan suunnitteluperiaatteiden mukaisesti x86-64-arkkitehtuurin assembly-kieli on suhteellisen abstraktia RISC-käskykantaan verrattuna eikä käskyjen odoteta suoriutuvan välttämättä yhden kellojakson aikana. (39.)

x86-64-arkkitehtuuri sisältää paljon historian perintöä, sillä se on edennyt evoluution kautta x86-arkkitehtuurista, jonka juuret juontavat 1970-luvulle. Näin ollen x86-64-arkkitehtuuri sisältää täydellisen binääriyhteensopivuuden aina 16-bittisiin ohjelmiin saakka, vaikka esimerkiksi uusimmat 64-bittiset Windowsit sisältävät tuen vain 32-bittisille ohjelmille. (40, s. 11–13; 41). Samaisista historiallisista syistä x86-64-arkkitehtuuri sisältää useita suoritustasoja, jotka on edelleen jaettu alatasoiksi. Tasoilla määritellään muun muassa. käytettävissä olevien muistiosoitteiden, operandien, rekistereiden ja vastaavien kokoja, muistimalleja sekä ohjelmien muistiavaruuden kirjoitus- ja lukuoikeuksia. (4, 3-1 – 3-24 Vol. 1)

4.3 Kääntäjäoptimoinnit

Hyödyt

Kääntäjäoptimointien suurimmat hyödyt ovat niiden tuoma suorituskykylisä ohjelmasuoritukseen, muistinkäytön pienentyminen, tuotetun ohjelmakoodin pienentyminen ja mahdollinen virrankulutuksen vähentyminen verrattuna ohjelmakoodiin, joka luotaisiin lähdekoodista sellaisenaan. Kääntäjien suorittamat optimoinnit ovat liki välttämättömiä, jotta korkean tason ohjelmointikieliä voidaan käyttää. Ensimmäistä Fortran-kääntäjää kirjoitettaessa suurin osa kääntäjän tekoon menneestä ajasta johtui optimointimene-

telmien etsimisestä, sillä projektin katsottiin epäonnistuvan, mikäli kääntäjän tuottama koodi olisi ollut 50 % hitaampaa kuin käsin kirjoitettu assembly. (6.)

Optimointivaiheen voidaan siis katsoa olevan liki välttämätön jokaiselle korkean tason kääntäjälle ja tulkille, mikäli suorituskyvyn halutaan olevan hyvä. Esimerkiksi työn pohjana toimiva teos ”Compilers Principles, Techniques & Tools” käyttää noin puolet sivumäärästään kääntäjien optimointivaiheelle.

Optimointitekniikat

Koodioptimoinnit voidaan jakaa peruslohkon optimointeihin ja globaaleihin optimointeihin. Peruslohko-optimoinnit optimoivat yksittäisiä koodirivejä, joihin on vain yksi tulopiste (rivi ei siis ole minkään hyppykäskyn kohde tms.) ja yksi poistumispiste. Globaalit optimoinnit vuorostaan optimoivat peruslohkojen välillä tapahtuvaa tietovirtaa ja tietoa (1, s. 533). Optimointivaihe ei saa muuttaa ohjelman semanttista merkitystä (1, s. 584). Optimointivaiheen tulokset ovat vahvasti riippuvaisia siitä, mitä halutaan optimoida ja mille alustalle optimoitua koodia tuotetaan, sillä eri alustoilla on omat vahvuutensa ja heikkoutensa.

Moni peruslohko-optimointien käyttämistä tekniikoista vaatii peruslohkojen muuntamisen DAG-esitykseksi. DAG-esityksestä voidaan poistaa kuollut koodi, jossa lasketaan arvoja, joita ei ikinä käytetä, toistuvat lausekkeet, joiden tulos on jo tiedossa, uudelleen järjestellä toisistaan riippumattomia lohkoja siten että väliaikaismuuttujien rekisteritarve on lyhyempi, ja uudelleen järjestellä kolmiosoitekoodi siten että laskuoperaatioista tulee mahdollisesti yksinkertaisempia. (1, s. 533.)

Globaalit optimoinnit perustuvat pääasiassa ohjelman tietovirtojen analysointiin käytettyihin algoritmeihin. Globaalit optimoinnit tekevät osittain samoja asioita kuin peruslohkon optimoinnit, mutta eri tasolla. Kuten peruslohko-optimoinnit, myös globaalit optimoinnit pyrkivät ohjelman tasolla poistamaan kuollutta koodia ja lausekkeita, joiden arvot ovat tiedossa. Niiden lisäksi globaalin optimoinnin tehtäviin kuuluu muun muassa toistorakenteiden ja taulukoiden indeksoinnin optimointi (1, s. 583–586.)

Kääntäjäoptimointien tuottama koodi ei ole koskaan optimaalista, sillä täydellistä kääntäjää ei voi kirjoittaa (30, s. 383–384). Lisäksi ohjelman käännökseen kuluvan ajan tulee olla kohtuullinen, mikä osaltaan rajoittaa optimointivaiheeseen käytettävissä olevaa

aikaa, joka yhdistettynä NP (nondeterministic polynomial time) luokan ongelmiin edelleen hankaloittaa optimointivaiheen työtä. Lopputulemana optimoitu koodi on aina kokoelma kompromisseja.

5 Kääntäjäni

5.1 Perustiedot

Työn tavoitteena on edellistä lukujen pohjalta kirjoittaa työtä varten luodulle ohjelmointikielelle toimiva kääntäjän front-end. Front-endin on siis tarkoitus muuntaa luettu lähdekoodi kolmiosoitekoodiksi.

Kääntäjäni on rajattu käsittämään pelkästään kääntäjän front-endiin työn laajuuden vuoksi. Työssä tehty ”kääntäjä” ei siten sisällä konekielisen koodin generointia eikä optimointivaihetta. Sen sijaan siihen sisältyvät esitellyn kääntäjäarkkitehtuurimallin mukaisesti leksikaalinen analyysi, semanttinen analyysi ja välikoodin generointi.

Kääntäjää varten loin VIC-20-tietokoneen BASICista vaikutteita ottaneen kielen. Kieli on suunnittelun puolesta Turing-yhteensopiva ja sisältää toisto- ja ehtorakenteet, hyppykäskyt ja aliohjelmat. Muuttujatyyppejä on neljä, ja kaikki muuttujat ovat näkyvyysalueiltaan globaaleja, eli yksi muuttujanimi viittaa läpi ohjelman samaan muistipaikkaan.

Tarkemmin kielen suunniteltuihin ominaisuuksiin voi tutustua liitteessä 1.

5.2 Kohdealusta

Kääntäjäni on tarkoitettu toimimaan 64-bittisellä Windows 10 pro -käyttöjärjestelmällä, joka on Microsoftin pitkän käyttöjärjestelmätuotehistorian uusin tuotos. Käyttöjärjestelmä esitettiin syyskuussa 2014 ja julkaistiin markkinoille heinäkuussa 2015. Käyttöjärjestelmän markkinaosuus tammikuussa 2017 kannettavista tietokoneista ja pöytätietokoneista oli w3schoolsin mukaan 31,7 % (42) ja 25,3 % (43) Netmarketsharen mukaan.

Pro-version huomattavimpia eroja tavalliseen kuluttajaversioon verrattuna (Windows 10 Home) ovat Windows Prohon kuuluvat erinäiset tietoturvaominaisuudet, Pron kyky suuremman muistiavaruuden käsittelyyn (128 gigatavua vrt. 2 teratavua) ja Prohon kuuluva Hyper-V-virtualisointiteknologian tuki (44).

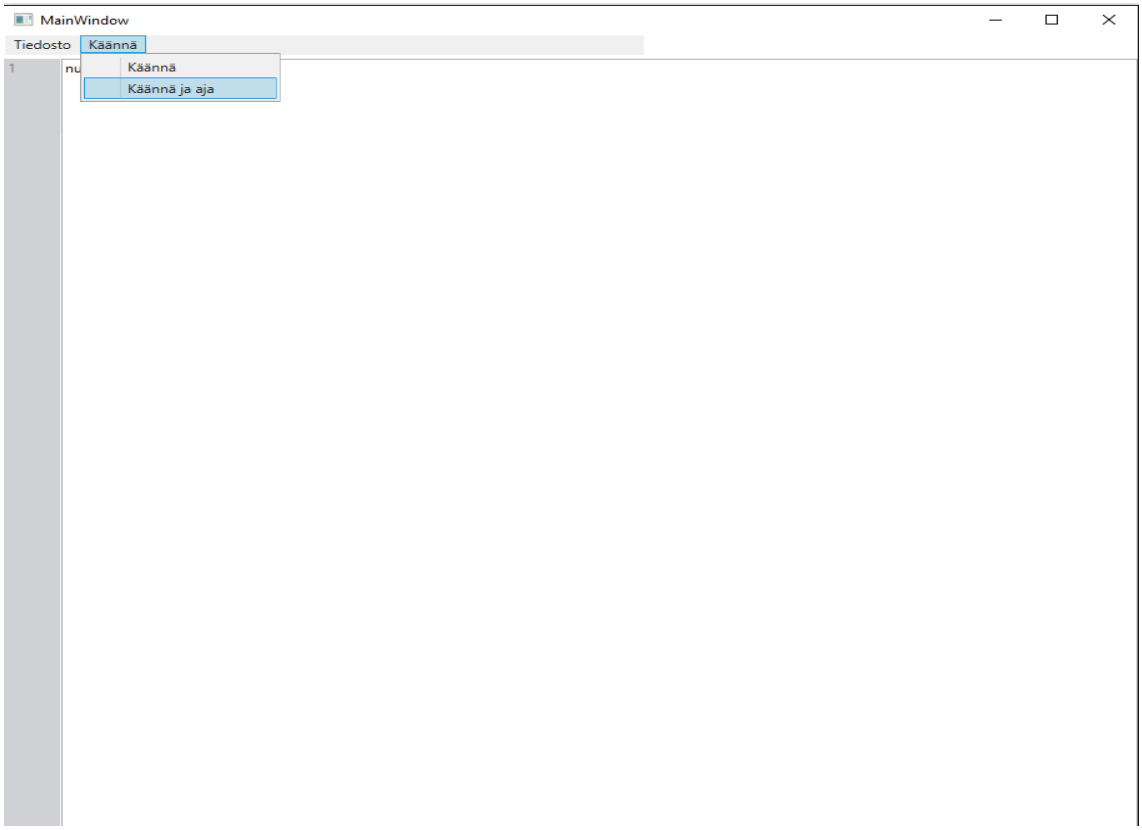
Käyttöjärjestelmä, ja sen myötä myös kääntäjäni, toimii x86-64-arkkitehtuuriin pohjautuvalla Haswell-E-arkkitehtuurin i7-5820K-suorittimella. Suoritin julkaistiin elokuussa 2014 korvaamaan aikaisemmat Ivy Bridge-E -arkkitehtuurin prosessorit. Suoritin koostuu 6 fyysisestä ytimestä, jotka kykenevät Intelin Hyper-Threading-ominaisuuden vuoksi suorittamaan 12 samanaikaista säiettä kerrallaan. Suoritin koostuu 2,6 miljardista transistorista, ja ilman ylikellotusta se toimii maksimissaan 3,6 gigahertsin taajuudella. Kääntäjien näkökulmasta Haswell-E-arkkitehtuuri ei sisällä mitään uutta, sillä sen sisältämät käskykantalaajennukset ovat jo 2013 julkaistusta Haswell-arkkitehtuurista. Näitä ovat mm. tuki AVX2 (Advanced Vector Extensions 2) käskykantalaajennukselle sekä uudet bittimanipulointikäskyt. (45; 46.)

