Tampereen yliopisto

Tuomas Knaapi

KÄÄNTÄJÄN TOIMINTA

Lähdekoodin muuntaminen konekielelle

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Tammikuu 2020

TIIVISTELMÄ

Tuomas Knaapi: Kääntäjän toiminta: lähdekoodin muuntaminen konekielelle
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Tammikuu 2020

Kääntäjä on ohjelma, joka mahdollistaa korkean tason ohjelmointikielien käytön. Korkean tason ohjelmointikielet ovat niin abstrakteja, että ne sopivat vain ihmisten käytettäväksi, eikä niillä toteutettua ohjelmaa voida suoraan ajaa tietokoneella. Tästä syystä tarvitaan kääntäjä, joka rakentaa käännetyn version ohjelmakoodista. Käännettävää koodia kutsutaan lähdekoodiksi, kun taas käännettyä koodia kohdekoodiksi.

Tässä tutkielmassa tarkastellaan, miten ohjelmointikielten kääntäjä toimii. Työssä tutkitaan käännösprosessin vaiheita, joihin kuuluvat analyysivaihe ja synteesivaihe. Analyysivaiheen päätehtävänä on tuottaa väliesitys lähdekoodista, joka välitetään synteesivaiheelle. Analyysivaihe jakautuu neljään pienempään osaan, joilla kaikilla on oma tehtävänsä välikoodin tuottamisessa. Osien nimet ovat leksikaalinen analyysi, syntaksianalyysi, semanttinen analyysi ja välikoodin generointi. Leksikaalinen analyysi tuottaa lähdekoodin sanoista tunnisteita, jotka välitetään syntaksianalyysille. Syntaksianalyysi muodostaa sanojen tunnisteista tietorakenteen nimeltä jäsennyspuu. Seuraavaksi sementtinen analyysi tarkastaa syntaksipuista, että jokaisella operaattorilla on vastaavat operandit. Analyysivaiheen viimeisenä osuutena on luoda välikoodi, jonka muodostaa välikoodin generointi.

Synteesivaiheen tehtävänä on luoda lopullinen kohdekoodi. Myös synteesivaihe jakautuu pienempiin osiin analyysivaiheen tapaisesti. Nämä osat ovat välikoodin optimointi, sekä kohdekoodin generointi ja optimointi. Kohdekoodin generointi luo analyysivaiheelta saadusta välikoodista lopullisen kohdekoodin. Tästä kohdekoodista saadaan tehokkaampi synteesivaiheeseen kuuluvilla optimoinneilla.

Työn tavoitteena oli tutkia aiheeseen liittyvää kirjallisuutta ja muita tieteellisiä tekstejä apuna käyttäen, kuinka ohjelmointikielen kääntäjä toimii. Tarkoituksena on luoda yleiskatsaus kääntäjän toiminnasta ja siihen kuuluvista vaiheista.

Avainsanat: Kääntäjä, ohjelmointikielen kääntäjä, lähdekoodi, kohdekoodi, symbolitaulukko, analyysivaihe, synteesivaihe, leksikaalinen analyysi, syntaksianalyysi, semanttinen analyysi, välikoodi.

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

Kääntäjä	Compiler, käännöksen tekevä ohjelma
Konekieli	Machine code, tietokoneen suorittimen ymmärtämä ohjelmointikieli
Välikoodi	Intermediate code, analyysivaiheen tuottama käännöksen välivaihe
Lähdekoodi	Source code, ohjelmakoodi, jonka kääntäjä kääntää
Lähdeohjelma	Source program, lähdekoodilla kirjoitettu ohjelma
Kohdekoodi	Target code, kääntäjän tuottama lopullinen ohjelmakoodi
Kohdeohjelma	Target program, kohdekoodilla kirjoitettu ohjelma
Kohdekone	Target machine, kone, jolle kohdeohjelma on tehty
Tietorakenne	Datastructure, rakenne, johon voi tallentaa dataa ja käsitellä sitä
Puu/puurakenne	Tree, tietorakennetyyppi, joka muodostuu toisiinsa linkitetyistä solmuista
Juuri	Root, Puun korkein solmu, jolla ei ole vanhempia (ylöspäin linkattuja solmuja)
Lehti	Leaf, Puun solmut, joilla ei ole lapsia (alaspäin linkattuja solmuja)
Sisäsolmu	Interior node, puun solmut, joilla on yksi tai enemmän lapsisolmuja
Lineaarinen lista	Linear list, lineaarisen rakenteen omaava tietorakenne
Hajautustaulu	Hash table, tietorakenne, jossa kaikella tallennetulla datalla on oma indeksiarvonsa, jolla tietyn datan löytää nopeasti
Binäärinen hakupuu	Binary search tree, puurakenne, jossa
Asymptoottinen suoritusaika	Asymptotic notation, kuvaa algoritmin suoritusaikaa, kun käsitellyn datan määrä kasvaa
O-notaatio	O-notation, kuvaa suoritusajan pahinta mahdollista tapausta
Lekseemi	Lexeme, lähdekoodissa esiintyvät merkit ja nimet
Tunniste	Token, leksikaalisen analyysin tuottama kaksiosainen tunniste lekseemistä

1. JOHDANTO

Suuri osa ohjelmointikielistä tarvitsee kääntäjän (engl. compiler) tai tulkin (engl. interpreter) toimiakseen. Tämä johtuu siitä, että näillä ohjelmointikielillä toteutettu lähdekoodi ei pysty suoraan välittämään tietokoneelle, mitä sen on tarkoitus suorituksen aikana tehdä. Näitä ohjelmointikieliä kutsutaan korkean tason ohjelmointikieliksi ja ne ovat tehty helposti ymmärrettäviksi ohjelmoijille, mutta eivät tietokoneille. Tästä syystä korkean tason ohjelmointikielillä kirjoitettu lähdekoodi tarvitsee kääntää matalan tason ohjelmointikielille, joita myös tietokoneet ymmärtävät. Toisin sanoen kääntäjä ja tulkki mahdollistavat ohjelman suorittamisen tietokoneella.

Ohjelmointikielen kääntäjiä ja tulkkeja on monenlaisia, mutta tässä työssä keskitytään vain kääntäjiin, jotka kääntävät korkean tason ohjelman konekielelle. Kääntäjä koostuu useasta eri vaiheesta, joilla on oma tehtävänsä käännöksessä. Nämä vaiheet mahdollistavat kattavan käännöksen lähdekoodista konekielelle.

Tämän kandidaatintyön tarkoituksena on tutkia, miten ohjelmointikielen kääntäjät toimivat. Työssä kuvataan aiheeseen liittyvän kirjallisuuden, sekä tieteellisten tekstien avulla käännösprosessissa tapahtuvat vaiheet ja niiden osat.

Ensiksi työn 2. luvussa tarkastellaan, mitä korkean tason ohjelmointikielet ovat. Tämän jälkeen 3. luvussa otetaan käsittelyyn matalantason ohjelmointikielet yhdessä konekielen ja assembly-kielen kanssa. 4. luvussa käsitellään kääntäjän toimintaa. Siinä esitellään käännöksen eri vaiheet, sekä käännökseen kuuluvat eri prosessit. Lopuksi 5. luvussa on yhteenveto työstä.

2. KORKEAN TASON OHJELMOINTIKIELI

Puhuttaessa ohjelmointikielistä tarkoitetaan yleensä korkean tason ohjelmointikieliä. Esimerkiksi C++, Python ja Java kuuluvat korkean tason ohjelmointikieliin.

Korkean tason ohjelmointikielet ovat lähempänä ihmisten luonnollista kieltä sekä korkeammalla abstraktiotasolla kuin matalan tason ohjelmointikielet [1, s. 30–31]. Tästä syystä ihmisten on helpompi lukea, ymmärtää, kirjoittaa ja oppia niitä. Korkean tason ohjelmointikieliä luodaan siis siitä syystä, että ohjelmoinnista saadaan helpompaa ja luontevampaa [2, s. 13].

Korkean tason ohjelmointikielet tarvitsevat erillisen ohjelman, jotta niillä toteutettu ohjelma voidaan suorittaa tietokoneella. Tällaisia ohjelmia ovat kääntäjät ja tulkit, jotka muuntavat ohjelmakoodin tietokoneelle ymmärrettävään muotoon eli bittitasolle. Tietokoneelle ymmärrettäviä ohjelmointikieliä ovat matalamman tason ohjelmointikielet, kuten konekieli ja assembly. Matalantason ohjelmointikieliä käsitellään luvussa 3.

