hyväksymispäivä arvosana

arvostelija

Historiakatsaus assemblykääntäjistä korkean tason kielten kääntäjiin

Arttu Kilpinen

Helsinki 30.11.2016

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Arttu Kilpinen

Työn nimi — Arbetets titel — Title

Historiakatsaus assemblykääntäjistä korkean tason kielten kääntäjiin

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	30.11.2016	16 sivua

Tiivistelmä — Referat — Abstract

Ensimmäiset ohjelmointikielten kääntäjät, assemblykääntäjät, käänsivät symbolisille konekielille kirjoitettuja ohjelmia konekielisiksi ohjelmiksi. Täsmällisten kuvausjärjestelmien kehitys sekä koodin generoinnin teoria mahdollistivat tehokkaampien ohjelmointikielten kehityksen. Nykyisin käytössä olevat korkean tason ohjelmointikielet kehittyivät hiljalleen kuvausjärjestelmien kehittyessä ja syrjäyttivät symbolisella konekielillä ohjelmoinnin lähes kokonaan. Tässä dokumentissa käydään läpi historiallisia vaiheita symbolisten konekielten kääntäjistä nykyaikaisiin korkean tason ohjelmointikielten kääntäjiin. Läpi käydään useita merkittäviä ohjelmointikieliä ja niiden ominaisuuksia.

ACM Computing Classification System (CCS):
Software and its engineering -> software notations and tools -> Compilers

Avainsanat — Nyckelord — Keywords
Historia, Kääntäjät, Symbolinen konekieli, Ohjelmointikielet

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Kääntäjät ovat tietokoneohjelmia, jotka kääntävät lähdekielisen ohjelmakoodin kohdekieliseksi ohjelmaksi [Bauer, 1974]. Kohdekielenä on usein jonkin prosessoriarkkitehtuurin ymmärtämä konekieli.

Ohjelmointikielet sekä niitä ymmärtävät kääntäjät ja tulkit ovat keskeisessä asemassa ohjelmistotuotannossa. Kääntäjät mahdollistavat ohjelmien kirjoittamisen korkean tason ohjelmointikielillä sekä symbolisilla konekielillä, jotka puolestaan helpottavat ja nopeuttavat ohjelmointia. Niiden käyttäminen tekee ohjelmakoodista myös ymmärrettävämpää ja helpompilukuista. Yleisesti ottaen ohjelmointikielen ymmärrettävyys kasvaa abstraktiotason kasvaessa. Esimerkiksi matemaattisesti tutulla tavalla kirjoitetut aritmeettiset lausekkeet ovat ymmärrettävämpiä kuin vastaava laskenta symbolisella konekielellä ohjelmoituna. Lisäksi useat korkean tason ohjelmointikielet mahdollistavat — mikäli tarvittavat kääntäjät ovat olemassa — saman ohjelmakoodin käyttämisen useissa eri laitteistoissa sekä useilla eri käyttöjärjestelmillä. Koska eri laitteistoissa on erilaiset käskykannat, poistuu korkean tason ohjelmointikieliä käyttämällä myös tarve uudelleenohjelmoinnille.

Symboliset konekielet sekä korkean tason ohjelmointikielet ovat lähes yhtä vanhoja kuin ohjelmointikin. Ensimmäiset korkean tason ohjelmointikielten kääntäjät puolestaan ovat ohjelmointikieliä huomattavasti nuorempia, sillä ohjelmointikielten teoriaa kehitettiin vuosia ennen kuin ensimmäiset kääntäjät valmistuivat. Esimerkiksi ensimmäisenä korkean tason ohjelmointikielenä pidetty Plankalkül kehitettiin jo vuonna 1946, mutta sitä ymmärtävä kääntäjä valmistui vasta vuonna 1972 [Knuth and Pardo, 1976].

Ennen korkeatasoisille lausekielille kehitettyjä kääntäjiä oli pitkään käytössä vain symbolisia konekieliä ymmärtäviä ohjelmia, assemblykääntäjiä. Korkean tason ohjelmointikielten kehityttyä saatiin myös niitä tukevia kääntäjiä valmistettua. Vuonna 1952 valmistunut AUTOCODEn kääntäjä oli yksi ensimmäisiä kaupallisessa ohjelmistotuotannossa käytettyjä korkean tason kielen kääntäjiä [Knuth and Pardo, 1976].