5.3 Toiminnan kuvaus

Tehty kääntäjän front-edin rakenne noudattaa pitkälti teosten ”Compilers Principles, Techniques, & Tools” sekä ”Modern compiler implementation in C” käyttämää rakennetta, sillä ne olivat pääasialliset kirjalliset lähteet työlleni.

Käännöksen alku

Käännösprosessi alkaa painamalla aja ja käännä -nappia (kuva 1) kirjoittamassani tekstieditorissa, jonka osaksi kääntäjän front-end on integroitu.



Kuva 1. Insinöörityönä tehdyn kääntäjän tekstieditori ja käännösprosessin aloittava Käännä ja aja -nappi.

Napin painallus lukee massamuistiin tallennetun .txt-muotoisen tiedoston. Kääntäjän front-endin prosessoimaa kieltä varten ei siis luotu omaa tiedostopäätettä, ja kieli on siten kirjoitettavissa käytännössä missä tahansa tekstieditorissa, mutta kielellä kirjoitettu tekstitiedosto pitää aina avata editorilla, johon kääntäjän front-end on integroitu. Luettu teksti erotellaan omaksi tietorakenteeksi, jossa yhtä lähdekoodiriviä vastaa yksi objekti. Tämän tietorakenteen ominaisuuksia ovat sisältö eli riviltä löytynyt teksti ja rivinumero. Näistä tietorakenteista tehty lista annetaan leksikaaliselle analyysille.

Leksikaalinen analyysi

Leksikaalinen analyysi saa parametrina listan koodirivejä kuvaavia tietorakenteita. Näistä tietorakenteista leksikaalinen analyysi erottelee lekseemejä, mutta ennen sitä jokainen koodirivi-tietorakenteen sisältö-kenttä muutetaan yksiulotteiseksi merkkijonotaulukoksi, jonka perään lisätään C-kielestä tuttu ’\0’-merkki. Tämä merkki C:ssä on merkkijonotaulukon päättävä null-merkki, jota C#:ssa ei käytetä merkkijonotaulukoiden päättämiseen. Leksikaalinen analyysi oli yksi niistä vaiheista, jotka ehdin kirjoittamaan

C:llä ennen projektin muuntamista C#:lle, joten minun oli helpompi lisätä merkki merkkijonotaulukon perään kuin kirjoittaa kaikki leksikaalisen analyysin vertailut uudestaan.

Kuten koodiriveillä, on myös lekseemeillä oma tietorakenteensa. Lekseemien tietorakenteeseen kuuluvat tiedot koodirivistä, jolla lekseemi sijaitsee, lekseemin tyyppi (eli tieto siitä, onko lekseemi varattu sana, arvo, muuttujanimi jne.) ja string-muotoinen tietokenttä, johon tallennetaan tieto lekseemin arvosta tai nimestä, jos lekseemi on arvo tai muuttujanimi-tyyppinen.

Lekseemien analysointi ei sisällä minkäänlaista kieliopillista validointia, mutta käännösprosessi voi loppua jo leksikaalisen analyysiin, mikäli analyysivaihe löytää merkkejä, joita kieleni ei tue. Näitä merkkejä ovat muun muassa kaikki englannin kielen aakkosista puuttuvat merkit kuten skandinaaviset ä, ö ja å ja muissa ohjelmointikielissä yleinen komennon päätösmerkki ;. Tosin nämä merkit ovat käytettävissä string- ja char-tietorakenteissa. Myös pisteen löytyminen float-, char- ja string-tietotyypin ulkopuolelta aiheuttaa käännösvirheen.

Lekseemit erotellaan toistorakenteessa, jonka sisällä on yksi switch-case-rakenne. Rakenteessa jokaisen varatun sanan ensimmäistä kirjainta kohden on case, joissa tarkistetaan, muodostavatko seuraavat merkit varatun sanan vai jotain muuta. Erottelu noudattaa luvussa 3.1 mainittua sääntöä, jonka mukaan pisin kielioppisääntöjen mukainen alimerkkijono muodostaa lekseemin.

Koodirivien erottelun tapaan myös lekseemit kerätään listaan. Koska kääntäjässäni ei ole jäsennysvaihetta, lähetetään lekseemi-tietorakenteita sisältävä lista sellaisenaan parametrina semanttiselle analyysille.

Virhetilanteessa leksikaalinen analyysi suorittaa goto-käskyllä siirtymisen normaalin ohjelmasuorituksen ulkopuolelle. Havaitusta virheestä lähetetään tieto ErrorMessageHandler-luokalle, joka tulostaa virhetyyppiin liitetyn virheviestin. Virheenkorjauksen puuttumisen vuoksi kääntäjän front-end lopettaa käännösprosessin tähän.

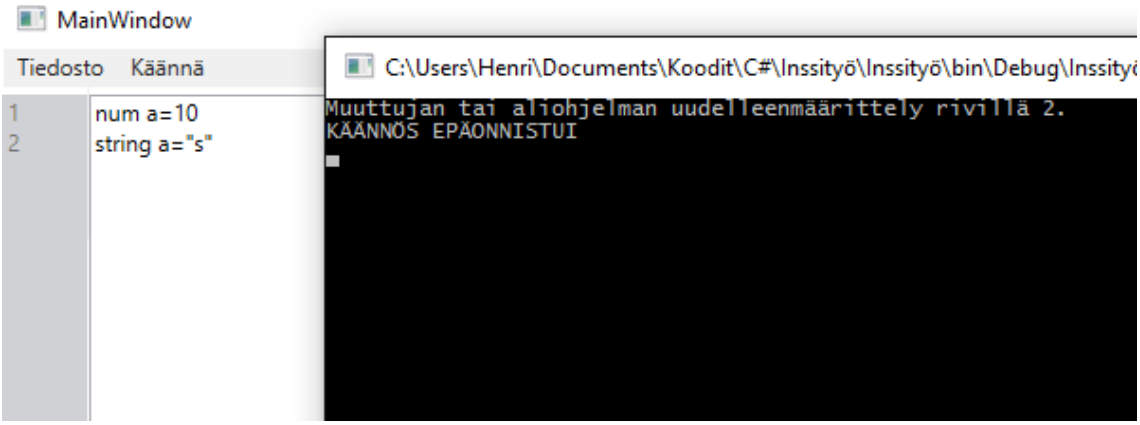
Tekemäni leksikaalinen analyysi on toiminnaltaan äärellinen deterministinen automaatti, sillä jokaisen merkin aiheuttama tila on ennakkoon tiedossa ja tiloja on rajallinen määrä.

Semanttinen analyysi

Kääntäjäni semanttinen analyysi saa parametrinaan listan lekseemi-tietorakenteita. Semanttista analyysia kuvaavan luokan konstruktori sisältää kutsut kolmen apuluokan luontiin, joista edelleen kaksi tarvitsee toimintaansa lekseemilistaa. Toinen näistä luokista avustaa kielioppitarkistusta tarkastamalla, onko indeksin x, jossa x [0, lekseemilistan pituus[, lekseemi mahdollisesti haluttua tyyppiä. Toinen apuluokista on laajempi apuluokka, joka sisältää laskentaan käytettäviä avustuksia.

Itse analyysivaiheessa koodi kerää talteen muuttujat yhteen hajautustaulurakenteeseen, jossa avaimena toimii muuttujan nimi ja arvona tietorakenne, joka sisältää tiedot muuttujan tyypistä, nimestä, arvosta, yleisistä käyttökerroista, käyttökerroista toistorakenteessa ja rivistä, jolla muuttuja esiteltiin. Kirjanpito käyttökerroista on kääntäjän nykytilassa tarpeeton, mutta tulevaisuudessa niitä on tarkoitus käyttää, kun muuttujan tarvetta rekisteripaikalle harkitaan.

Toiseksi analyysivaiheessa tarkistetaan, että lekseemit seuraavat toisiaan kieliopillisesti oikeassa järjestyksessä. Mikäli lekseemien järjestyksen havaitaan rikkovan kielioppisääntöjä, katkeaa analysointivaihe välittömästi. Järjestysrikkomuksella tarkoitetaan tässä tilanteessa esimerkiksi tapauksia, joissa vääräntyyppisiä muuttujia tai vakioarvoja yritetään sijoittaa muuttujiin tai vääränlaiset käskyt seuraavat toisiaan, kuten vaikkapa sijoitus, jossa sijoitettavan arvon tai muuttujan sijaan yritetään suorittaa aliohjelmakutsu. Näissä tilanteissa ohjelmasuoritus hyppää goto-käskyllä normaalin ohjelmasuorituksen ulkopuolelle kutsumaan virheentulostusluokkaa havaitulla virhetyypillä. Malliesimerkkinä tästä on kuvan 2 tilanne.



Kuva 2. Esimerkki tilanteesta, jossa kääntäjä havaitsee virheen ja lopettaa käännösprosessin virheviestin kera. Tässä tapauksessa on yritetty määritellä uudestaan muuttuja a.

Kolmanneksi analyysi luo hajautustaulu-tietorakenteen, jossa avaimena toimii juokseva int-muuttuja ja arvona nelikkomuotoinen kolmiosoitekoodia kuvaava tietorakenne. Tietorakenteen kenttiä ovat tulos, operaattori, argumentti 1 ja argumentti 2. Kaikki kentät ovat object-tyyppisiä, ja siten niihin voidaan tallentaa mitä tahansa perustietotyyppejä. (47.) Int-tyyppiselle avaimelle ei nykyisellään ole mitään käyttöä, mutta tulevaisuuden ohjauskäskyjen on tarkoitus perustua niihin. Nykytilassa semanttinen analyysi osaa erotella aliohjelmat lekseemilistalta ja lisäksi se osaa tarkistaa, että kaikille hyppykäskyille löytyy osoite, mutta niitä ei hyödynnetä missään eikä niiden toiminnan oikeudesta ole tietoa. Lisäksi ehtorakenteiden toteutus puuttuu täysin. Analyysivaiheen suuri switch-case-rakenne sisältää caset ehdoille, mutta ne eivät toistaiseksi sisällä minkäänlaista implementaatiota.

Semanttisen analyysin toiminta perustuu myös yhden toistorakenteen sisällä olevaan switch-case-rakenteeseen. Siinä missä leksikaalisen analyysin switch-case-rakenteen caset rakentuivat yksittäisiin ohjelmointikielen tukemiin merkkeihin, perustuvat semanttisen analyysin caset käsittelyssä olevan lekseemin tyyppiin. Jokainen (toteutuksen sisältävä) case sisältää jonkinlaisen mielivaltaisen määrän tarkistuksia ja toimenpiteitä kielioppisääntöihin ja seuraaviin lekseemeihin perustuen. Semanttisen analyysin kolme tehtävää on toteutettu sekaisin pitkin switch-case-rakennetta aina casen tarpeiden mukaisesti. Toteutuksen ongelmiin palataan luvussa 5.4.

Analyysivaiheen aikana kerätyt kaksi hajautustaulukkoa paketoidaan C#:n Tuple-rakenteeseen, minkä asioista analyysimetodi palauttaa käytännössä kaksi arvoa. Nä-

mä kaksi hajautustaulua lähetetään seuraavaksi parametrina käännetyn koodin suoritusvaiheeseen.

Leksikaalisen analyysin tapaan myös semanttinen analyysi hyppää virheitä kohdatessa goto-käskyllä normaalin ohjelmasuorituksen ulkopuolelle ja lähettää samaan tapaan ErrorMessageHandler-luokalle parametrina havaitun virhetyypin. Saadun tyypin perusteella ErrorMessageHandler-luokka antaa tulosteen havaitusta virheestä. Virheenkorjauksen puuttumisen vuoksi tehdyn front-endin suoritus päättyy tähän. Koska suoritus lakkaa heti ensimmäisen virheen kohdalla, ei lähdekoodin muista virheistä tiedetä vielä tässä vaiheessa.