Huonona puolena korkean tason ohjelmointikielissä on niiden tehottomuus verrattuna matalan tason ohjelmointikieliin. Käännetty korkean tason ohjelma toimii hitaammin kuin suoraan kirjoitettu matalan tason ohjelma, sillä matalan tason kieltä käyttävillä ohjelmoijilla on enemmän hallintaa laitteistotasolla. Lisäksi korkean tason ohjelman kääntäminen vie myös aikaa. Tästä huolimatta korkean tason ohjelmointikieliä suositaan niiden helppouden vuoksi. Kuitenkin tehokkuutta on mahdollista kompensoida kääntäjän optimoinnilla. [2, s. 17]

3. MATALAN TASON OHJELMOINTIKIELI

Aho et al. mainitsevat kirjassaan [2, s.13], että ohjelmointikielet voidaan luokitella sukupolvien mukaan. Matalan tason ohjelmointikielet kuuluvat ensimmäisen ja toisen sukupolven ohjelmointikieliin. Ensimmäisen sukupolven ohjelmointikielet, kuten konekielet, ovat suoraan ymmärrettävissä tietokoneelle, kun taas toisen sukupolven ohjelmointikielet eli assembly-kielet käyttävät assembler-ohjelmaa kääntyäkseen konekielelle [3]. Ne eivät siis tarvitse kääntäjää tai tulkkia toimiakseen.

Matalan tason ohjelmointikielet tarjoavat vain vähän tai eivät ollenkaan abstraktiota konekielestä. Tästä syystä matalan tason ohjelmaa on todella vaikea toteuttaa verrattuna korkean tason ohjelmaan. Kuitenkin abstraktion puutteen vuoksi matalan tason ohjelmat ovat muistitehokkaampia [3] ja nopeampia suorittaa tietokoneella.

3.1 Konekieli

Kuten luvun alussa mainittiin, konekieli kuuluu ensimmäisen sukupolven ohjelmointikieliin, joilla toteutetut ohjelmat ovat suoraan suoritettavissa prosessorilla. Tällöin minkäänlaista kääntäjää, tulkkia tai edes assembleria ei tarvita. Täten kaikki tietokoneella suoritettavat ohjelmat, huolimatta siitä millä ohjelmointikielellä ne on kirjoitettu, suoritetaan lopulta konekielellä [4].

Halpern kertoo tutkimusartikkelissaan [6, s.1044], että konekielestä on yhä hankalampi antaa yksinkertaista ja tarkkaa määritelmää, sillä erilaisten laitteiden määrä kasvaa jatkuvasti. Kuitenkin Halpernin mukaan konekielen määritelmäksi riittää, että se on ohjelmointikieli, jonka jokin määritetty koneen suoritin voi suorittaa ilman edeltävää ohjelmistokäännöstä ja jonka tyypillinen käsky koostuu operaattori-operandiparista. Koska tietokoneen koko toiminta perustuu bitteihin, nämä käskyt ovat kaikki binäärilukujen muodossa [3] [4].

Konekielisen käskyn operandiosa on binäärinen kokonaisluku, joka on kohdeosoite, johon pääsee käskyn operaattoriosan suorittamisen yhteydessä. Operaattoriosa kertoo käskyn toiminnon, joka voi olla esimerkiksi aritmeettinen operaatio tai käskyn sisällön liikuttaminen rekistereiden välillä. [6, s.1044]

```
000101000000000000000000000001100100
000100000000000000000000000001100101
000110000001000000000000000001100110
```

Kuva 1: Kolme konekielistä käskyä [6, s.1044–1045].

Halpern näyttää kirjoituksessaan [6, s.1044] esimerkin konekielisistä käskyistä (kuva 1). Kyseessä on binäärilukuja, joissa vasemman puolen 12 ensimmäistä bittiä on käskyn operaattoriosa ja oikean puolen 15 ensimmäistä bittiä on operandi-osa. Tässä tapauksessa ensimmäisen rivin käskyn operaattoriosa sisältää toiminnon, joka asettaa operandi-osasta saadun osoitteen sisällön tietokoneen laskuriin (engl. accumulator). Seuraavaksi toinen käsky summaa operandi-osastaan saadun sisällön laskurissa olevaan sisältöön. Lopuksi sisältöjen summa tallennetaan viimeisen käskyn operandi-osassa olevaan osoitteeseen. [6, s.1044–1045]

3.2 Assembly

Konekielestä seuraavan sukupolven ohjelmointikieli on assembly-kieli [2, s.13]. Koska assembly tarjoaa vain hieman abstraktiota konekielestä, sille riittää kääntäjäksi ohjelma nimeltä assembler. Assembler on hyvin yksinkertainen kääntäjä verrattuna tavalliseen kääntäjään, sillä se kääntää jo valmiiksi matalantason ohjelman konekielelle.

Ensimmäiset assembler-ohjelmat mahdollistivat konekäskyn operaattoriosalle symbolisen syntaksin, sekä operandi-osalle pitkän binääriluvun sijasta desimaali– tai oktaaliluvun. [6, s.1046] Kuvassa 2 on esitettynä assembly-kielen selkeämpi esitystapa kuvan 1 konekielisistä käskyistä.

```
CLA 100
ADD 101
STO 102
```

Kuva 2: Assembly-kielen syntaksi kolmelle konekieliselle käskylle [6, s. 1046].

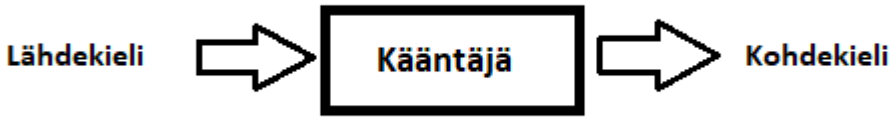
Tietokoneiden kehityksen alkuaikoina suurta suorituskykyä vaativat ohjelmistot kirjoitettiin assembly-kielellä. Tämä johtui siitä, että tarpeeksi hyvin optimoituja kääntäjiä korkean tason ohjelmointikielille ei ollut vielä kehitetty. Kuitenkin ajan kuluessa kääntäjien optimointi kehittyi merkittävästi ja niiden tuottaman koodin suorituskyky lähestyi jo assembly-kielellä toteutettua koodia. [5, s.2]

Nykyisin assembly-kieli on täysin symbolinen kieli. Kaikki käskyn operaatiot ja operandit ovat tavallisesti esitettynä niille ennalta määritellyillä nimillä. [6, s. 1046] Kuvassa 2 ensimmäisen konekielisen käskyn operaattoriosa on nimetty syntaksilla ”CLA”, toinen ”ADD”, sekä viimeinen ”STO”. Tästä huomataan, että operaattoriosien nimet kuvaavat niiden toiminnallisuutta. Esimerkiksi nimitys ”ADD” (addition) tarkoittaa lisäystoimintoa ja ”STO” (storage) tallennustoimintoa [6, s.1044–1046]. Käskyjen operandi-osat ovat desimaalilukuina, mutta ne voisivat myös olla nimettyinä operaattoriosien tapaan [6, s.1046].

Käännös konekielelle tapahtuu assemblerilla siten, että jokaiselle assembly-kielen symboliselle esitystavalle löytyy suora konekielinen, eli numeerinen vastaavuus. Aina kun assembler kohtaa symbolisen osoitteen ohjelmassa, se korvaa symbolin binääriekvivalentilla [6, s. 1047].

4. KÄÄNTÄJÄN VAIHEET

Kääntäjä on ohjelma, joka pystyy lukea jollakin ohjelmointikielellä toteutetun ohjelman ja kääntää sen jollekin toiselle kielelle, niin että ohjelman toiminnallisuus säilyy samana. Ohjelmointikieltä, jolla kääntäjän lukema lähdeohjelma on toteutettu, kutsutaan lähdekieleksi (engl. source language). Kun taas ohjelmointikieltä, jolle ohjelma tullaan kääntämään, kutsutaan kohdekieleksi (engl. target language).