2 Symbolinen konekieli ja assemblykääntäjät

Samoille asioille on tietojenkäsittelytieteessä annettu hieman toisistaan poikkeavia määritelmiä. Erään määritelmän mukaan assemblykääntäjä on kääntäjä, joka kääntää yksi yhteen symbolisella konekielellä kirjoitettuja komentoja konekielisiksi komennoiksi [Salomon, 1993]. Koska jokaisella laitteistolla on oma konekielensä ja tämä konekieli on myös ohjelmointikieli, pätee yleinen kääntäjien määritelmä myös assemblykääntäjiin. Lähdekielenä assemblykääntäjän ymmärtämä symbolinen konekieli tarkoittaa konekieltä, jossa laitteen ymmärtämät binääriset konekäskyt on korvattu ihmisille helpommin muistettavilla sanoilla eli symboleilla.

2.1 Historia ensimmäisistä assemblykääntäjistä

Koska ennen ensimmäisiä assemblykääntäjiä ei ollut mitään ohjelmointia helpottavia työkaluja, tuli ensimmäiset kääntäjät ohjelmoida suoraan konekielellä kuten yksi ensimmäisiä, vuonna 1949 valmistunut EDSAC tietokoneen assemblykääntäjä toteutettiin. Vaikka korkean tason kielten kääntäjät alkoivat kehittyä lähes heti ensimmäisten assemblykääntäjien valmistuttua, pysyi symbolisilla konekielillä ohjelmointi pitkään suosiossa. Alkuun korkean tason kielten automaattista käännöstyötä pidettiin lähinnä teoreettisena tutkimisena ja käytännössä kaikki ohjelmoijat uskoivat ettei automaattisesta koodin generoinnista tule ikinä tarpeeksi tehokasta oikeaan ohjelmointiin [Knuth and Pardo, 1976].

Vaikka nykyaikaiset korkean tason ohjelmointikielien kääntäjät tuottavat hyvin optimoitua koodia, on hyvän ohjelmoijan kirjoittama symbolinen konekieli silti lähes poikkeuksetta parempaa. Tämän takia symbolisia konekieliä käytetään jonkin verran matalan tason ohjelmoinnin lisäksi suurta laskentatehoa vaativien ohjelmien optimointiin. Ennen symbolisten konekielten kehitystä ohjelmointi tapahtui kirjoittamalla laitteistoriippuvaista tietyn prosessorin ymmärtämää binäärikoodia. Siitä huolimatta, että käskykannat olivat nykyiseen verrattuna suhteellisen yksinkertaisia, oli ohjelmointi hidasta ja työlästä. Tietokoneiden kehittyessä ja ohjelmien monimutkaistuessa tarve ohjelmointikielille kasvoi. Symboliset konekielet kehitettiin varhain ja nykyisin lähes kaikki sovellusohjelmat kirjoitetaan korkean tason ohjelmointikielillä.

2.2 Assemblykääntäjien toiminnasta ja toteutuksesta

Assemblykääntäjät ymmärtävät jotakin symbolista konekieltä ja osaavat tuottaa tästä konekielisen suoritettavan ohjelman. Symboliset konekielet ovat matalan tason laiteriippuvaisia ohjelmointikieliä, jotka kääntävät lähdekoodia yksinkertaisin ennalta määrätyin ehdoin kohdekielelle. Suurin osa ohjelmakoodista on siis käännettävissä yksi yhteen laitteiston ymmärtämän konekielen kanssa. Poikkeuksena on kuitenkin ohjelman osoitteina käytettävät tunnukset (label), joiden arvot assemblykääntäjä voi vapaasti päättää. Tunnuksina ovat joko paikat ohjelman koodiosassa tai muuttujina käytetyt muistipaikat. Symbolisen konekielen avainsanat ovat siis symboleja laitteiston ymmärtämälle konekielelle. Konekielellä on mahdollista kirjoittaa suoraan suorittimen rekistereihin. Tämä tekee symbolisilla konekielillä ohjelmoimisesta yhtä laiteläheistä kuin suoraan konekielillä ohjelmointikin. Laiteläheisyys puolestaan tekee ohjelmista laitteistoriippuvaisia, sillä eri suorittimilla voi olla erilaiset käskykannat. Symbolien käyttäminen vähentää huomattavasti kirjoitusvirheiden määrää ja tekee koodista helpomman kirjoittaa ja lukea. Tunnisteiden käyttö puolestaan poistaa tarpeen muistaa muuttujien sekä konekäskyjen osoitteita.