Yksinkertaisuuden nimissä ja aktiivisten ympäristöjen lukumäärän (tasan yksi) vuoksi semanttinen analyysini sisältää vain yhden symbolitaulun. Luvussa 3.1 mainituista imperatiivisesta ja funktionaalisesta toteutuksesta edustaa symbolitaulun toteutus imperatiivista, eli tauluista löytyvät arvot korvataan päivitystilanteessa uusilla.

Vaikka kääntäjän front-endin suoritusvaiheen toteutuksesta jäi puuttumaan kontrollirakenteiden tukeminen (ja osittain myös semanttisesta analyysista), se sisältää silti eräänlaisen backpatching-vaiheen. Jokaisesta tavatusta hypystä luodaan uusi merkintä symbolitauluun. Ennen semanttisesta analyysista paluuta tarkistetaan, että kaikille hyppykäskyille on varmasti olemassa osoite, tässä tapauksessa aliohjelma.

Välikoodiesityksen suoritusvaihe

Sen sijaan, että kääntäjäni loisi suoritettavan konekielisen sovelluksen, tulkitaan suoritusvaiheessa semanttisen analysointivaiheen aikana kerättyjä hajautustauluja. Suoritus tapahtuu yhdessä toistorakenteessa, jossa käydään läpi hajautustaulussa olevia kolmiosoitekoodin tietorakenteita läpi. Se, mitä käsiteltävänä oleva kolmiosoitekoodi tekee, riippuu sen operaattori-kentästä. Operaattori-kenttää käytetään suorituksen avaimena, jotta suoritus ohjautuu oikean apuluokan käsiteltäväksi.

Tehdyn front-end-ratkaisun ja käytetyn back-end-ratkaisun vuoksi kääntäjäkokonaisuus on JIT-tyyppinen. Toisin kuin puhtaasti tulkatuissa ratkaisuissa, käy tehty kääntäjän front-end suoritettavan ohjelmakoodin kokonaisuudessaan läpi tarkastaen sen semanttisen eheyden ja valmiiden kirjastokutsujen sijaan ohjelman suoritus perustuu välitason koodiesityksen tulkintaan. Toisaalta AOT-ratkaisussa olisi olemassa jonkinlainen back-

end, joka tekee kerätystä abstraktista välikoodista konekielisen version. Tätä vaihetta ei tässä työssä ole tehty. Sen sijaan toteutettu kääntäjän front-end kääntää luodun kolmiosoitekoodiesityksen C#-muotoon, joka edelleen käännetään C#-kääntäjän toimesta tavukoodiesitykseksi. Tämä CIL-muotoinen esitys käännetään edelleen ajon aikana valitun ympäristön konekieliseksi esitykseksi.

Semanttisen analyysin tapaan myös suoritusvaihe käyttää toiminnassaan apuluokkia. Suoritusvaiheen apuluokkien käyttö liittyy muuttujien arvojen ylläpitoon, jotta niiden logiikkaa ei sekoiteta itse suorituksen logiikkaan ja arvojen tulostamiseen konsoliin.

Apuluokat

Tehty kääntäjän front-end sisältää joitain apuluokkia, joiden tarkoitus on eriyttää käännösprosessin vaatimien toimenpiteiden toiminnallisuus irti prosessivaiheen ydintoiminnasta. Tällä tavoitellaan hyvän ohjelmointitavan mukaista toimintalogiikan pilkkomista mahdollisimman pieniksi yksiköiksi, millä voidaan minimoida private-näkyvyysalueiden metodien käyttö ja maksimoida yksikkötestauksen potentiaalinen testikate.

Laajimmin käytössä oleva apuluokka liittyy lekseemien järjestyksen kieliopilliseen eheyteen. Luokka ottaa konstruktorissa parametrina listan löydetyistä lekseemitietorakenteista ja tarjoaa ulospäin yhden public-metodin, joka vastaanottaa ensimmäisenä parametrina taulukon sallituista lekseemeistä ja toisena parametrina indeksin, josta lekseemin tyyppiä halutaan tutkia.

Toinen tarkistuksiin liittyvä apuluokka liittyy tyyppien tarkistamiseen. Luokka on tyypiltään staattinen, sillä se ei sisällä mitään ajonaikaista tietoa, joka voisi muuttua ja sen toiminta on aina siten kaikille sama. Toisin sanoen, luokalla ei ole mitään ulkopuolisia riippuvuuksia, jotka vaatisivat uusien instanssien luontia tilanteesta riippuen. Kyseinen apuluokka sisältää kolme public-tyyppistä staattista metodia, jotka tarkistavat, onko niille syötetty muuttuja kokonaisluku, liukuluku tai yksittäinen merkki.

Kolmas apuluokka on apuluokista suurin, ja se kokoaa loput kääntäjän käyttämät apuluokat yhteen. Yhteen koottuja apuluokkia voi tarpeen tullen käyttää myös yksittäin, mutta kokoava apuluokka tarjoaa näiden kaikkien yksittäisten luokkien toiminnan yhden metodin sisällä, mikä säästää koodirivejä toisaalla ja on aina ylläpidettävyyden näkö-

kulmasta etu. Nämä kaikki apuluokat liittyvät kielen tarjoamiin matemaattisiin operaatioihin lukuun ottamatta jakojäännöstä, joka jäi implementaation ulkopuolelle.

Laskusuoritusten apuluokat syntyivät rekursion tarpeesta, sillä ennen niitä semanttinen analyysi liikkui goto-käskyillä kääntäjän lähdekoodia lukijan näkökulmasta koodia alhaalta ylös. Tätä rakennetta kutsutaan spagettikoodiksi, jota kaikessa ylläpidettäväksi aiotussa koodissa pyritään välttämään, sillä kuten nimestä voi päätellä, koodi hyppii paikasta toiseen mielivaltaisesti tehden suorituksen seurannasta hankalaa tai mahdotonta. Rekursion tarve vuorostaan syntyi tapauksesta, jossa yhdellä käskyrivillä laskettiin useampia arvoja yhdessä, esim. a = 1 + 2 + 3 + 4 + 5.

5.4 Huomiot kääntäjän lähdekoodista

Tämän luvun tarkoitus on tarkastella toteutetun kääntäjän front-endin ja sen yhteydessä tehdyn tekstieditorin toteutusta ohjelmistotuotannon näkökulmasta. Luvulla ei ole yhteyksiä muuhun työhön, mutta mielestäni se on perusteltu koulutusalan työelämänäkökulman painotuksen vuoksi.

Rakenne

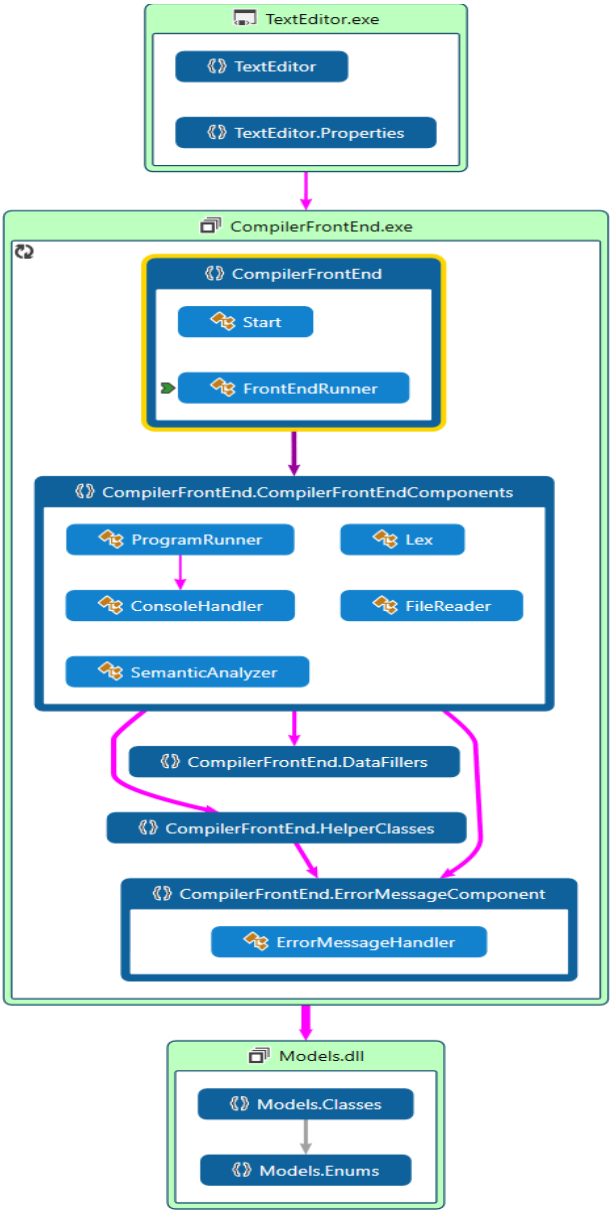
Kääntäjä on kirjoitettu kokonaisuudessaan C#:lla, joka on Microsoftin kehittämä moniparadigmainen ohjelmointikieli ja sisältää muun muassa olio-, imperatiivisen ja funktionaalisen ohjelmoinnin piirteitä ja ominaisuuksia.

Tekstieditorin ja kääntäjän arkkitehtuuri on varsin yksinkertainen. Kerätty tieto liikkuu vain yhteen suuntaan, eli kutsuvaa luokkaa ei koskaan kysellä kutsuttavassa luokassa ja tasoissa liikesuunta on samoin yksisuuntainen, eli esimerkiksi kun tekstieditorissa kerran tiedostosta talteen luettu teksti lähetetään tehdylle kääntäjän front-endille, se ei enää koskaan kysele mitään lisätietoja tekstieditorilta, vaan tulee toimeen saadulla tiedolla.

Kääntäjän front-endin käyttämät tietorakenteet ovat eriytetty omaksi irralliseksi tasokseen ohjelman suorituslogiikasta. Tietorakenteita kuvaavat luokat eivät sisällä minkäänlaista logiikkaa vaan pelkästään kyseisen tietorakenteen tarvitsemat tietokentät. Tietokentät täytetään aina jossain kohtaa ja luetaan seuraavassa kohdassa, mutta ker-

ran asetettua tietoa ei muokata ajon aikana. Tietorakenteita voidaan siis ajatella read-only-tyyppisinä ratkaisuina, joihin sallitaan kirjoitus kerran mutta sen jälkeen niitä on tarkoitus vain lukea.

Työssä käytetyn kehitysympäristön näkökulmasta tietorakenteiden, kääntäjän front-endin ja tekstieditorin eriyttäminen näkyy siten, että kaikille edellä mainituille on oma projektinsa. Lisäksi projektien sisällä lähdekoodit on eroteltu toisistaan omiin kansioihinsa aihealueen mukaan (kuva 3).



Kuva 3. Visual Studiolla luotu kartta tekstieditorin ja kääntäjän front-endin nimiavaruuksien ja tekstissä mainittujen luokkien suhteista toisiinsa. Violetti nuoli kuvaa kutsuja ja harmaa nuoli palautuksia. Nuolen suunta kuvaa violetissa tapauksessa kutsuttavaa luokkaa tai nimiava-

ruutta ja harmaan tapauksessa palautettavaa luokkaa tai nimiavaruutta. Kuva ei näytä kaikkia ohjelmassa tapahtuvia kutsuja.