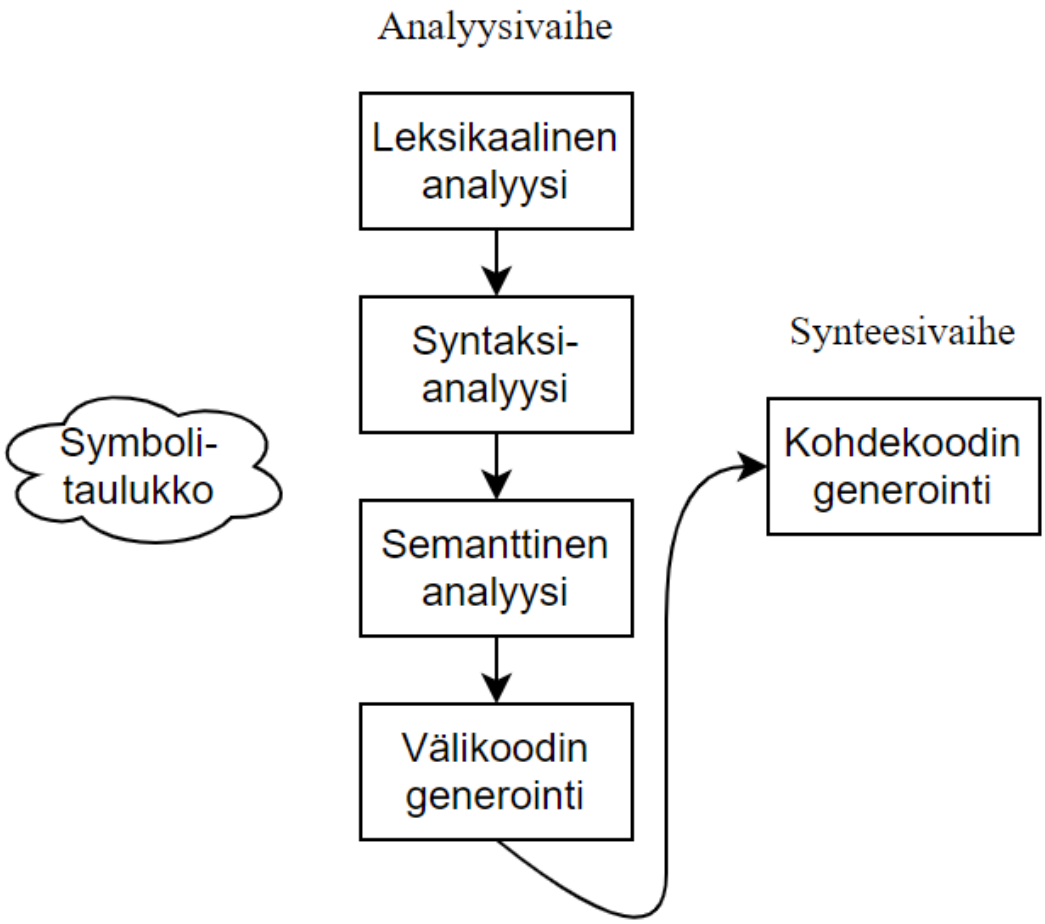
Kuva 3: Kääntäjän perusidea.

Kokonaisuudessaan käännösprosessi, jossa lähdekielestä päädytään kohdekielelle, koostuu useista eri vaiheista. Tässä luvussa käsitellään kaikki nämä vaiheet. Nämä vaiheet tyypillisesti jaetaan kahteen suurempaan vaiheeseen, joita kutsutaan synteesivaiheeksi ja analyysivaiheeksi [2, s.4]. Toinen yleinen nimitys näille vaiheille on front end ja back end [2, s.4] [7, s.251]. Nimensä mukaisesti analyysivaihe vastaa lähdekoodin analysoinnista, ja synteesivaihe kohdekoodin luomisesta ja syntesoinnista [7, s.251].

Analyysivaiheen päätehtävänä on luoda väliesitys lähdekoodista (engl. intermediate representation), joka välitetään synteesivaiheelle. Väliesitystä kutsutaan myös välikoodiksi (engl. intermediate code) [2] [8] [9]. Jotta välikoodi saadaan virheettömästi välitettyä, analyysivaiheeseen kuuluvat pienemmät prosessit suorittavat virheiden tarkastusta ja tuottavat informatiivisia virheviestejä käyttäjälle. [2, s.4] [7, s.251–252] Kuvan 4 mukaisesti analyysivaiheen prosesseihin kuuluu leksikaalinen analyysi, syntaksi analyysi, semanttinen analyysi ja välikoodin generointi.

Synteesivaiheen tarkoituksena on luoda lopullinen kohdeohjelma analyysivaiheelta saadusta välikoodista. Ennen kohdeohjelman luomista synteesivaiheeseen kuuluu myös vaiheita, joissa optimoidaan väliesityksen koodia, jotta kohdeohjelmasta saadaan tehokkaampi. [2, s.5–10] [7] On kuitenkin myös mahdollista, että yksinkertaisimmassa tapauksessa kohdeohjelma voidaan rakentaa suoraan välikoodista [7, s.256].

Kuvassa 4 nähdään kääntäjän tyypillinen rakenne.



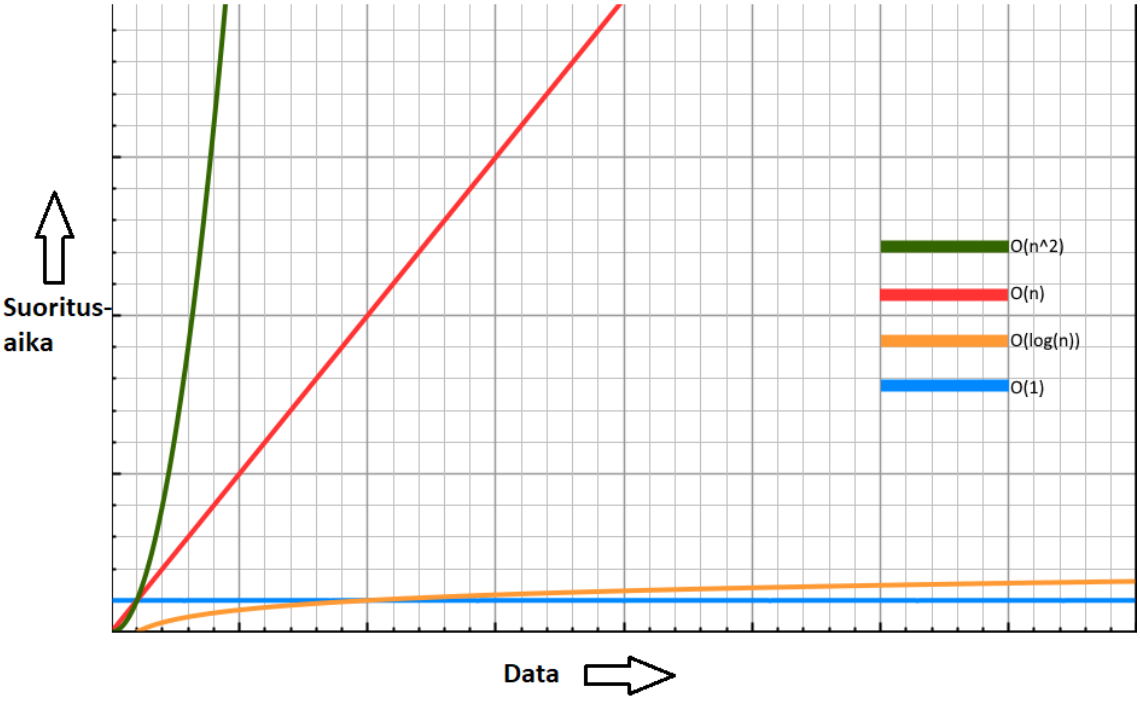
Kuva 4: Kääntäjän rakenne (pohjautuu lähteisiin [2] [8] [9]).

4.1 Symbolitaulukko

Väliesityksen rakentamisen, sekä virheviestien välittämisen lisäksi analyysivaiheen tehtäviin kuuluu myös tallettaa tietoa lähdekoodista. Tietoa tallennetaan kääntäjässä käytettävään tietorakenteeseen nimeltä symbolitaulukko (engl. symbol table), jota käytetään käännösprosessin molempina vaiheina. [2, s.4–11] Symbolitaulukkoon tallennettu tieto sisältää lähdeohjelmassa käytettyjen entiteettien, kuten muuttujien ja funktioiden nimet, tyypit ja palautusarvot [9, s.479]. Toisin sanoen symbolitaulukko pitää sisällään entiteettien semantiikan, eli merkityksen.