Kuva 1 selventää symbolisten konekielten määrittelemien symbolien sekä ohjelmoijan määrittelemien tunnisteiden eron. Esimerkkikoodi on TTK91 [Ttk, 1991] virtuaaliprosessorille tehty ohjelma, joka tulostaa käyttäjälle luvut 0...5. Keltaisella pohjalla olevat symbolit ovat niin sanottuja tunnisteita, joilla voi olla eri arvo käännöskerrasta ja kääntäjästä riippuen. Kaikki harmaalla pohjalla oleva koodi käännetään siis täysin ennalta määrätysti.

```
        LOAD    R1, = 0
        STORE   R1,   X
LOOP    LOAD    R1, = 5
        COMP    R1,   X
        JGRE          END
        LOAD    R1,   X
        OUT     R1, = CRT
        ADD     R1, = 1
        STORE   R1,   X
        JUMP          LOOP
END     NOP
```

Kuva 1: TTK91 esimerkkikoodi

3 Historiaa korkean tason kielten kääntäjistä

Korkean tason ohjelmointikielellä tarkoiteteen tässä dokumentissa ohjelmointikieltä, jossa lähdekielikieli sekä siitä käännettävä konekieli ovat selkeästi eri abstraktiotasoilla ja kääntäminen edellyttää muutakin, kuin mekaanista sanojen vaihtamista ennalta määrättyjen sääntöjen perusteella. Tämän määritelmän perusteella korkean tason ohjelmointikielillä tarkoitetaan tässä dokumentissa niitä kieliä, jotka eivät ole symbolisia konekieliä.

3.1 Täsmällisten kuvausjärjestelmien kehitys

Tietojenkäsittelytieteilijät ovat jo tietokoneiden alkuajoista lähtien yrittäneet kuvailla ohjelmien suoritusta ja algoritmeja konekieltä abstraktimmalla tasolla. Alan Turingin julkaisussa vuonna 1936 esitettiin määritelmä tietojenkäsittelijöiden hyvin tuntemasta laskentalaitteesta, Turingin koneesta. Laitteen yhteydessä määriteltiin myös matemaattinen esitystapa, jolla sen toimintaa voitiin täsmällisesti kuvailla. Vaikka esitystapa oli vaikea eikä kyseisiä Turingin esittelemää laitetta ollut kuin teoreettisella teoriassa, Turingin esitystapa edusti kehittyneintä formaalia kuvausta, 'kieltä', joka siihen aikaan oli olemassa.

Toisen maailmansodan jälkeen vuonna 1945 saksalainen Konrad Zuse aloitti oman tietokoneohjelmien kuvailuun tarkoitetun kielen Plankalkülin kehittämisen. Zusen sanoin Plankalkülin tarkoitus oli luoda puhtaasti formaali esitystapa mille tahansa laskentaongelmalle [Knuth and Pardo, 1976]. Tässä hän onnistuikin varsin hyvin. Plankalkulissa voidaan määritellä aritmetiikan ja ohjausrakenteiden lisäksi rajaton määrä sisäkkäisiä tietorakenteita ja Zusen työhön viitataankin usein ensimmäisenä korkean tason ohjelmointikielenä. Vaikka kyseessä oli huomattavan edistyksellinen järjestelmä, se ei kuitenkaan vaikuttanut ohjelmointikielten kehitykseen juuri lainkaan. Zusen artikkelit julkaistiin vasta vuonna 1972 muiden, kehittyneempien kielten jo olemassa ollessa. Vaikka Plankalkülille toteutettiinkin kääntäjä, ei sitä juuri koskaan käytetty koska silloin oli jo Plankalkülia huomattavasti kehittyneempiä ohjelmointikieliä.