Ohjelman rakennetta ajatellen huomattavan heikkona ratkaisuna pidän ErrorMessageHandler-luokkaa, joka on tyypiltään staattinen yhden metodin suurehko switch-case-rakenne, joka vaatii aina uuden casen jokaista uutta virhettä kohden. Mahdollisesti elegantimpi ratkaisu olisi ollut lähettää Exception-luokasta periytyvän luokan instanssi, johon virheviesti olisi paketoitu ja vastaanottaa tämä virhe ErrorMessageHandler-luokassa, jossa olisi tulostettu virheeseen paketoitu viesti. Näin luokan ei tarvitsisi arvuutella mikä virhe on kyseessä ja etsiä tulostetta eri vaihtoehdoista. Nämä ovat kuitenkin jo luokkaa kutsuessa tiedossa.

Valitettavasti kääntäjän front-endin lähdekoodin rakenne jättää paljon toivomisen varaa. Sekä leksikaalinen että semanttinen analyysi sisältävät switch-case-rakenteen sisällä vielä ehtolauseita, mikä on ylläpidettävyyden, testattavuuden ja laajennettavuuden näkökulmasta erittäin haasteellista. Lisäksi kummankin komponentin switch-case-rakenteet ovat laajoja, mikä ei helpota testaamisen vaikeutta millään tasolla. Valitun rakenteen ongelmia voisi mahdollisesti helpottaa leksikaalisen analyysin tapauksessa luomalla jokaista switch-case-rakenteen casea varten oman luokan, jonka sisällä tapahtuisi nyt casen sisällä tehtävät päättelyt. Vaikka tämä johtaisi luokkien määrän huomattavaan kasvuun, olisivat testattavat kokonaisuudet pienempiä ja paremmin hallittavissa. Lisäksi näiden luokkien laajentaminen on virheiden välttämisen näkökulmasta helpompaa kuin casen sisäisen ehtorakenteen laajentaminen tai niiden lisääminen. Laajojen ehtorakenteiden lisäksi koodipohjaa heikentää laiskasti ohjelmoitu ohjausrakenne, jossa paikasta toiseen siirrytään retrohenkisesti goto-käskyllä. Tilanne tosin tältä osin ei ole niin paha, kuin käskyn osalta voisi olettaa, sillä hypyt tapahtuvat lähdekoodin lukusuuntaan (vasemmalta oikealle, ylhäältä alas) eikä koskaan sitä vastaan, joten hyppyjä on ohjelmoijan näkökulmasta melko vaivatonta seurata. Lisäksi goto-käskyä käytetään vain virhetilanteissa, jolloin sillä hypätään normaalin ohjelmasuorituksen ulkopuolelle ja tilanteissa, joissa havaitaan merkkijonon muodostavan varatun sanan sijaan muuttuja- tai aliohjelmanimeä.

Ylläpidettävyys ja laajennettavuus

Sekä leksikaalinen että semanttinen analyysi tehdään yhdessä suuressa rakenteessa, jonka luettavuus on heikko ja sitä myötä riski rikkoa jotain uusia ominaisuuksia kieleen lisätessä on suuri. Heikko luettavuus tekee laajentamisesta myös siinä mielessä han-

kalaa, että ohjelmoijan on hankalahko löytää sitä kohtaa, johon laajennuksen vaatimaa uutta koodia voi tuottaa.

Näitä kahta puutetta lukuun ottamatta koodipohja on kohtuullisessa kunnossa. Lähdekoodin lukeva luokka, front-endin eri toiminnollisuuksia kutsuva luokka ja pienet apuluokat ovat yksinkertaisia eivätkä sisällä paljoa koodia. Toisaalta näiden luokkien tehtävä on varsin suoraviinainen, mikä pitää niiden rakenteen yksinkertaisena verrattuna kahteen edellä mainittuun ongelmatapaukseen. Erityisesti tekstin luku on sellainen operaatio, minkä C# hoitaa liki tulkoon itsekseen, kuten ohjelmoija haluaa (toisin kuin esimerkiksi C:ssä, jossa manuaalinen muistinhallinta pakottaa kirjoittamaan huomattavat määrät muistinhallintaa koskevaa koodia tiedostonluvun ympärille).

Tulevaisuudessa, mikäli aion jatkokehittää tehtyä kääntäjän front-endiä ja tekstieditoriani, ongelmia tulee luultavasti aiheuttamaan tekstieditorin koodi. Nykyisellään kaikki käyttöliittymän toiminnot ovat yhdessä luokassa, mikä tekee tekstieditorin toiminnallisuuden laajentamisesta jossain määrin hankalaa. Käyttöliittymän eri komponentit tulisi jotenkin eriyttää omiksi luokikseen ja niiden toimintalogiikkaan liittyvä koodi siirtää kyseessä oleviin luokkiin, jotta niiden toimivuus voitaisiin testata järjellisellä työmäärällä.

Samalla implikaatiolla joutunen myös eriyttämään eri tietotyyppeihin liittyvät validoinnit ja alustukset. Nykyisellään kaikkiin tietotyyppeihin liittyvät validoinnit ovat yhdessä luokassa ja kaikkiin tietotyyppeihin liittyvät alustukset ovat yhdessä luokassa. Tämä toimii ja on luettavaa niin kauan, kuin tietotyyppien lukumäärä on todella rajallinen, mikä onkin tällä hetkellä tilanne kielessäni.

Olemassa olevan koodin laajentamisen hankaluuden vastapainoksi kääntäjän front-endin laajentaminen uusilla vaiheilla on helppoa. Riittää, että luo CompilerFrontEnd-projektin CompilerFrontEndComponents-kansioon uuden luokan, jossa optimitilanteessa kuvataan prosessin toiminta päätasolla. Mikäli arkkitehtuuria ei muuta, näkyy uusi vaihe ProgramRunner-luokassa yhtenä koodirivinä, jossa tapahtuu kutsu uuteen vaiheeseen.

Muita kyseenalaisia ja laajennettavuuteen haastetta tuovia ratkaisuja ovat lekseemitietorakenteen ja muuttujien nimet tyyppiin sitovan tietorakenteen keskenään ristiriitaiset tyyppimerkinnät. Lekseemillä tyyppimerkinnät Char, Integer, Float tai String tarkoittavat, että lekseemi on varattu jokin neljästä varatusta sanasta ja tyyppimerkintä Value

on jokin arvo. Muuttujanimet tyyppeihin sitovissa rakenteissa vastaavat merkinnät tarkoittavat, että jokin muuttuja on tyyppiä char, integer. float tai string. Value-tyyppi ei ole enää sidosrakenteissa käytössä. Koska sitomisvaiheessa käytetään sekaisin lekseemejä ja sidosrakenteita, on tyyppimerkinnän kaksoismerkityksellisyys virhealtis ratkaisu.

Testattavuus

Tätä lukua on hyvä alustaa sillä, että tarkastelen kääntäjän testattavuutta yksikkötestaamisen näkökulmasta white box testauksena. White box -testaus tarkoittaa, että testaajalla on pääsy ohjelman lähdekoodiin. Sen vastakohtana toimii black box -testaus, jossa testaaja joutuu kirjoittamaan testinsä julkista rajapintaa vasten näkemättä ohjelman toimintaa lähdekooditasolla.

Kääntäjän ja tekstieditorin yksikkötestattavuus on kaikkinensa melko heikolla tasolla. Yksikkötestauksen näkökulmasta ongelmallisia tapauksia ovat jälleen kerran kaksikko leksikaalinen ja semanttinen analyysi. Niiden ongelmallisuus johtuu yllä mainituista ongelmista eli heikosta rakenteesta.

Kääntäjän front-endin lisäksi tekstieditorin, johon front-end on integroitu, yksikkötestaaminen on ongelmallista, sillä kaikki käyttöliittymätapahtumiin liittyvät metodit ovat näkyvyydeltään private. Private-tyyppisiä metodeja ei periaatteessa pitäisi yksikkötestata, sillä niiden on tarkoitus kapseloida tieto siten, etteivät ulkopuoliset luokat tiedä eivätkä kykene vaikuttamaan kapseloituun tietoon. Yksikkötestit luonteensa puolesta rikkovat tätä periaatetta, sillä niissä on tarkoitus validoida käsiteltävän tiedon tilaa.

Yhtä kaikki, front-endin private-metodit tekevät käsiteltävälle tiedolle suorituskriittisiä toimenpiteitä, joten metodien yksikkötestaaminen olisi enemmän kuin mielekästä ja suotavaa. Vastaavaa private-metodien ”väärinkäyttöä” on myös erityisesti semanttisesta analyysistä, mutta vähemmässä määrin leksikaalisessa analyysissä. Siellä kyseisen näkyvyysluokan metodit suorittavat vähemmän kriittisiä toimintoja, kuten lekseemejä kuvaavan tietorakenteen luokasta instansseja tekevän luokan kutsuja. Tätä luokkaa voi kuvailla tehtaaksi, joka vastaanottaa tietotyyppejä ja palauttaa niiden pohjalta muodostettuja lekseemi-tietorakenteita. Kyseisen tehtaan yksikkötestaaminen on helppoa, sillä se sisältää vain yhden kolme riviä pitkän public-metodin, joka ei edes sisällä ehtolauseita.

Jos tosissani olisin yksikkötestaamassa tehtyä kääntäjän front-endiä ja tekstieditoriani, minua odottaisi huomattavan suuri uudelleenkirjoitustyö. Kahdesta analyysivaiheesta kumpikin pitäisi kirjoittaa huomattavassa määrin uusiksi, jotta ne olisivat testattavissa, mutta koska uskon joku päivä vielä laajentavani kehittämääni kieltä ja kääntäjää, on mainittu työ vain tehtävä, sillä se säästää monelta päänvaivalta ja suojaa selustan laajennustyössä todennäköisesti tapahtuvilta virheiltä jo olemassa olevaan koodiin.

Tietotyyppien toteutus

Kieleni tietotyypit on sisäisesti toteutettu C#:n vastaavilla: string on string, joka on keinotekoisesti rajoitettu historiallisista syistä (ks. luku 8) 160 merkkiin, dec on C#:n double, numia vastaa C#:n BigInteger-tietotyyppi ja char on char. Muita tietotyyppejä kieleni ei sisällä.

C#:n string-tyyppi muodostuu nollasta tai useammasta UTF (Unicode transformation format)-16-koodatusta char-tyypistä. Näin ollen C#:n char vastaa koodauksen puolesta täysin työn char-tyyppiä. Toisin kuin luomani 160 merkin raja, kulkee C#:n stringin rajoitus merkkien määrän sijaan 2 gigatavussa. (48.)

Sekä C#:n double että työssäni käytetty dec noudattavat samaa IEEE 754 -standardia eivätkä siten poikkea käytökseltään toisistaan.

C#:n BigIntegerin kokoa rajoittaa vain käytettävissä olevan muistin määrä. Kun se ylittyy, aiheutuu OutOfMemoryException, joka keskeyttää ohjelman suorituksen. Tehdyn kääntäjän front-end ei sisällä minkäänlaista toiminnallisuutta num-muuttujatyyppiä varten, vaan koko toteutus nojaa sellaisenaan C#:n BigIntegerin toimintaan.

Tulevaisuuden haasteisiin kuuluu, miten aion mallintaa uudet tietotyypit. Erityisesti luokkien tapaiset tietoa kapseloivat ja omaa tietojenkäsittelyä sisältävien rakenteiden implementoiminen on jäänyt tämän työn myötä selvittämättä, ja haluaisin ottaa enemmän selvää.

5.5 Tiedossa olevat virheet

Tiedossa olevia virheitä on jäänyt vain kääntäjän käyttämään tekstieditoriin. Itse kääntäjässä ei ole tiedossa olevia virheitä, mikä ei tietenkään tarkoita, etteikö siitä virheitä löytyisi.