Koska symbolitaulukko kulkee käännöksen mukana, sekä analyysivaiheessa, että synteesivaiheessa, täytyy tiedon hakeminen ja tallentaminen olla nopeaa [2, s.11]. Tästä syystä sopivan tietorakenteen valitseminen on erityisen tärkeää kääntäjän suunnittelussa. Tietorakenne voi olla esimerkiksi lineaarinen lista, hajautustaulu tai binäärinen hakupuu. Listaa käytetään yleensä tapauksessa, jossa tallennettavaa dataa on hyvin vähän, kun taas hajautustaulu on näistä kaikista yleisimmin käytetty tietorakenne [8, s.102]. [10]

Voidaan päätellä, että hajautustaulun suosio kääntäjissä johtuu muun muassa sen tehokkuudesta hakea ja tallentaa dataa. Tutkimalla edellisessä kappaleessa mainittujen tietorakenteiden asymptoottisia suoritusaikoja, huomataan, että normaalitapauksessa hajautustaulu on tehokkain [11]. Datan haku ja tallentaminen ovat hajautustaulussa vakioaikaisia [11]. Tämä tarkoittaa sitä, että näiden operaatioiden tehokkuus on riippumaton datan määrästä. Eli hajautustaulu säilyy tehokkaana myös suurilla datamäärillä.



Kuva 5: O-notaatioita.

Taulukko 1: Erilaisten symbolitaulukoiden O-notaatiot keskeisille operaatioille.

	Lista (linkitetty lista)	Binäärinen hakupuu	Hajautustaulu
Haku	O(n)	O(log(n))	O(1)
Tallennus	O(1)	O(log(n))	O(1)

Yksi tapa kuvata tietorakenteiden suorituskykyä ovat O-notaatiot. Ne kuvaavat sitä, kuinka monta toimintoa maksimissaan joudutaan suorittamaan, kunnes haluttu operaatio on valmis [12]. Kuvasta 5 nähdään, että toimintojen määrä riippuu tietorakenteessa olevan datan määrästä, ellei kyseessä ole vakioaikainen notaatio O(1). Esimerkiksi lineaarisella notaatiolla O(n) toimintojen määrä on suoraan verrannollinen datan määrään. Taulukkoon 1 on koottu symbolitaulukkoina käytettävien tietorakenteiden O-notaatiot.

4.2 Leksikaalinen analyysi

Kääntäjän analyysivaihe alkaa leksikaalisella analyysilla (engl. lexical analysis), jonka ensitehtävänä on pilkkoa lähdekoodin merkkivirrat lekseemeiksi (engl. lexeme). Lekseemejä ovat esimerkiksi numerot, aritmeettiset merkit, muuttujiennimet, välimerkit, sekä ohjelmointikielille ominaiset avainsanat (engl. keyword) (esim. for tai while). Jokaisesta lekseemistä muodostetaan tunnisteet (engl. token), jotka toimivat leksikaalisen analyysin ulostulona. Tunniste on rakennettu pariksi, joka koostuu tunnisteen nimestä ja valinnaisesta arvosta. [2, s.6, s.109–114] [9, s.3]

$$\langle token\text{-}name, attribute\text{-}value \rangle$$

Kuva 6: Tunnisteen rakenne [2, s.6].

Tunnisteen nimi kuvaa mihin kategoriaan tunniste kuuluu [2, s.111]. Esimerkiksi aritmeettiset operaattorit ja käyttäjän määrittämät nimet kuuluvat omiin kategorioihinsa. Tällaisia kategorioita on useampia, mutta tyypillisimpiä on kirjattu taulukkoon 2, jossa on C++ -kielessä käytettäviä tunnisteita.

Tunnisteen arvon avulla saadaan lisää informaatiota lekseemistä, josta tunniste on muodostettu. Informaatio sisältää esimerkiksi tunnistetta vastaavan lekseemin, sen tyypin sekä paikan lähdekoodissa. Informaatiota tallennetaan symbolitaulukkoon, josta sen löytää tunnisteen arvolla. [2, s.112] Arvo toimii siis osoittimena sitä vastaavan tunnisteen informaatioon symbolitaulukossa [2, s.6] [8, s. 102]. Informaatio on hyvin tärkeä kääntäjän myöhemmissä vaiheissa, kuten kohdekoodin generoinnissa, jossa ei riitä tieto tunnisteen nimestä, vaan täytyy tietää myös mikä lekseemi on kyseessä [2, s.112].

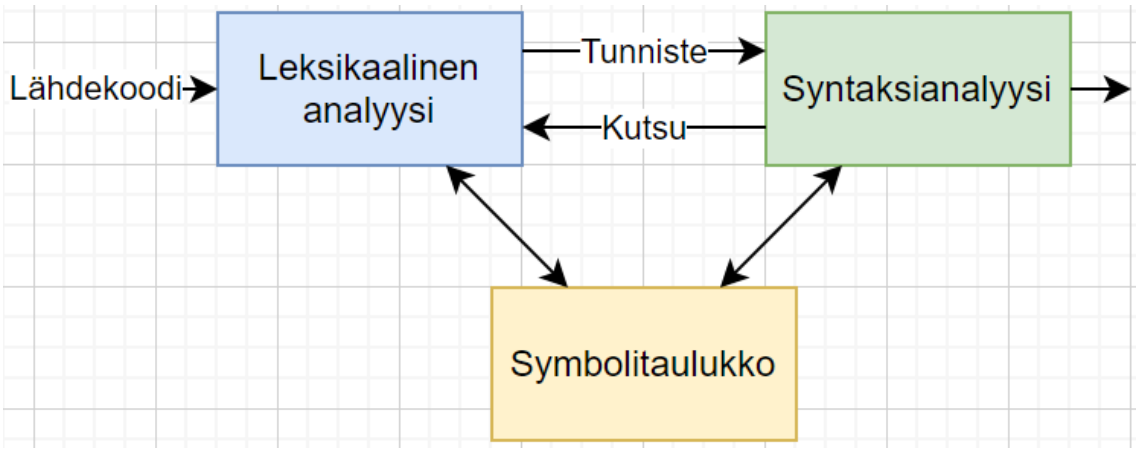
$$\&Symbol\_table[identifier\_name]$$

Ohjelma 1: Esimerkki tiedon hausta symbolitaulukosta [8, s.102].

Taulukko 2: C++ -kielessä käytettäviä tunnisteita (pohjautuu lähteisiin [2, s.112] [13]).

Tunnisteen nimi		Tunnisteen arvo (esimerkkejä)
Keyword	Ohjelmointikielen avainsanat, joilla ennalta määritelty merkitys	While, for, if, else, int
Identifier	Ohjelmoijan määrittelemät nimet esimerkiksi muuttujille ja funktioille	X, y, result
Operator	Erilaiset laskuoperaattorit ja vertailumerkit	=, +, -, *, <, >
Punctuator	Erilaiset ohjelmointikielelle ominaiset välimerkit ja erottimet.	{, }, ;
Literal	Literaali edustaa suoraan jotakin arvoa, joka voi olla esimerkiksi luku, totuusarvo tai merkkijono	True, false, 15, ”color”
Number	Mikä tahansa numeerinen vakio	1, 3.14, 0

Leksikaalinen analyysi toimii yhdessä syntaksianalyysin kanssa, joka on kääntäjän toinen vaihe. Leksikaalisen analyysin ulostulo eli tunniste toimii syntaksianalyysi sisääntulona. Tavallisesti syntaksianalyysi kutsuu leksikaalista analyysiä aina kun se tarvitsee uutta tunnistetta. [2, s.109–110] [9, s.3]



Kuva 7: Leksikaalisen analyysin toiminta syntaksianalyysin ja symbolitaulukon kanssa (pohjautuu lähteeseen [2, s.110]).

Lähdekoodi sisältää myös paljon leksikaalisen analyysin kannalta turhia asioita, joista ei haluta muodostaa tunnisteita. Näitä ovat esimerkiksi kommentit ja kaikki tyhjä tila, kuten välilyönnit ja tyhjät rivit. Leksikaalisen analyysin tehtäviin kuuluu näiden poistaminen. [2, s.110] Poikkeuksena on kuitenkin merkkijonoissa (engl. string) esiintyvät välilyönnit, sillä tässä tapauksessa välilyönnin poistaminen muuttaisi merkkijonon merkitystä [8, s.61].