Samoihin aikoihin Zusen kanssa Yhdysvaltalaiset Herman Goldstine ja John von Neumann koittivat ratkaista samaa ongelmaa. Heidän ratkaisunsa algoritmien ja tietokoneohjelmien kuvaamiseen oli varsin erilainen. Von Neumann ja Goldstine esittivät ratkaisuksi lohkokaaviota (flow diagram), esitystapaa jossa ohjelmat kuvataan nuolien ja laatikoiden avulla.

Vuonna 1946 Marylandissa työskennellyt Haskell B. Curry kehitti ENIAC tietokoneelle aikaansa nähden monimutkaista ohjelmaa. Curryn työ ENIACIN parissa sai hänet ehdottamaan formalismia ohjelmistojen toiminnalle. Hänen formalisminsa perustui uuteen ajatukseen ohjelman suorituksen lohkomaisesta rakenteesta, mitä hän nimitti divisiooniksi. Divisioonien tulisi olla rakennettu niin että niiden laskenta olisi toisistaan riippumatonta. Tämän voisikin rinnastaa esimerkiksi C-kielen paikallisiin tietorakenteisiin ja käännösyksiköihin perustuvaan suoritukseen. Curryn formalismi oli kuitenkin hieman luonnoton sillä suoritusyksiköillä oli useita lopetuskohtia sekä nykykielistä poiketen useita aloituskohtia. Historiallisesti työ oli kuitenkin merkittävä, sillä se sisälsi algoritmeja joilla kuvauksesta pystyttiin tuottamaan konekoodia. Näitä rekursiivisia — vaikkakin toteuttamatta jääneitä — algoritmeja voidaankin pitää ensimmäisinä koodin generointiin tarkoitettuina algoritmeina.

3.2 Kohti ensimmäisiä kääntäjiä

Millekään aiemmin mainituista ohjelmointikielistä ei tähän mennessä oltu toteutettu kääntäjiä. Ne toimivat ohjelmoijien käsitteellisenä apuna auttaen ohjelmien suunnittelussa, mutta jättäen toteutuksen ihmisille. Tästä huolimatta ne kaikki olivat merkittäviä askeleita kohti parempia ohjelmointikieliä sekä niiden kääntäjiä. Ilman täsmällisiä esitystapoja ei koodin generointi ikinä olisi tullut mahdolliseksi.

Ensimmäinen korkean tason ohjelmointikieli, jolle toteutettiin tulkki oli Short Code. Sitä kehitti John W. Mauchly vuonna 1949 ja William F. Schmitt toteutti sille tulkin [Knuth and Pardo, 1976]. Tulkki toimi alkuun BINAC tietokoneella mutta se ohjelmoitiin myöhemmin myös UNIVACille. Yksityiskohtia Short Coden toiminnasta ei ikinä julkaistu, joten sen tarkemmasta toiminnasta ei ole tietoa. Vuonna 1955 julkaistusta ohjelmoijille tarkoitetussa manuaalissa kerrotaan kuitenkin kuinka ohjelmaa voidaan käyttää. Short Code oli siis algebrallinen tulkki, joka osasi suorittaa aritmeettisia laskutoimituksia ilman konekielistä ohjelmointia. Ohjelma luki syötettä ja suoritti vastaavat toiminnot ajetulla laitteistolla.

1950-luvun alussa Heiniz Rutishauser ja Corrado Böhm työskentelivät Zürichin teknillisessä yliopistossa Sveitsissä. Vaikka he työskentelivät samassa paikassa ja saman aiheen parissa, eivät he työskennelleet yhdessä. Rutishauser julkaisi 1952 artikkelin, jossa hän kuvasi hypoteettisen tietokoneen sekä siinä toimivan kääntäjän kehittämälleen ohjelmointikielelle. Julkaisu oli merkittävä, sillä siinä kuvattiin ensimmäistä kertaa menetelmä kääntäjien toteuttamisesta sekä koodin generoinnista.