Luultavasti näkyvin editorin virheistä on tekstin rivejä seuraava rivinumerointi. Lukuvaiheessa rivinumerointi hajoaa täysin eikä osaa näyttää rivinumeroita ensimmäistä riviä pidemmälle. Lukuvaiheessa on myös havaittavissa huomattavaa hitautta, jonka syytä en ole ehtinyt ottaa selville.

Toinen käyttäjälle hyvin näkyvä virhe on käännä-napin olemassaolo. Tällä hetkellä nappiin ei ole implementoitu minkäänlaista toiminnallisuutta. Nappia valikkoon lisätessä oli suunnitelmissa luoda käännösvaiheesta versio, jossa käännettävästä lähdekoodista luodaan suoritettavissa oleva tiedosto jonnekin, mutta sitä ei suoritettaisi välittömästi käännöksen jälkeen. Tämä vaihe jäi kuitenkin tekemättä.

Tekstieditori ei myöskään osaa seurata tekstin muutoksia kunnolla vaan kyselee turhaan, haluaako käyttäjä tallentaa tekemänsä muutokset. Ongelma ei ole suoritukselle haitallinen, mutta kuten jatkuvasti ruudulle hyppivät dialogit yleensä, on ongelma todella ärsyttävä.

6 Työprosessi ja johtopäätökset

6.1 Lähtökohdat

Insinöörityön lähtökohta oli oma mielenkiinto kääntäjiä kohtaan ja halu ymmärtää niiden toimintaa ja ohjelmointikielten käännösprosessia. Työllä ei missään vaiheessa ollut taloudellisia intressejä, enkä sen markkinallisen marginaalisuuden huomioon ottaen yrittänyt saada kaupallisia toimijoita työhön mukaan. Työn alkuperäisinä tavoitteina oli oman kääntäjän kirjoittaminen, käännösprosessin tutkiminen, kääntäjän lopputuotteen vertailu muihin kääntäjiin ja C-kielen osaamisen kehittäminen.

Sain ajatuksen omaan kääntäjään luultavasti joskus ammattikorkeakoulun ensimmäisen opiskeluvuoden aikana. Ajatus kypsyi muutaman vuoden mielessä, ja lopulta päätin, että akateemisen mielenkiinnon vuoksi olisi varmaan hyvä idea yrittää tehdä lopputyö aiheesta. Ajatuksena oli, että insinöörityö on itseä ja oppimista varten, enkä saanut mielenkiintoisempaa aihetta mieleeni, joten päätös oli valmis.

Lopulliseen päätökseen vaikuttivat myös alustavat suunnitelmat jatko-opinnoista. Lähtökohtana oli, että jos onnistun nyt tekemään jonkinlaisen kääntäjän, en tee siihen optimointivaihetta. Olin ideaa kypsytellessä lukenut kääntäjistä, että mikäli kääntäjän haluaa olevan hyvä, sen tulee sisältää optimointivaihe. Näin ollen ensimmäinen lähtökohta oli kirjoittaa kääntäjä ja tutkia käännösprosessia ja jatko-opinnoissa tutkia tämän työn pohjalta tarkemmin kääntäjäoptimointeihin, sillä ne vaikuttavat olevan aivan oma maailmansa ja taiteenlajinsa.

6.2 Suunnittelu

Perustin työni suunnittelun pääasiassa luettuun kirjamateriaaliin eli erityisesti teoksiin ”Compilers Principles, Tehcniques, & Tools” ja ”Modern Compiler Implementation In C”. Suunnitelmien mukaan sekä työssä tutkittu kääntäjien toimintamalli että tehty kääntäjä, ja alussa suunnitelmissa oli vielä täysimittainen kääntäjä, olisivat perustuneet rakenteeltaan näiden kahden kirjan rakenteeseen ja projekti niissä esiteltyyn kääntäjäarkkitehtuuriin. Suunnitteluvaiheessa en ottanut kantaa siihen, minkälaisia tietomalleja työssäni käyttäisin työn projektiosuuden eri funktioiden välillä ja millaisia algoritmeja tulisin työssäni käyttämään käännösprosessin aikana. Suunnittelu pysyi arkkitehtuuritasolla.

Kuten työn projektiosuus, perustui työn raporttiosa myös pääasiallisesti kahteen edellä mainittuun teokseen. Tosin teokset keskittyivät niin kääntäjien toimintaan ja käännösprosessiin, että katsoin suunnitteluvaiheessa aiheelliseksi myös tehdä lyhyen selvityksen kääntäjien historiasta.

Suunnitteluvaiheessa kääntäjää varten luotiin myös oma kieli joka, kuten on jo aiemmin mainittu, otti vaikutteita VIC-20-tietokoneen BASICista. Tavoitteena oli luoda riittävän helppo ja yksinkertainen kieli, jotta kääntäjä pysyisi mahdollisimman yksinkertaisena tehdä. Luotu kieli on Turing-yhteensopiva ja osaa yksinkertaisimmat aritmeettiset ope-

raatiot, mutta siitä on tarkoituksella jätetty kaikki ”hieno” ja kehittynyt toiminnallisuus pois, kuten tietorakenteet.

Suunnitteluvaiheessa projektiosuuden piti tuottaa AOT-tyyppinen kääntäjä, jonka back-end olisi tuottanut MASM-assemblerille sopivaa assemblya ja assembler olisi suorittanut viimeisen käännösvaiheen konekieleksi. Projektiosuus oli tarkoitus kirjoittaa C:llä, ja sen tuloksista oli myös tarkoitus verrata kääntäjän tuottaman ohjelman tehokkuutta muistinkäytön ja nopeuden puolesta muihin kääntäjiin. Koska suunnitteluvaiheen luotu kieli on Turing-yhteensopiva, oli myös kirjoitettavan kääntäjän tarkoitus täyttää tuo yhteensopivuus.

Suunnitteluvaiheessa kääntäjän oli tarkoitus olla Turing-yhteensopiva eli sen piti kyetä käsittelemään ohjelmasuorituksen haarautumiset ja mielivaltainen (joskin koneen muistimäärän rajoittama) määrä muuttujia. Kääntäjän kääntämä kieli on Turing-yhteensopiva, sillä se sisältää haarautumiskäskyt eikä muuttujien määrää ole rajoitettu.

Suunnitteluvaiheen seurauksena on toteutetussa kääntäjän front-endissä edelleen olemassa 160 merkin rajoite string-tyyppisille muuttujille. Poistamalla muistinkäsittelystä dynaamisuus oli tarkoitus helpottaa muuttujatyypin toteutusta, vaikka siitä on jälkikäteen tullut lähinnä ylimääräistä työtä rajoituksen seuraamisen vuoksi.

Jälkikäteen on todettavissa suunnitteluvaiheen olleen liian optimistinen ja pinnallinen. Pinnallinen siinä mielessä, etten kirjoittanut suunnitelmiani kielen ominaisuuksia lukuun ottamatta muistiin enkä vienyt suunnitelmia riittävän yksityiskohtaiselle tasolle. Liika optimismi lienee seurausta siitä, ettei etukäteen ollut täysin tiedossa, kuinka työlästä täysiverisen kääntäjän tekeminen todellisuudessa on. Tiedostin aiheen hankalaksi ja työlääksi, mutta kaikesta huolimatta teorian määrä ja sen toteuttamisen hankaluus yllättivät.

Positiivisena seikkana voidaan pitää suunnitteluvaiheessa luodun kielen yksinkertaisuutta. Niiltä osin mitä kielen ominaisuuksista toteutettiin, osoittautuivat suunnitelmat toteuttamiskelpoisiksi. Toisaalta suunnitteluvaiheessa mielessäni luoma rakenne työn raporttiosuudelle on mielestäni osoittautunut hyväksi ja toimivaksi kokonaisuudeksi.

6.3 Toteutus

Työn toteutusvaihe oli hyvin pitkä ja vaiherikas. Toteutusvaiheen aikana tuli varsin pian ilmi valitun aiheen hankaluus ja suunnitteluvaiheen liiallinen optimismi. Erityiseksi ongelmaksi työvaiheen aikana osoittautui suunnitteluvaiheessa täysin ylenkatsottu työelämän vaatima aika ja sen vaikutukset niin projektiosuuden kuin raportointiosuuden käytössä olevaan aikaan.

Työ alkoi raportointiosan kirjoittamisella. Raportointiosan tekeminen ensin oli mielestäni perusteltua, sillä se pakotti minut kirjoittamalla käymään selkeästi läpi kääntäjien toimintaa ja käännösprosessia.

Raporttiosan kirjoittamisen suurimmaksi haasteeksi paljastuivat käännösten löytämisen vaikeus ja asiassa pysyminen. Ensimmäiseen suuri apu oli suomalaisten yliopistojen tietojenkäsittelytieteiden kääntäjiä käsittelevä luentomateriaali, josta termistön saattoi tarkistaa. Toinen ongelma oli seurausta siitä, että kääntäjät tarvitsevat ohjelmointikielen olemassaolonsa perusteluksi, ja ohjelmointikielet tarvitsevat kääntäjiä ollakseen käyttökelpoisia, oli erottelu näiden kahden välillä hankalaa. Luonnollista haastavuutta raporttiosan kirjoitukseen toi itse aihe ja pelkästään esitetyn aiheen ymmärtäminen ja kääntäminen ymmärrettäväksi suomeksi.

Raporttiosuutta suuremman vaivan aiheutti työn projektiosuus. Työn lähdemateriaalin abstrakti kääntäjien ja käännösprosessin lähestyminen ei antanut suoria vastauksia siihen, mitä ja millaista tietoa kunkin käännösvaiheen tulisi käsitellä. Niinpä jouduin lopulta yrityksen ja erehdyksen kautta etenemään projektin kanssa.