Leksikaalinen analyysi osallistuu myös kääntäjän virheilmoituksien tekemiseen. Vaikka lähdekoodin tyhjää tilaa poistetaankin, leksikaalinen analyysi voi pitää kirjaa esimerkiksi tyhjistä riveistä, jotta virheilmoitukset saadaan oikeille riveille. [2, s.110] [8, s.61]

$$result = x + y * 5;$$

Ohjelma 2: Yksinkertainen matemaattinen operaatio

Seuraavaksi tutkitaan lähteiden [2] ja [14] pohjalta, miten leksikaalinen analyysi pilkkoo ohjelman 2 tunnisteiksi. Ensiksi erotetaan ohjelmasta lekseemit, joista tunnisteet muodostetaan. Lekseemit ovat kirjattu taulukkoon 3.

Taulukko 3: Ohjelman 2 lekseemit.

Tunnisteen nimi	Lekseemit	Tunnisteen arvot
Identifier	result, x, y	”result”, ”x”, ”y”
Operator	=, +, *	=, +, *
Number	5	5
Punctuator	;	;

Taulukko 4: Ohjelman 2 tunnisteet.

< Identifier, ”result” >
< Operator, = >
< Identifier, ”x” >
< Operator, + >
< Identifier, ”y” >
< Operator, * >
< Number, 5 >
<Punctuator, ; >

Taulukossa 3 lekseemeistä koottujen tunnisteiden nimet ovat valittu taulukon 2 selitteiden mukaisesti. Näissä taulukoissa arvo on sama kuin itse lekseemi, jolloin haettaessa informaatiota esimerkiksi identifier-tunnisteista, arvo toimii avaimena symbolitaulukossa olevaan osoittimeen. Osoittimella päästään käsittelemään avainta vastaavan tunnisteen informaatiota. [2, s.88] [8, s.102] Yksi tapa hakea symbolitaulukosta tietoa on esitetty ohjelmassa 1, jossa identifier_name on tunnisteen arvo.

Taulukon 3 perusteella saadaan muodostettua ohjelmalle 2 tunnisteet. Tunnisteet ovat esitetty taulukossa 4.

4.3 Syntaksianalyysi

Kääntäjän toinen vaihe on syntaksianalyysi, jota kutsutaan myös jäsentäjäksi (engl. parser). Sen tehtävänä on muodostaa leksikaaliselta analyysiltä saaduista tunnisteista ohjelman hierarkkinen rakenne. [2, s.8, s.41] [9, s.4] Toisin sanoen syntaksianalyysi jäsentää analyysivaiheen tunnistevirran. Kuvassa 7 on esitettynä leksikaalisen analyysin ja syntaksianalyysin toiminta yhdessä.

Tavallisesti syntaksianalyysi kuvaa ohjelman rakennetta puurakenteilla. [2, s.8, s.192] Rakennetta voidaan myös kuvata muillakin tietorakennetyypeillä, kuten syntaksidiagrammilla [9, s.4], mutta tässä työssä keskitytään vain puurakenteisiin. Tämä johtuu siitä, että työssä käytetyissä lähteissä käsitellään lähinnä vain puurakenteita, josta voidaan päätellä niiden olevan yleisimpiä.

Syntaksianalyysissä muodostettavaa puurakennetta kutsutaan tavallisesti syntaksi– tai jäsennyspuuksi. Nämä kuitenkin jossain määrin eroavat toisistaan. Jäsennyspuulla tarkoitetaan enemmän puuta, joka on jäsennetty tarkasti tietyn kieliopin mukaan, kun taas syntaksipuut eivät välttämättä sisällä kaikkia kielioppiin sisältyviä komponentteja. [2, s.69–70] [8, s.9–10]. Tästä syystä työssä tullaan käyttämään nimitystä jäsennyspuu, kun käsitellään puurakenteen muodostamista. Ohjelmoinnin kielioppia ja jäsennyspuita tarkastellaan lisää seuraavissa luvuissa (4.3.1 ja 4.3.2).

4.3.1 Kontekstiton kielioppi

Jokaiselle ohjelmointikielelle on määritelty säännöt, jotka määräävät ohjelmien kieliopin. [2, s.191] Kielioppi määrittelee lähdeohjelman hierarkkisen rakenteen [2, s.42, s.106] [14]. Koska syntaksianalyysin tehtävänä on kuvata tätä rakennetta, se käyttää lähdeohjelman kielioppia rakentaakseen jäsennyspuun.

Lähdekirjallisuuden [2, s.42–43] [8, s.35] [9, s.167] mukaan kontekstittomaksi kieliopiksi (engl. context-free grammar) kutsutaan kielioppia, joka koostuu neljästä komponentista:

1. Tunnisteet, jotka ovat käsitelty luvussa 4.2, ovat ensimmäinen komponentti. Kuitenkin syntaksianalyysissä niitä kutsutaan päätetunnuksiksi (engl. terminal symbol).
2. Välitunnukset (engl. nonterminal symbol) ovat symboleja, jotka edustavat tiettyä joukkoa päätetunnuksia. Esimerkiksi välitunnus ’operator’ voisi edustaa joitain aritmeettisia merkkejä, kuten plus–, miinus– tai kertomerkkiä.