Rutishauserin kollega Corrado Böhm kehitti myös ohjelmointikieltä sekä tämän kääntäjää. Hänen julkaisunsa oli Rutishauserin julkaisua vieläkin merkittävämpi, sillä hän oli toteuttanut kääntäjän tämän omalla kielellä. Böhmin kieli ei kuitenkaan osannut käsitellä muita kuin positiivisia kokonaislukuja, joten sen käyttöarvo jäi melko pieneksi. Kääntäjien teorian kehityksen kannalta se oli kuitenkin korvaamaton. Böhmin kääntäjä kykeni tarkistamaan koodin syntaksia lineaarisessa ajassa kun Rutishauserin kääntäjä toimi suuruusluokassa n^2. Lisäksi Böhmin kääntäjä hallitsi matemaattisten operaattoreiden sidontajärjestyksen, sekä osasi käsitellä sulkeita aritmeettisissa lausekkeissa. Lisäksi Böhm oli ensimmäinen tietojenkäsittelijä, joka todisti matemaattisesti ohjelmointikielensä voivan laskea minkä tahansa laskettavan funktion.

Vaikka Rutishauser ja Böhm olivat kumpikin valmistaneet omat kääntäjänsä, pidetään ensimmäisenä 'oikeana' kääntäjänä silti Alick E. Glennien 1952 valmistamaa AUTOCODE ohjelmistoa. Aiemmista kääntäjistä poiketen AUTOCODE toteutettiin oikealle laitteistolla ja sen tuottama konekieli oli oikeasti suoritettavissa. AUTOCODEa pystyttiin siis käyttämään oikeiden, käyttökelpoisten ohjelmien tekemiseen [Knuth and Pardo, 1976].

Vuoden 1954 alussa John Backus rupesi kehittämään kokoamansa kehittäjätiimin kanssa automaattisen ohjelmoinnin järjestelmää. Järjestelmän oli tarkoitus olla hyvin kehittynyt, joten suureksi haasteeksi muodostui järjestelmän saaminen tarpeeksi tehokkaaksi. Loppuvuodesta 1954 kehittäjäryhmä julkaisi suunnitelman järjestelmästä 'The IBM Mathematical FORmula TRANstating system' — FORTRAN. Kuten jo aiemmin oli todettu, tehokkaan koodin tuottaminen ei ollut lainkaan helppoa. Ryhmän julkaisu alkoikin painottamalla sitä tosiasiaa, että FORTRAN oli tehokas. Aiemmin ohjelmoijien tuli valita helpon ohjelmoinnin ja hitaan suorituksen tai työlään ohjelmoinnin ja nopean suorituksen väliltä, mutta FORTRANin tarjoaisi parhaat puolet molemmista [IBM, 1954]. FORTRAN 0 dokumentti esittää myös ensimmäisen yrityksen esittää ohjelmointikielen syntaksi täsmällisesti. Tätä voidaan pitää Backuksen myöhemmin esittelemän kielioppimuodon Backus Naur Formin (BNF) edeltäjänä.

Kun FORTRAN kaksi ja puoli vuotta myöhemmin saatiin toteutettua, oli se aikansa tehokkain sekä monipuolisin ohjelmointikieli. FORTRAN tuotti kohtuullisen tehokasta koodia ja kehittäjät sanoivat sen olevan lähes yhtä tehokasta kuin hyvän ohjelmoijan kirjoittama symbolinen konekieli. FORTRANissa oli myös paljon ominaisuuksia, joita ei oltu aiemmin nähty. Se oli esimerkiksi ensimmäinen ohjelmointikieli,

jossa muuttujien nimet voivat olla useamman merkin pituisia [Knuth and Pardo, 1976].

Ensimmäisen julkaisun jälkeen FORTRANissa oli kuitenkin useita ognelmia. Virheitä oli paljon ja eräs FORTRANIN kehittäjistä, Saul Rosen, sanoikin ettei uskonut FORTRANin ikinä tulevan toimimaan [Rosen, 1964]. Vaikeuksista huolimatta FORTRANista tuli hyvin suosittu ja sitä käytetiin enemmän kuin oltiin osattu odottaa.