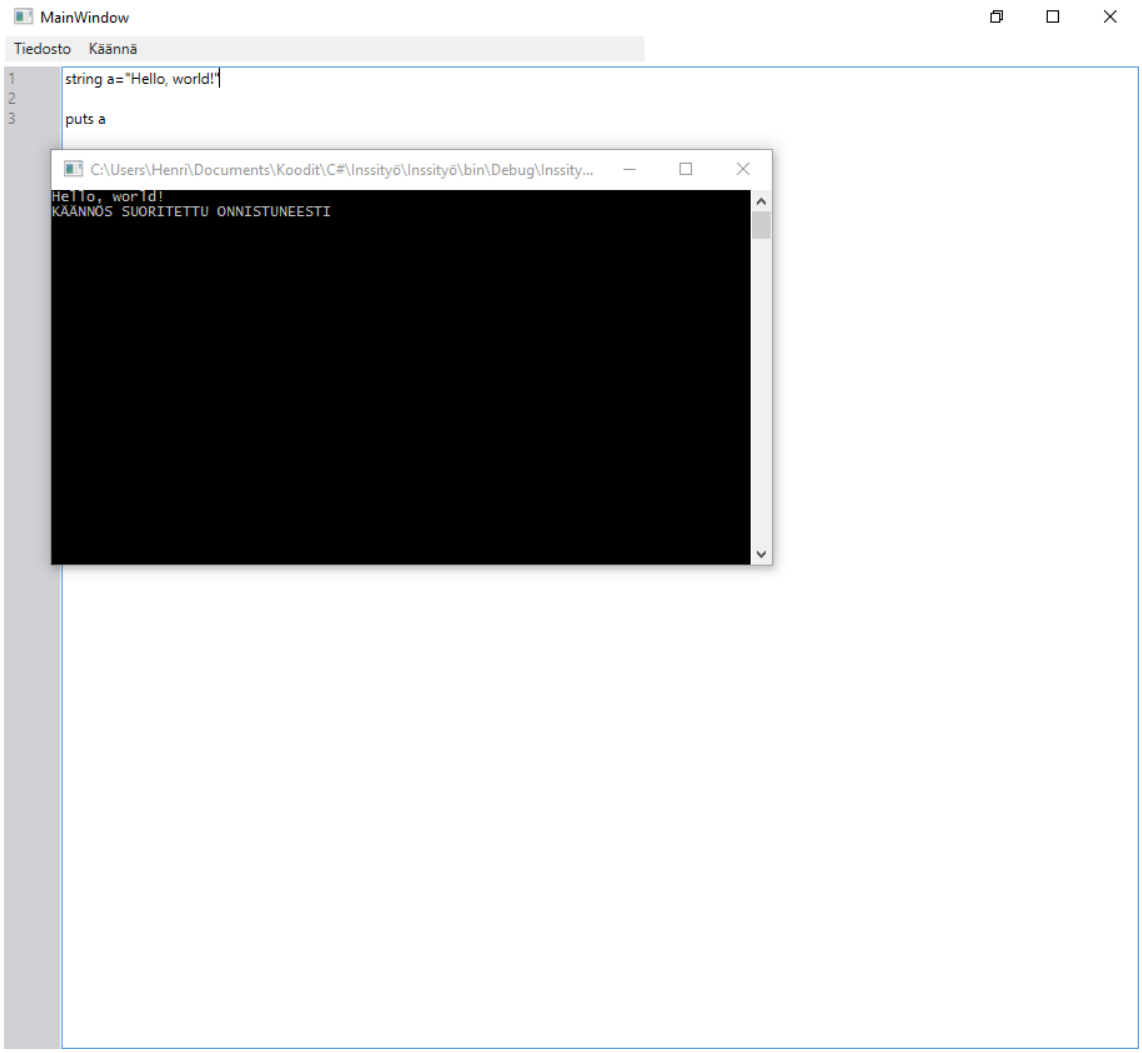
Suurta vaivaa projektiosuudessa aiheutti valittu kieli ja kehitysympäristö. C ei perustyyppien lisäksi tarjoa paljoa valmiita rakenteita, ja kielen dynaamisen muistin käsittely on täysin manuaalista. Tämä yhdistettynä siihen, etteivät valitun työkalun virheenetsintä ja -korjausominaisuudet olleet minulle tuttuja, johti siihen, ettei työ edennyt kovin nopeasti. Huomattava osa ajasta meni muistin oikeellisuuden tarkistamiseen.

C:llä minun onnistui lopulta luoda liki toimiva lekseeminen analyysi, ja siirryin jo kirjoittamaan semanttista analyysia, kun huomasin muistinkäsittelyssä huomattavan virheen: ohjelman muistinvapautustoteutukset olivat viallisia. Vapautusproseduurit onnistuivat muistin vapauttamisessa, eli muistivuotoja ei tapahtunut, mutta vapautetun arvon osoit-

taminen NULL-arvoon epäonnistui. Tämä oli seurausta siitä, että vapauttavat funktiot olivat void-tyyppisiä ja vastaanottivat parametrinaan osoittimen. Osoittimen asettaminen NULL-arvoon kyseisessä funktiossa ei johtanut siihen, että se olisi osoittanut NULL-arvoon suorituksen jatkuessa muissa funktioissa, mikä aiheutti NULL-vertailujen epäonnistumisen ja lopulta vääjäämättä muistialueen ylitykseen ja ohjelmasuorituksen epäonnistumiseen.

Ongelma olisi ollut kierrettävissä vaihtamalla osoitin-parametrit osoittimen osoittimeen (49, s. 70.) tai vaihtamalla void-tyyppisten vapautusfunktioiden tyyppiä, mutta tämä olisi vaatinut suurehkon työn. Työn projektiosuus oli edennyt tarpeettoman hitaasti, sillä suuri osa ajasta meni työn kannalta toissijaisiin asioihin, kuten työn muistin oikeellisuuden varmistamiseen ja tietorakenteiden toteuttamiseen. Projektiosuuden valmistumisen kannalta näitä kahta tärkeämpää oli saada jotain projektiosuuden tarkoitukseen liittyvää tehtyä, joten vaihdoin kielen C:stä C#:iin. C#:n etuihin kuuluvat muun muassa oliopohjaisuus ja automaattinen muistinkäsittely, mutta heikkoutena sillä kirjoitettu koodi ei ole yhtä tehokasta kuin osaavasti kirjoitettu C. Toiseksi tämä tarkoitti sitä, että jouduin luopumaan C-kielen osaamisen kehittämisestä.

Ohjelmointikielen vaihtaminen voitiin työn itsensä puolesta katsoa erinomaiseksi ratkaisuksi. Sillä sain huomattavasti lisävauhtia projektiosuuden kehittämiseen, sillä kielen ominaisuuksien puolesta en joutunut enää keskittymään työn kannalta toissijaisiin asioihin. C:llä tuotetun koodipohjan kääntäminen C#:lle ei ollut kovin iso työ.



Kuva 4. Ensimmäinen onnistunut käännös.

Toteutusvaiheessa työn tavoitetta myös tarkistettiin. Osoittautui, että täysiverinen kääntäjä olisi ollut turhan suuri ja työläs tavoite, joten työssä tyydyttiin lopulta toteuttamaan suunnitteluvaiheessa tehdylle kielelle kääntäjän front-end. Samalla työn tavoitteeksi tarkentui raportin osalta käännösprosessin tutkiminen.

Päätös oli kaksijakoinen. Uudelleenmäärittelystä seurasi, ettei työn tuloksena olisi kokonaista kääntäjää, mikä tarkoitti, etteivät suunnitteluvaiheen vertailut toisiin kääntäjiin enää onnistuneet. Toisaalta työn itsensä kannalta tarkempi rajaus voidaan katsoa eduksi, sillä se rajasi työn käsittelemään alkuperäisten suunnitelmien ydinideaa eli vastaamaan kysymyksiin, mitä ovat kääntäjät, millaisia kääntäjiä on, miten ne eroavat toisistaan ja mitä käännösprosessissa tapahtuu. Jo näissä neljässä kysymyksessä on paljon vastattavaa.

Toteutusvaiheen tuloksena saatiin lopulta jokseenkin toimiva kääntäjän front-end. Se ei kyennyt täysin täyttämään kaikkia suunnitteluvaiheen kielen vaatimuksia, mutta toteutetun osan katsottiin olevan riittävä osoitus siitä, että työn raportointiosan viesti on sisäistetty siten, että sitä on kyetty soveltamaan myös käytäntöön.

6.4 Insinöörityön toteutuksen jatkokehitys ja tulevaisuus

Vaikka alkuperäinen kääntäjä jäi toteuttamatta, on tehty kääntäjän front-end jatkokehityskelpoinen, ja kuten luvussa 5.1 kirjoitin, on minulla suunnitelmia tehdylle työlle tulevaisuudessa mahdollisten jatko-opintojen myötä.

Ensimmäinen ja luultavasti arvattavin jatkokehityksen kohde on itse kääntäjän front-end. Sitä ei saatu suunnitelmien tasolle toiminnallisuuden puolesta, ja haluaisin saada tehdyt suunnitelmat valmiiksi.

Toinen jatkokehityksen kohde on irrottaa kääntäjän front-end C#:sta luomalla sille kyseisestä kielestä riippumaton back-end koodigenerointivaiheineen ja optimointivaiheineen. Jatko-opiskelusuunnitelmien näkökulmasta tämä vaihe on välttämätön, mikäli haluan niissä käyttää tämän työn tuloksia. Tämän myötä tehty kääntäjän front-end olisi lopulta laajentunut aidoksi kääntäjäksi ja siten vastaisi aivan alkuperäisiä tavoitteita.

Kolmas jatkokehityksen aihe on itse kääntäjän front-endin koodipohja. Koodipohja on melko sotkuinen ja paikoittain hankalasti luettava, mikä hankaloittaa jatkokehitystä. Osittain tätä ongelmaa voisi helpottaa ottamalla kehitykseen mukaan jonkinlaisen versionhallinnan, joka tukisi olemassa olevien komponenttien uudelleenkirjoitusta. Uudelleenkirjoituksen yhteydessä tapahtuvat virheet eivät olisi niin hankalasti korjattavissa, sillä versionhallinnasta saisi aina viimeisimmän toimineen version tilalle tai sieltä voisi tarkistaa, mikä mahdollinen ajatusvirhe uudelleenkirjoitettuun koodin on päätynyt.

Edellä mainittujen lisäksi haluaisin laajentaa työtä varten kehitettyä kieltä käyttökelpoisemmaksi ohjelmointikieleksi. Tämä tarkoittaisi esimerkiksi tukea useammalle lähdekooditiedostolle, uusia tietorakenteita ja -tyyppejä, string-tyypin 160 merkin rajoituksen poistamista, tukea kirjastoille ja uusien näkyvyysalueiden lisäämistä globaalin näkyvyysalueen lisäksi.

6.5 Johtopäätökset

Insinöörityön lopputuloksena on sinällään erittäin käyttökelpoinen ja jatkokehitykseen sopiva kääntäjän front-end. Vaikka kääntäjän koodipohja jättää toivomisen varaa, kuten luvussa 5.4 on todettu, ovat sen ongelmat ainakin tiedossa ja niihin on jo kehityksen aikana puututtu uusia ominaisuuksia toteutettaessa, vaikka esiintyneet ongelmat on jätetty vanhoihin toteutuksiin (uudelleenkirjoitusta ei siis ole harrastettu).

Vaikka työn front-endiin jäi puutteita myös sen puolesta, mitä toteutettiin ja mitä suunniteltiin, on toimiva osa kuitenkin osoitus siitä, että front-endin ajatus on ihan oikean suuntainen ja lähinnä toteutusta vailla. Puutteistaan huolimatta toteutettu kääntäjän front-end on kykeneväinen tuottamaan kolmiosoitekoodia, joka voidaan onnistuneesti kääntää C#:ksi ja siitä edelleen.

Itse kääntäjän front-endin teoriasta suhteessa toteutettuun työhön voidaan päätellä seuraavat kolme pääasiaa:

1) Prosessi vaatii paljon käsityötä ja sen toteuttamisessa on haastavaa noudattaa hyviä ohjelmointitapoja. Leksikaalinen analyysi on raskasta merkkijonovertailua, joka ilman säännöllisiä lausekkeita johtaa suureen koodipohjaan ja pitkiin vertailurakenteisiin, kuten tässä työssä. Toisaalta säännölliset lausekkeet ovat hankalasti luettavia verrattuna tavallisiin vertailurakenteisiin.

2) Teoreettisen materiaalin muuntaminen toimivaksi koodiksi on haasteellista. Lähdemateriaali oli todella teoriapainotteinen, ja sen ymmärtäminen ja toimivaksi ohjelmaksi saattaminen oli tällä koulutustasolla ongelmallista. Semanttisen analyysin naiivi toteutus, eli lähdemateriaalissa sille osoitettujen kolmen tehtävän toteuttaminen ilman ohjelmistotuotannollista näkökulmaa, johti vaikeasti ylläpidettävään koodiin. Lähdemateriaalissa ja tämän työn raporttiosuudessa on myös useammassa kuin yhdessä vaiheessa esitelty erilaisia puurakenteita (katso luvut 3.1 ja 3.2), mutta näitä ei ole käytetty laisinkaan toteutetussa kääntäjän front-endissä. Työn toteutuksessa sen sijaan tyydyttiin käyttämään lista- ja hajautustaulu-rakenteita.

3) Lähdemateriaalissa esitetty front-endin kääntäjäarkkitehtuuri ei ole ainoa oikea tapa tehdä kääntäjän eri osia. Toteutetusta kääntäjän front-endistä puuttuu

esimerkiksi täysin luvussa 3.1 esitelty jäsennysvaihe. Lisäksi lähdemateriaalissa esitettyjä tietorakenteita ei ole juuri käytetty, kuten kohdassa 2 on mainittu.