3. Päätetunnukset ja välitunnukset muodostavat produktioita (engl. production). Ne koostuvat vasemmasta ja oikeasta puolesta, joita erottaa nuoli. Vasemmalla puolella on yksi välitunnus, kun taas oikealla puolella on useampia välitunnuksia ja päätetunnuksia. On myös mahdollista, että oikealla puolella ei ole yhtään väli- tai päätetunnusta. Ahon mukaan vasenta puolta voidaan kutsua myös produktion pääksi (engl. head), ja oikeaa puolta kehoksi (engl. body).
4. Viimeisenä komponenttina on aloitussymboli (engl. start symbol). Se on produktion vasemmalla puolella oleva välitunnus.

```
expression → expression '+' term | term
term → term '*' factor | factor
factor → digit | '(' expression ')'
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Kuva 8: Esimerkki produktioista (pohjautuu lähteisiin [8, s.10, s.13] [9, s.180]).

Kuvassa 8 on neljä produktiota, jotka edustavat kontekstitonta kielioppia. Kuvan produktioissa nuolen vasemmalla puolella on välitunnus ja oikealla puolella sekä välitunnuksia että päätetunnuksia. Lähdekirjallisuuden [8, s.10, s.13] ja [9, s.180] mukaisesti merkki | erottaa eri vaihtoehdot johon produktion vasen puoli voi muokkautua. Lisäksi produktioiden päätemerkit ovat merkitty heittomerkeillä, kuten Grune et al. [8, s.10, s.13] kielioppiesimerkeissään.

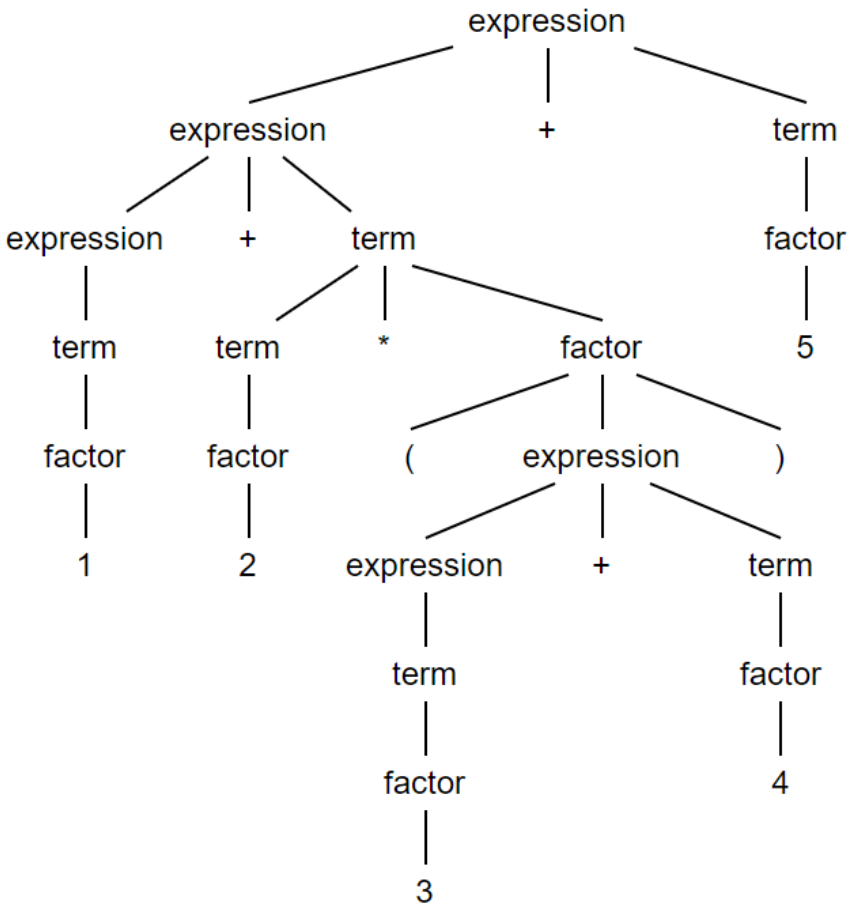
4.3.2 Jäsennyspuu

Kuten luvussa 4.3 jo todettiin, syntaksianalyysin ulostulona saadaan jäsennyspuu (engl. parse tree), joka kuvaa ohjelman rakennetta. Jäsennyspuu rakentuu kontekstittoman kieliopin komponenteista seuraavasti:

1. Jäsennyspuun juuri on aloitussymboli.
2. Jäsennyspuun lehdet ovat päätetunnuksia.
3. Jäsennyspuun sisäsolmut ovat välitunnuksia.
4. Sisäsolmulla ja sen lapsisolmuilla täytyy olla yhteinen produktio. Esimerkiksi sisäsolmun A ja sen lapsisolmujen X_1 ja X_2 produktio olisi $A \rightarrow X_1\ X_2$.

[2, s.45–46] [8, s.117]

Näiden piirteiden perusteella kuvan 8 kieliopilla on muodostettu jäsennyspuu komennolle 1+2* (3+4) +5, joka on esitetty kuvassa 9. Prosessoidessa tätä jäsennyspuuta alhaalta ylöspäin saadaan operaatio (3+4) suoritettua ensin. Tämän jälkeen se kerrotaan luvulla 2 ja lisätään luku 1. Lopuksi lisätään luku 5. Tällaista puun läpikäyntiä kutsutaan syvyyssuuntaiseksi läpikäynniksi (engl. depth-first traversal) [9, s.180].

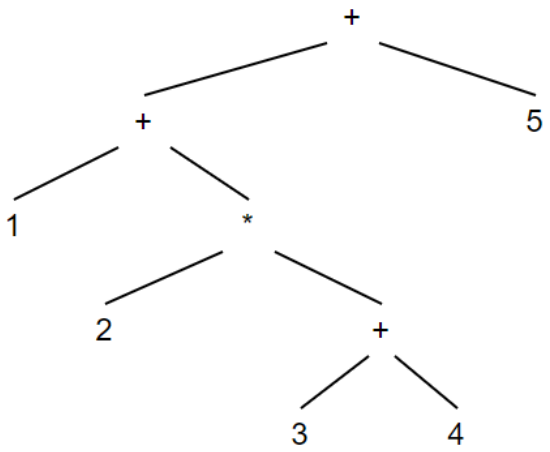


Kuva 9: Jäsennyspuu (pohjautuu lähteeseen [9, s.181]).

Jäsennyspuun rakentamiseen liittyy myös erilaisia menetelmiä, joita kutsutaan jäsennysmenetelmiksi. Näistä taalallisimpia ovat top-down ja bottom-up -menetelmät. Top-down -menetelmässä jäsennyspuu muodostuu juuresta lehtiin, kun taas bottom-up -menetelmässä lehdistä juureen. Tässä työssä ei kuitenkaan käsitellä jäsennysmenetelmiä tämän enempää.

Jäsennyspuu ei kuitenkaan ole kaikista käytännöllisin puu ottaen kääntäjän seuraavat vaiheet huomioon, sillä se sisältää niille turhaa tietoa, kuten välitunnukset ja erottimet. [8, s.10–11] [9, s.181] Ottamalla nämä tiedot pois puusta, saadaan abstrakti syntaksi-

puu (engl. abstract syntax tree), joka on esitetty kuvassa 10. Tämä abstrakti syntaksipuu on siis muodostettu saman kieliopin ja komennon mukaisesti kuin kuvan 9 jäsennyspuu.



Kuva 10: abstrakti syntaksipuu (pohjautuu lähteeseen [8, s.11] [9. s.181]).

Abstraktissa syntaksipuussa sisäsolmut vastaavat operaattoreita (esimerkiksi kuvassa 10 plusmerkkiä tai kertomerkkiä), kun taas sisäsolmujen lapset edustavat operaattorin operandeja. [2, s.69–70] Esimerkiksi kuvassa 10 abstraktin syntaksipuun juurena toimii operaattori +, ja juuren vasemmasta lapsesta muodostuvat alipuut edustavat komentoa 1+2* (3+4). Juuren oikeanpuoleinen lapsi edustaa numeroa 5. Näistä solmuista saadaan syvyyssuuntaisella läpikäynnillä muodostettua koko komento 1+2* (3+4) +5.

4.4 Semanttinen analyysi

Analyysivaiheen viimeinen osa ennen välikoodin generointia on semanttinen analyysi. Tämän tehtävänä on suorittaa tyyppitarkastus (engl. type checking) syntaksianalyysin muodostamalle jäsennyspuulle. Tyyppitarkastus etsii jäsennyspuusta yhteensopimattomuuksia. Eli se tarkastaa, että jokaisella jäsennyspuun operaattorilla on sitä vastaavat operandit, sekä väärää tietotyyppiä tai alustamattomia muuttujia ei käytetä. Toisin sanoen tyyppitarkastus varmistaa, että jokainen jäsennyspuussa oleva määre on oikean tyyppinen siihen, missä kontekstissa sitä käytetään. [2, s.8, s.24, s.98]

Jos semanttinen analyysi havaitsee edellä mainittuja virheitä, se välittää käyttäjälle virheilmoituksia. [2, s.24] Esimerkiksi ohjelmassa 3 if-lauseen väitteen oletetaan olevan boolean-tyyppiä, eli määritelty todeksi tai epätodeksi. Jos näin ei ole, semanttinen analyysi välittää virheilmoituksen.

$$If\ (VÄITE)\ \{\ \}$$

Ohjelma 3: Esimerkki if-lauseesta (pohjautuu lähteeseen [2, s.98])

Lisäksi Aho et al. mukaan [2, s.9] semanttinen analyysi lisää ohjelmasta 2 tehtyyn jäsennyspuuhun operaation, joka muuntaa luvun 5 liukuluvuksi (engl. floating-point number) jos sillä kerrottava muuttuja x on liukuluku. Tämä johtuu siitä, että kerrottaessa kokonaisluku liukuluvulla, vastauksesta tulee liukuluku.

4.5 Välikoodin generointi

Kuten luvun 4 alussa mainittiin, kääntäjän analyysivaiheen päätehtävänä on luoda välikoodi, joka välitetään synteesivaiheelle. Välikoodin generointi suorittaa tämän tehtävän loppuun ja on täten analyysivaiheen viimeinen osa.

Välikoodista muodostetaan synteesivaiheessa kohdekoodi, joten sen täytyy olla helposti tulkittavissa. Tyypillistä välikoodin muotoa kutsutaan three-address-koodiksi (engl. three-address code), joka muistuttaa assembly-kieltä (kuva 2). Three-address-koodin jokainen rivi on assembly-kielen tyylinen käsky, jossa on kolme operandia poikkeuksia lukuun ottamatta. [2, s. 9] Kuvassa 11 on esimerkki three-address-koodista, joka on tehty ohjelmalle 2.

Aho et al. mukaan operandien määrän lisäksi three-address-koodi noudattaa kahta muuta sääntöä. Käskyn oikealla puolella on enintään yksi operaatio, joka varmistaa sen, että operaatiot suoritetaan oikeassa järjestyksessä. Esimerkiksi kuvassa 11 kertolasku suoritetaan ennen lisäystä. Lisäksi kääntäjän täytyy luoda väliaikainen muuttuja jokaiselle arvolle, kuten R1, R2 tai R3 kuvassa 11.

```
R1 = inttofloat(5)
R2 = id3 * R1
R3 = id2 +R2
id1 = R3
```

Kuva 11: Esimerkki three-address-koodista ohjelmalle 2 (pohjautuu lähteeseen [2, s.9])

Kuvassa 11 ensimmäinen käsky muuntaa luvun 5 liukuluvuksi ja tallentaa sen muuttujaan R1. Seuraava käsky kertoo muuttujan R1 id3:lla, joka on ohjelman 2 muuttuja y. Kertolaskun tulos tallennetaan muuttujaan R2. Tämän jälkeen kertolaskun tulos R2 lisätään arvoon id2, joka on ohjelman 2 muuttuja x. Lopuksi asetetaan id1:lle eli ohjelman 2 muuttujalle ’tulos’ oikea arvo, joka on edellisen käskyn tulos R3.

4.6 Välikoodin optimointi

Koska synteesivaiheessa välikoodista muodostetaan kohdekoodi, sitä optimoimalla saadaan myös kohdekoodista laadukkaampi [9, s.2]. Tämä tarkoittaa sitä, että kohdekoodista tulee lyhyempi, nopeampi ja tehokkaampi. [2, s.10]

Esimerkiksi kuvan 11 välikoodi voidaan optimoida siten että inttofloat-operaatio jätetään tekemättä. Välikoodin optimoija pystyy päättelemään, että kokonaisluku 5 voidaan muuttaa liukuluvuksi 5.0 käännöshetkellä, joten inttofloat-operaatio voidaan jättää tekemättä kokonaan. [2, s.10] Tällöin jäsennyspuuhunkaan ei tarvitse lisätä tätä operaatiota ja three-address-koodistakin vähenee käskyjen määrä. Optimoitu välikoodi on esitetty kuvassa 12.

```
R1 = id3 * 5.0
id1 = id2 + R1
```

Kuva 12: Optimoitu versio kuvan 11 välikoodista (pohjautuu lähteeseen [2, s.10])

Välikoodin optimointi -vaihetta kutsutaan myös koneesta riippumattomaksi optimoinniksi (engl. machine-independent optimization, sillä sen ideana on vain tehdä muutoksia välikoodiin välittämättä siitä, millaiselle koneelle kohdekoodi on suunnattu. Välikoodin optimointi ei myöskään ole välttämätön kääntäjän toiminnan kannalta. Sitä käytetään silloin kun kohdekoodista halutaan tehdä mahdollisimman suorituskykyinen. [2, s.5] [2, s.505] Grune et al. mukaan [8, s.316] optimointivaiheet ovatkin kääntäjän lisäominaisuuksia ja ne toteutetaan viimeisenä kääntäjää rakentaessa.

4.7 Kohdekoodin generointi

Kohdekoodin generointi on kääntäjän rakenteen viimeinen vaihe, jos kohdekoodin optimointia ei tehdä. Kohdekoodin generoinnin tehtävänä on muodostaa analyysivaiheelta saadusta välikoodista lopullinen kohdekoodi. Tavallisesti kohdekoneen arkkitehtuuri määrää, miten kohdekoodin generointi tehdään. Vaikka generointitapoja on useita, kohdekoodi rakennetaan aina kolmen päätehtävän kautta:

1. Käskyjen valikointi (engl. instruction selection)
2. Rekisterin allokointi (engl. register allocation)
3. Käskyjen järjestäminen (engl. instruction scheduling)

[2, s.505–511] [8, s.317–319]

Näiden tehtävien monimutkaisuus riippuu välikoodin rakenteesta, kohdekoneen arkkitehtuurista ja siitä kuinka laadukas kohdekoodi halutaan generoida. Mitä matalamman tason välikoodi on kyseessä, sitä helpompaa on generoida tehokas kohdekoodi. [2, s.505–511] Lisäksi tehtäviin liittyy useita eri strategioita, mutta tässä työssä ei käsitellä niitä. Tarkoitus on vaan kuvata niiden rooli kohdekoodin generoinnissa.

4.7.1 Käskyjen valikointi

Käskyjen valikoinnin tehtävä on kartoittaa välikoodi koodiketjuksi, jonka kohdekone pystyy suorittamaan. Eli kaikki välikoodin käskyt muutetaan kohdekielen mukaisiksi käskyiksi. Koska työn tarkoituksena on muuttaa lähdekoodi konekielelle, three-address-koodi voidaan muuntaa yksinkertaisesti assembly-kielelle, josta muunnos konekielelle

onnistuu helposti assemblerilla. [2, s.508–510] Esimerkiksi kuvan 11 tai 12 three-address-koodille saadaan assembly-kielinen koodiketju, joka on esitetty kuvassa 13.

Kuvassa 13 koodiketjun ensimmäinen rivi tallentaa luvun 5.0 rekisteriin R1, jonka jälkeen se kerrotaan id3:lla ja tallennetaan taas rekisteriin R1. Tämän jälkeen R1, jossa on kertolaskun vastaus, summataan muuttujaan id2 ja tallennetaan taas samaan rekisteriin. Lopuksi rekisteri R1 säilötään muuttujaan id1. Tällä tavalla saadaan suoritettua haluttu ohjelma 2.

```
LD  R1,  5.0
MUL  R1,  R1,  id3
ADD  R1,  R1,  id2
STO  id1,  R1
```

Kuva 13: Koodiketju kuvan 11 tai 12 three-address-koodille (pohjautuu lähteeseen [2, s.509]

Täytyy kuitenkin ottaa huomioon, että kuvassa 13 on vain yksi koodiketjun toteutus ohjelmalle 2. Toteutuksia on tavallisesti monia ja niiden suorituskyvyt voivat erota toisistaan huomattavasti. [2, s.509]

4.7.2 Rekisterin allokointi

Rekisterin allokointi on prosessi, jossa päätetään mitkä välikoodin arvoista pidetään missäkin rekistereissä. Tavallisesti kohdekoneessa ei ole tarpeeksi rekistereitä kaikille välikoodin arvoille, joten osa arvoista täytyy pitää muistissa. Rekisterin allokointi siis päättää mitkä näistä arvoista menevät rekistereihin ja mitkä muistiin. [2, s.510–511]

Koodiketjun käskyt, jotka sisältävät rekistereitä muistin sijaan, ovat paljon tehokkaampia [2, s.510] [9, s.673]. Tämä johtuu siitä, että nykyaikaisissa koneissa prosessorinopeudet ovat suurempia kuin muistinopeudet. [2, s.553] Tästä syystä voidaan sanoa, että rekisterin allokoinnissa jokainen koodiketju pyritään muodostamaan käyttämällä mahdollisimman vähän rekistereitä, jotta niitä säästyy useammille arvoille.

4.7.3 Käskyjen järjestäminen

Kohdekoodin generoinnin kolmantena päätehtävänä on määritellä rakennetun koodiketjun käskyille järjestys. Tässä voidaan käyttää samaa järjestystä kuin välikoodin generoinnissa, sillä se tuottaa toimivan koodiketjun. Kuitenkin jos tavoitteena on generoida mahdollisimman suorituskykyinen kohdekoodi, täytyy ottaa huomioon, että jotkut käskyjärjestykset ovat tehokkaampia kuin toiset. [2, s.511–512] [2, s.710–711] [8, s.319, s.397]

4.8 Kohdekoodin optimointi

Kohdekoodin generointi -vaiheen tuottamaa koodia voidaan parantaa kohdekoodin optimoinnilla. Kohdekoodin optimointi on konekohtaista optimointia (engl. machine-dependent optimization), sillä kohdekoodi on generoitu kohdekoneen arkkitehtuurin mukaisesti.

Lähdekirjallisuuden [2] [8] [9] mukaan optimointia voidaan tehdä jo kohdekoodin generointi vaiheessa valitsemalla sopivat strategiat päätehtäville ja toteuttamalla ne huolellisesti. Aho et al. mainitsee, että useimmat kääntäjät tuottavat laadukkaan kohdekoodin tällä tavalla.

Jotkut kääntäjät generoivat kohdekoodin yksinkertaisemmin ja optimoivat sitä jälkikäteen erilaisilla optimointimenetelmillä, joita on esitetty lähdekirjallisuudessa monia. Yksi näistä on ovisilmäoptimointi (engl. peephole optimization). [2, s.549]

Ovisilmäoptimoinnissa pyritään poistamaan kaikki tarpeettomat käskyt kohdekoodin generoinnin rakentamasta koodiketjusta. Esimerkiksi kuvassa 14 tallettamiskäskyn ’STO’ voisi poistaa kokonaan, sillä muuttuja x on jo tallennettu rekisteriin R1 sitä edeltävässä käskyssä. [2, s.550–552]