Mitä implikaatioita valituilla tietorakenteilla esitettyihin tietorakenteisiin ja toteuttamatta jääneillä vaiheilla on, siihen ei tämä työ valitettavasti kykene kaikilta osin vastaamaan. Voi olla, että valittujen rakenteiden vuoksi toteutettu kääntäjän front-end on nopeuden tai muistinkäytön puolesta epäoptimaalinen. Tämä vaatisi jatkotutkimuksia ja vertailuja muiden kääntäjien front-endeihin. Kääntäjän front-endin toteutuksessa käytetystä kielestä voidaan tosin päätellä, että toteutus ei ole optimaalinen, mikäli tavoitteena on äärimmäinen nopeus. Vaikka C# mahdollistaa pointteriaritmetiikan joillekin tietotyypeille, ei tätä ole käytetty hyväksi työtä tehdessä. Tämä yhdistettynä register-varatun sanan puuttumiseen, automaattiseen muistinkäsittelyyn ja kielen sekä muihin turvallisuusominaisuuksiin on todennäköisesti johtanut siihen, ettei leksikaalisen analyysin viittaustapa taulukoihin ole nopeuden näkökulmasta optimaalinen eikä suorituskyky työssä muutenkaan ole sitä, mihin asiansa osaava C- tai assembly-kehittäjä kykenisi. Toisaalta menetetyn suorituskyvyn voidaan katsoa olevan marginaalinen, sillä toteutuksen puutteiden vuoksi työtä varten kehitetty kieli ei ole vielä kypsä ohjelmointikieli. Microsoft on myös osoittanut C#:n olevan kääntäjän tekemiseen pätevä kieli, sillä sen ”Roslyn”-niminen C#-kääntäjä on kirjoitettu C#:lla (kääntäjän Visual Basicia kääntävä osuus on tehty Visual Basicilla). (50.)

Työssä tutkitusta kääntäjämallista voidaan tehdä se johtopäätös, että sen edustamassa mallissa kääntäjän front-end on täysin alustariippumaton, mutta vahvasti riippuvainen prosessoitavasta kielestä. Jos olisi olemassa leivänpaahdin, jolle olisi mahdollista pystyttää C#:n suoritusympäristö, palauttaisi tekemäni kääntäjän front-end työtä varten luodulle kielelle täysin samanlaisen tuloksen kuin Windows 10 -alustalla olettaen, ettei kielellä luotu lähdekoodi muutu alustojen välillä. Sen sijaan se on täysin käyttökelvoton esimerkiksi C:n prosessointiin alustalla, jolle työ on tehty. Sen sijaan kääntäjän back-end on tutkitussa mallissa täysin kielestä riippumaton, mutta vahvasti alustariippuvainen. Tämä kaikki on seurausta luvusta 3.2, ja mikäli se jätettäisiin tekemättä, ei tilanne olisi edellä kuvatun kaltainen.

Ollakseen täysin Turing-yhteensopiva, vaatii tehty kääntäjän front-end jatkokehitystä. Tällä hetkellä se osaa yksinkertaisimmat aritmeettiset operaatiot eli summauksen, vähennyksen, kerto- ja jakolaskut, mutta ei kykene suoriutumaan kontrollirakenteista, eli ehto-, toisto- ja siirtymärakenteet eivät vielä joko tuota kolmiosoitekoodia tai niitä ei

välikoodiesityksen suorituksessa huomioida. Ollakseen täysiverinen kääntäjä, vaatisi tehty kääntäjän front-end mahdollisesti jonkinlaisen itsetehdyn back-endin. Toisaalta kuten luvussa 3.2 on mainittu, käytti ensimmäinen C++-kääntäjä välikoodigeneroinnissa välikielenä C:tä, sillä sille löytyi valmiita kääntäjiä.

Työn projektiosuudesta puuttuvan kääntäjän back-endin takia saattaa työn raportointiosuuden luku 4 vaikuttaa jonkin verran irralliselta kokonaisuudelta, mutta työn eräs tavoite oli tutkia käännösprosessia. Tästä näkökulmasta luvun mukana olo työssä on mielestäni perusteltua, sillä itsenäisessä kääntäjäkokonaisuudessa luvun sisällöllä on paikkansa, erityisesti AOT-tyyppistä optimoivaa kääntäjää tutkittaessa.

Tehdyn kääntäjän front-endin jatkokehityksen lisäksi työ herätti jatkokysymyksiä, joihin haluan perehtyä myöhemmin:

1) Kuinka paljon ohjelmointikielen tyyli vaikuttaa kääntäjään? Työn projektiosa keskittyi imperatiivisen ohjelmointikielen kääntäjän front-endin tekemiseen, mutta millaisia muutoksia kääntäjän front-endiin pitää tehdä, jos käännettävä kieli onkin esimerkiksi funktionaalinen?

2) Miten kääntäjät huolehtivat ohjelmointikielten näkyvyysalueista? Nyt käytetyssä ohjelmointikielessä ei ollut minkäänlaisia rajoituksia näkyvyydelle. Miten esimerkiksi Javasta, C#:sta ja C++:sta tutut private- ja public-näkyvyysalueet toimivat kääntäjän näkökulmasta?

Kaikkiaan kääntäjät tämän työn käsittelemässä syvyydessä ovat huomattavan teoreettisia ohjelmia, joiden ympärillä on tehty valtava määrä työtä ja joita varten on tehty paljon tutkimusta erityisesti optimointivaiheen osalta. Kääntäjien toimintaa kuvaavan matematiikan ja algoritmien toiminta mennee ammattikorkeakoulu-tason opiskelijan osaamisen tuolle puolen, mutta toisaalta tehty kääntäjän front-end on todiste siitä, että riittävällä uteliaisuudella ja kokeilunhalulla jonkinlainen käännösprosessi valitulle tai luodulle ohjelmointikielelle on tehtävissä, jos vain yrittää. Lisäksi se on todiste siitä, että kaikesta monimutkaisesta teoriasta ja matematiikasta huolimatta teoria on jossain määrin myös tällä tasolla mahdollista sisäistää ja raportoida. Raporttiosuuden suurin vajavaisuus lienee kääntäjien matemaattisen esityksen puute, mutta mielestäni se selvittää ymmärrettävällä tavalla kääntäjien toimintaa ja käännösprosessia. Toiseksi raportti on ehkä akateemisesti vajavainen, sillä se tutkii käännösprosessia vain yhden kääntäjä-

arkkitehtuurin näkökulmasta, vaikka tekstin seassa muistutetaan, ettei selvitetty malli ole ainoa tapa asioiden tekoon.

Ohjelmistotuotannollisesta ja tavoitteellisesta näkökulmasta työn projektiosuus on melko puutteellinen, eikä onnistunut saavuttamaan kaikkia sille asetettuja tavoitteita, mutta ydintavoitteessaan se on onnistunut. Kun editoriin kirjoittaa luotujen kielioppisääntöjen mukaisesti ”Hello, World!” ja painaa käännä ja aja-nappia, tulostaa ohjelma konsoliin kyseisen tekstin. Siitä olen tyytyväinen.

Lähteet

1 Aho, Alfred V., Lam, Monica S., Sethi, Ravi & Ullman, Jeffrey D. 2007. Compilers Principles, Techniques, & Tools. Addison-Wesley.

2 Salomon, David. 1993. Assemblers And Loaders. Verkkodokumentti. <http://www.davidsalomon.name/assem.advertis/asl.pdf>. Luettu 3.10.2015.

3 Fairhead, Harry. Assemblers and assembly language. Verkkodokumentti. <http://www.i-programmer.info/babbages-bag/301-assemblers-compilers-and-interpreters.html?start=1>. Luettu 3.10.2015.

4 Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D. Verkkodokumentti. Intel. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>. Luettu 3.10.2015.

5 Smotherman, Mark. EDSAC. Verkkodokumentti. <https://people.cs.clemson.edu/~mark/edsac.html>. Luettu 3.10.2015.

6 Padua, David. 2000. The Fortran I Compiler. Verkkodokumentti. <http://polaris.cs.uiuc.edu/publications/c1070.pdf>. Luettu 4.10.2015.

7 Graham, Paul. 2001. What made LISP different. Verkkodokumentti. <http://www.paulgraham.com/diff.html>. Luettu 10.10.2015.

8 Proposing Cobol. Verkkodokumentti. Smithsonian. <http://americanhistory.si.edu/cobol/proposing-cobol>. Luettu 10.10.2015.

9 de Beer, Huub. 2006. The History of the ALGOL Effort Summary and Conclusions. Verkkodokumentti. <https://heerdebeer.org/ALGOL/conclusion.html>. Luettu 10.10.2015.

10 de Beer, Huub. 2006. The History of the ALGOL Effort. Verkkodokumentti. <http://heerdebeer.org/ALGOL/The_History_of_ALGOL.pdf>. Luettu 10.10.2015.

11 Dvorak, John C.. 2009. How the Itanium Killed the Computer Industry. Verkkodokumentti. <http://www.pcmag.com/article2/0,2817,2339629,00.asp>. Luettu 11.10.2015.

12 Turley, Jim. 2013. Sinking the Itanic. Verkkodokumentti. <http://www.eejournal.com/archives/articles/20130306-itanium/>. Luettu 11.10.2015.

13 Knuth, D.& Binstock, A. 2008. Interview with Donald Knuth. Verkkodokumentti. <http://www.informit.com/articles/article.aspx?p=1193856>. Luettu 11.10.2015.

14 Aycock, John. 2003. A Brief History of Just-In-Time. Verkkodokumentti. <http://eecs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/JustInTimeCompilation.pdf>. Luettu 11.10.2015.

15 Hansen, Gilbert Jopseh. 1974. Adaptive systems for the dynamic run-time optimization of programs. Verkkodokumentti. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.376.3638&rep=rep1&type=pdf>. Luettu 11.10.2015.

16 JIT compiler overview. Verkkodokumentti. IBM. <https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.win.70.doc/diag/understanding/jit_overview.html>. Luettu 25.10.2015.

17 Todorov, Tsvetomir Y. 2013. Understanding .NET Just-In-Time Compilation. Verkkodokumentti. <http://www.telerik.com/blogs/understanding-net-just-in-time-compilation>. Luettu 1.11.2015.

18 Andreasson, Eva. 2012. JVM performance optimization, Part 2: Compilers. Verkkodokumentti. <http://www.javaworld.com/article/2078635/enterprise-middleware/jvm-performance-optimization-part-2-compilers.html>. Luettu 26.10.2015.

19 Bouakaz, Adnan., Puaut, Isabelle & Rohou, Erven. Predictable Binary Code Cache: A First Step Towards Reconciling Predictability and Just-In-Time Compilation. Verkkodokumentti. <http://www.irisa.fr/alf/downloads/puaut/papers/RTAS2011.pdf>. Luettu 1.11.2015.

20 Fairhead, Harry. 2011. Javasript Is Slow. Verkkodokumentti. <http://www.i-programmer.info/news/86-browsers/3492-javascript-is-slow.html>. Luettu 1.11.2015.

21 Types of Programming Languages: Compiled, Interpreted or Both? 2011. Verkkodokumentti. Rookiecode <http://www.rookiecode.com/2011/02/compiled-interpreted-or-both/>. Luettu 1.11.2015.

22 Cross assembler-artikkeli. Verkkodokumentti. PCMag. <http://www.pcmag.com/encyclopedia/term/40490/cross-assembler>. Luettu 26.3.2017.

23 Hyde, Randy. 2010. HLA Reference Manual. Verkkodokumentti. <http://www.plantation-productions.com/Webster/HighLevelAsm/HLADoc/HLARef/HLARef_pdf/HLAReference.pdf>. Luettu 26.3.2017.

24 Advantages of Inline Assembly. Verkkodokumentti. Microsoft. <https://msdn.microsoft.com/en-us/library/80ccffx3.aspx>. Luettu 26.3.2017.

25 James, Mike. Interpreters, VMs And JIT. Verkkodokumentti. <http://www.i-programmer.info/babbages-bag/352-interpreters.html>. Luettu 22.11.2015.

26 McCarthy, John. 1996. The implementation of LISP. Verkkodokumentti. <http://www-formal.stanford.edu/jmc/history/lisp/node3.html>. Luettu 22.11.2015.

27 TIOBE Index. 2015. Verkkodokumentti. Tiobe. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Luettu 22.11.2015.

28 Carbonnelle, Pierre. PYPL. Verkkodokumentti. <http://pypl.github.io/PYPL.html>. Luettu 22.11.2015.

29 TIOBE Index is being gamed. Verkkodokumentti. Wordpress. <https://blog.timbunce.org/2009/05/17/tiobe-index-is-being-gamed/>. Luettu 22.11.2015.

30 Appel, Andrew W. 2004. Modern Compiler Implementation in C. Cambridge university press.

31 The C Preprocessor. Verkkodokumentti. Free Software Foundation, Inc. <https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html#SEC7>. Luettu 22.11.2015

32 Johnson, Maggie & Zelenski, Julie. 2008. Lecture notes. Verkkodokumentti <http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf.>. Luettu 22.11.2015

33 The Mono Runtime. Verkkodokumentti. Xamarin. <http://www.mono-project.com/docs/advanced/runtime/>. Luettu 28.3.2017.

34 GCC Front Ends. Verkkodokumentti. Free Software Foundation, Inc. <https://gcc.gnu.org/frontends.html>. Luettu 28.3.2017.

35 Status of Supported Architectures from Maintainers' Point of View. Verkkodokumentti. Free Software Foundation, Inc. <https://gcc.gnu.org/backends.html>. Luettu 28.3.2017.

36 Rentzsch, Jonathan. 2005. Data alignment: Straighten up and fly right. Verkkodokumentti. <https://www.ibm.com/developerworks/library/pa-dalign/pa-dalign-pdf.pdf>. Luettu 10.1.2016.

37 Gatlin, Kang Su. 2006. Windows Data Alignment on IPF, x86, and x64. Verkko-dokumentti. <https://msdn.microsoft.com/en-us/library/aa290049%28v=vs.71%29.aspx>. Luettu 10.1.2016.

38 Turley, Jim. 2014. Intel vs. ARM: Two titans' tangled fate. Verkkodokumentti. <http://www.infoworld.com/article/2610369/processors/intel-vs--arm--two-titans--tangled-fate.html>. Luettu 29.3.2017.

39 Stokes, Jon. 1999. RISC vs. CISC: the Post-RISC Era: A historical approach to the debate. Verkkodokumentti. <https://arstechnica.com/features/1999/10/rvc/>. Luettu 29.3.2017.

40 AMD64 Architecture Programmer's Manual Volume 2: System Programming. Verkkodokumentti. AMD. <http://support.amd.com/TechDocs/24593.pdf>. Luettu 29.3.2017.

41 64-bit versions of Windows do not support 16-bit components, 16-bit processes, or 16-bit applications. Verkkodokumentti. Microsoft. <https://support.microsoft.com/fi-fi/help/896458/64-bit-versions-of-windows-do-not-support-16-bit-components-16-bit-processes-or-16-bit-applications>. Luettu 29.3.2017.

42 OS Platform Statistics. Verkkodokumentti. W3Schools. <https://www.w3schools.com/browsers/browsers_os.asp>. Luettu 5.3.2017.

43 Desktop Operating System Market Share. Verkkodokumentti. NetMarketShare. <https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpsp=216&qpnp=1&qptimeframe=M>. Luettu 5.3.2017.

44 Casserly, Martyn. 2017. Windows 10 Home vs Pro. Verkkodokumentti. <http://www.pcadvisor.co.uk/feature/windows/windows-10-home-vs-windows-10-pro-uk-difference-3618710/>. Luettu 5.3.2017.

45 Intel Core i7-5960X & 5820K (Haswell-E). 2014. Verkkodokumentti. Muropaketti. <https://muropaketti.com/artikkelit/prosessorit/intel-core-i7-5960x-5820k-haswell-e1>. Luettu 5.3.2017.

46 Buxton, Mark. 2011. Haswell New Instrction Descriptions Now Available! Verkko-dokumentti. <https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/>. Luettu 5.3.2017.

47 object (C# Reference). 2015. Verkkodokumentti. Microsoft. <https://msdn.microsoft.com/fi-fi/library/9kkx3h3c.aspx>. Luettu 29.3.2017.

48 String Class. Verkkodokumentti. Microsoft. <https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx#Remarks>. Luettu 29.3.2017.

49 Reese, Richard. 2013. Understanding and Using C Pointers. O'Reilly.

50 Kunk, Joe. 2012. 10 Questions, 10 Answers on Roslyn. Verkkodokumentti. <https://visualstudiomagazine.com/articles/2012/03/20/10-questions-10-answers-on-roslyn.aspx>. Luettu 30.3.2017.

Insinöörityötä varten luodun kielen kielioppisäännöt

Tämä liite sisältää kielen kielioppisäännöt ja syntaksin

1) Muuttujatyypit, -esittely ja operaattorit

Kaikki muuttujat on ilmoitettava ohjelman alussa, ja ne ovat muistissa koko ohjelman suorituksen ajan. Kielen näkyvyysalue (eng. scope) on globaali, joten kerran ohjelman alussa esitelty muuttuja on sama kaikkialla. Muuttujanimissä vain kirjaimet ovat sallittuja. Muuttujia ei voi uudelleen määritellä.

Muuttujat alustetaan seuraavalla tapaa:

```
num x=5
dec y=1.6
char z="c"
string str="Moi" //Alustetaan 3 merkin mittainen merkkijono
string str //160 merkin pituinen merkkijono
```

Lukutyypit:

- Kokonaisluvut ovat tyyppiä num. Muuttujan esittely tapahtuu num x, y, z…ö, jossa kirjain on muuttujan nimi. Luvut alustetaan automaattisesti arvoon 0. Luvuilla ei ole ylintä taikka alinta sallittua arvoa vaan luvun koko rajoittuu ainoastaan käytettävissä olevan muistin määrään.

- Liukuluvut ovat tyyppiä dec. Muuttujan esittely tapahtuu kuten kokonaisluvuillakin, ja myös liukuluvut alustetaan arvolla 0. Liukuluvuissa käytetään 64 bitin tarkkuutta IEEE 754–1985-standardin mukaisesti, mikä antaa lukualueeksi ± 2,23*10^$^{-308}$-1,8*10^{308}.

Merkit ja merkkijonot:

- Yksittäinen merkki on tyyppiä char, ja sen esittely sekä alustus tapahtuvat kuten lukujen. char käyttää utf-16-koodaustapaa.

- Merkkijono on tyyppiä string x, jossa x on muuttujan nimi. Merkkijonon koko on käännöksen jälkeen vakio, eli siihen ei voi ajon aikana vaikuttaa.

- Merkkijonoja on mahdollista yhdistää toisiinsa kuten lukuja käyttämällä + merkkiä. Tässä kannattaa olla tarkkana 160 merkin rajoitteen kanssa.

Taulukot:

- Liuku- ja kokonaisluvut voidaan taulukoida kuten merkit ja niitä koskettavat merkkijonojen säännöt. Toisin sanoen, kaikki taulukot ovat käännöksen jälkeen vakiokokoisia, ja ne ovat muistissa koko ohjelman suorituksen ajan. Luvut esitellään pilkulla toisistaan eroteltuina.

- Taulukot ovat aina yksiulotteisia jonoja.

- Taulukoiden indeksointi alkaa nollasta kuten liki kaikissa muissakin ohjelmointikielissä. Indeksiin osoitetaan seuraavasti

```
char *str="Hei"
char str0()="Hei"
num numeroita()=1,2,3,4,5,6

puts str (1) //e
puts str0(2) //i
puts numeroita(5) //6
```

Operaattorit:

- Vertailu tapahtuu merkeillä <, <=, >, =>, != ja ==. Auki kirjoitettuna vertailut ovat järjestyksessä: pienempi, pienempi tai yhtä suuri, suurempi, suurempi tai yhtä suuri, erisuuri ja yhtä suuri. Vertailumerkit ovat siis samoja kuin C:ssä, Javassa ym. ”arkielämän” ohjelmointikielissä.

- Peruslaskutoimitukset tapahtuvat niitä vastaavilla merkeillä +, -, * ja / eli järjestyksessä yhteen-, vähennys-, kerto- ja jakolasku. Eksponentit onnistuvat merkillä ^.

- Loogiset vertailut tapahtuvat avainsanoilla and, or ja not.

- Arvon sijoitus muuttujaan tapahtuu merkillä =.

2. Ohjausrakenteet ja komennot

Toistorakenteet:

- Varattu sana loop, jolle annetaan parametrina toiston alkukohta, yläraja ja päivitysväli. Kaikki parametrit ovat vapaaehtoisia.

```
num i
num upperLimit=10

loop i to upperLimit by 1
   puts i //0, 1, 2...10.

loop
   puts i //10, 10, 10, 10, 10...10. Ikisilmukka.

upperLimit=11

loop i to upperLimit
   puts i //10, 10, 10, 10...10. Ikisilmukka.

loop i by 1
   puts i //10, 11, 12, 13, 14, 15...n, jossa n suurin mahdollinen luku ennen muistin loppumista
```

Siirtymät funktioiden välillä ja ehtolauseet:

Kieli ei sisällä funktioita samaan tapaan kuin C, Java ja muut vastaavat ohjelmointikielet. BASICin tapaan hyppykäskyt koodin osasta toiseen tapahtuvat varatulla sanalla GOTO, mutta siinä missä BASICissa käytettiin rivinumeroita, on tässä työssä käytössä varattu sana tag x, jossa x on tagin nimi. Alifunktiosta palataan takaisin pääohjelmaan varatulla sanalla return. Alifunktiot eivät ota vastaan parametreja eivätkä palauta mitään arvoja.

Pääohjelmaa ei tarvitse erikseen esitellä, vaan sen suoritus alkaa välittömästi muuttujaesittelyiden jälkeen. Pääohjelman suoritus loppuu return-komentoon.

Ehtolauseet noudattavat tavanomaista if-else if-else-rakennetta.

Alla esimerkki ehtolauseista funktiokutsuista ja paluusta pääohjelmaan.

```
num i=1
num upperLimit=100

loop i to upperLimit by 1
   if i%3==0 and i%5==0
      goto FizzBuzz
   else if i%5==0
      goto Buzz
   else if i%3==0
      goto Fizz
   else
      goto PrintI

tag FizzBuzz
   puts "FizzBuzz"
   return  //Ilman tätä avainsanaa ohjelman suoritus pysähtyy

tag Fizz
   puts"Fizz"
   return

tag Buzz
   puts "Buzz"
   return

tag PrintI
   puts i //Globaali näkyvyys
   return
```

3) I/O-operaatiot

Kieli tukee I/O-operaatioista vain syötteen lukua näppäimistöltä ja tulostamista näytölle. Tiedostojenkäsittely ja muu vastaava puuttuu.

Tulostamiseen on kaksi vaihtoehtoista komentoa: print ja puts. puts lisää tulosteeseen automaattisesti rivivaihdon, mutta muutoin komennoilla ei ole mitään eroa.

Näppäimistöltä luku onnistuu komennolla get. Alla esimerkki.

```
num i
i=get "Syötä kokonaisluku: "
```