```
LD  R1,  x
STO  x,  R1
```

Kuva 14: Esimerkki ovisilmäoptimoinnista (pohjautuu lähteeseen [2, s.550])

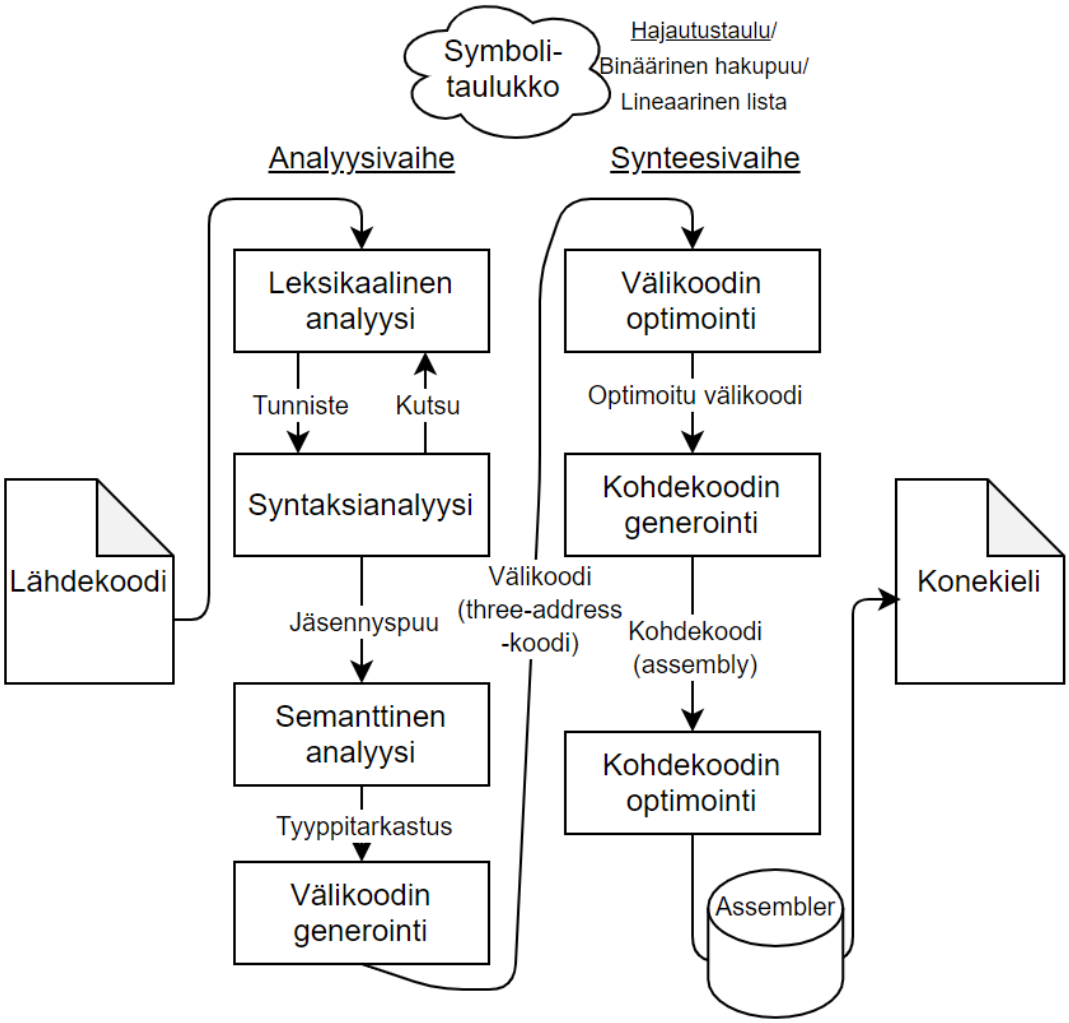
Ovisilmäoptimoinnilla voidaan myös muuttaa aritmeettisia operaattoreita tehokkaammiksi vastaavuuksiksi, sekä tarvittaessa poistamaan niitä. Aritmeettinen operaatio, jossa

lopputulos ei muutu, voidaan poistaa kokonaan. Tällaisia ovat esimerkiksi lausekkeet $x = x + 0$ ja $x = x * 1$. Kummassakin tapauksessa muuttuja x ei muutu, joten ne voidaan poistaa tehokkuuden lisäämiseksi. Tiettyjen aritmeettisten operaattorien konekäskyt ovat halvempia kuin toisten, joten ovisilmäoptimointi voi vaihtaa sen toiseen, joka vastaa samaa operaattoria. Esimerkiksi x^2 voidaan vaihtaa muotoon $x * x$, sillä tällöin ei tarvitse käyttää eksponenttilaskuille tarkoitettuja rutiineja. [2, s.552]

Ovisilmäoptimointi käyttää myös hyödykseen kohdekoneen arkkitehtuuria. Kohdekoneen laitteistolla saattaa pystyä toteuttamaan joitain toimintoja tehokkaammin kuin normaalisti. [2, s.552] Esimerkiksi tietyillä laitteistoilla lisäysoperaatio $x = x + 1$ on mahdollista toteuttaa käskyllä ’INC’ (increment), jolloin operaatio saadaan muotoon INC x, joka on huomattavasti tehokkaampi tapa [2, s.509–510].

5. YHTEENVETO

Työn tavoitteena oli selvittää, miten ohjelmointikielen kääntäjä toimii ja sitä kautta selvittää, miten lähdekoodista päästään konekielelle. Kuvassa 15 on esitetty yhteenveto työssä esitetyistä eri vaiheista, miten kääntäminen konekielelle tapahtuu.



Kuva 15: Kääntäjän toiminta ja rakenne.

Analyysivaihe muodostaa lähdekoodista kuvan 15 mukaisesti välikoodin, joka välitetään synteesivaiheelle. Synteesivaihe rakentaa välikoodista kohdekoodin, joka oli tässä työssä assembly-kielinen koodiketju, joka saadaan muutettua vaivattomasti assemblerin avulla konekieleksi. Symbolitaulukko luodaan käännöksen aikana ja sieltä saadaan tietoa lähdekoodin entiteetteihin liittyen. Sitä käytetään kääntäjän molemmissa vaiheissa.

Aiheesta löytyvän lähdekirjallisuuden perusteella kääntäjän toimintaan kuuluu valtava määrä eri osa-alueita, jotka jätettiin työssä käsittelemättä. Tämän kannalta työllä olisi ollut potentiaalia käsitellä aihetta myös syvällisemminkin. Toisaalta työn tarkoituksena olikin luoda yleiskuva kääntäjän toiminnasta ja vaiheista, joka onnistui tutkimalla tarkemmin vain aiheen oleellisimpia kohtia.

LÄHTEET

[1] Singh Ravendra, Vivek Sharma, and Manish Varsheny, *Design and Implementation of Compiler*, New Delhi: New Age International P Ltd, 2009.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools, 2nd Edition*, Pearson Education, 2006.

[3] Brett Schules, High Level and Low Level Languages, Medium, verkkosivu, Saatavissa (viitattu 18.10.2020): *https://medium.com/@brettschules/high-level-and-low-level-languages-62776d0b89f0*

[4] Michael L. Schmit, *Pentium™ Processor: Optimization Tools*, Elsevier Science & Technology, 1994, pp. 13–18.

[5] Hyde, Randall, *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level*, No Starch Press, 2006.

[6] Mark Halpern, *Machine- and assembly-language programming*, Enclopedia of Computer Science, Jan 2003, pp. 1043–1056.

[7] Ole Agesen & Sriram Sankar, *Compiler*, Encyclopedia of Computer Science, Jan 2003, pp. 251–258.

[8] Dick Grune, Kees van Reeuwijk, Henri E. BalCeriel J.H. Jacobs, Koen Langendoen, *Modern Compiler Design Second Edition*, Springer, 2000.

[9] Allen I. Holub, *Compiler Design in C*, Prentice Hall, Jan 1990.

[10] Symbol Table, javatpoint, verkkosivu, Saatavissa (viitattu 25.11.2020): *https://www.javatpoint.com/symbol-table*

[11] Ben Grunfeld, A brief Overview of Data Structures and Big-O Notation, Medium, verkkosivu, Saatavissa (viitattu 18.12.2020): *https://medium.com/@binyamin/data-structures-and-big-o-notation-ec7ac060f186*

[12] Rob Farber, *CUDA Application Design and Development*, Elsevier Inc., 2012, pp. 1–31.

[13] Colin Robertson & Matthew Sebolt, Tokens and character sets, Microsoft, verkkosivu, Saatavissa (viitattu 8.12.2020): *https://docs.microsoft.com/en-us/cpp/cpp/character-sets?view=msvc-160*

[14] Gabriele Tomassetti, Parsing In C# And Libraries, Saatavissa (viitattu 9.12): https://tomassetti.me/parsing-in-csharp/