4 kääntäjien toteutus korkealla abstraktiotasolla

However one could discuss how an existing compiler could propagate an image of itself to another machine. This technique popularly referred to as bootstrapping or cross compiling [Reynolds, 2003].

An interesting problem is finding the simplest subset of the language that enables us to create the next generation of the compiler [Reynolds, 2003].

if a compiler for language L is implemented in L, then it should be able to compile itself.

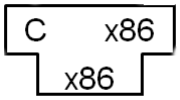
A conventional C compiler, written in C, is said to be bootstrapped if it compiles itself. Now suppose a new version of the compiler source is weirren, that uses different registers for passing arguments. The old compiler can compile this source, yelding a new compiler. But Look! The executable version cc' of the new compiler uses the old parameter passing style, but generates code that uses the new style. onecan use the new compiler however to recompile all the libraries and the new version itself and get a new new executable thet both uses and generates the new parameter passing style [Appel, 1994].

-Cross compilation has become popular viimeistään 76. -Tarkoittaa sorsan kääntämistä masiinalla joka outputtaa toisen masiinan objektikoodia. -intermediate language as a tool to reduce duplivaton of effort. -aika- ja rautarajotuksista johtuen korkean tason kielten käyttö on increasingly popular minicomputers on usein epäkäytännöllistä -tämän takia cross compilation saanut paljon huomiota. -cc on prosessi of one machine accepting a source program as input and producing an object code thet is executable on another machine. -ongelma kääntäjien määrien kanssa, siksi IL. -Saanut alkunsa UNCOLsta -YMS [Speetjens, 1976].

Tekniikkaa voidaan käyttää usealla tavalla, useaan eri ongelmaan [Earley and Sturgis, 1970]. full, incremental ja cross!

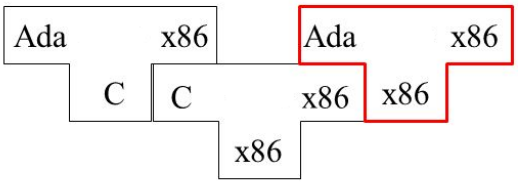
4.1 T-kaaviot

Kääntäjien suunnittelussa ja mallintamisessa on käytetty useita erilaisia kaavioita. Käytetyin ja tunnetuin lienee Harvey Bratmanin 1961 ehdottama kääntäjää kuvaava kaavio [Bratman, 1961], josta käy ilmi kääntäjän ymmärtämä kohde- ja lähdekieli sekä kieli, jolla kääntäjä toimii. Kaaviota kutsutaan Bratman-kaavioksi tai T-kaavioksi. Jälkimmäinen nimi tulee kaavion muodosta, jossa T-kirjaimen muotoisessa alueessa vasen pääty kertoo lähdekielen, oikea pääty kohdekielen ja alaosa kertoo millä kielellä kääntäjä toimii.



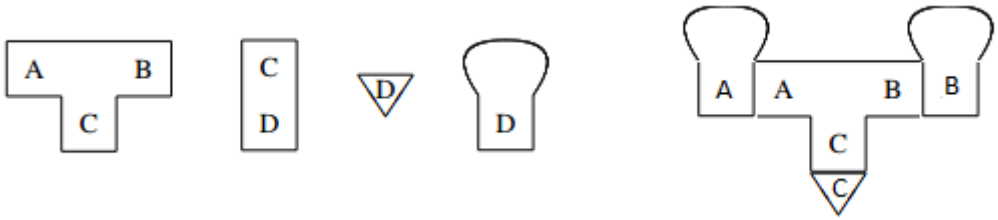
Kuva 2: T-kaavio, joka kuvaa x86 arkkitehtuurilla toimivaa c-kääntäjää, jonka kohdekieli on x86 konekieli

Kaavioita toisiinsa liittämällä voidaan havainnollistaa monimutkaisiakin toimintaketjuja, joita kääntäjät suorittavat. Kuvassa 3 oletetaan että käytössä on c-kielellä kirjoitettu kääntäjä, joka kääntää ada-kieltä x86 konekielelle. Lisäksi käytössä on edellisen kuvan esimerkissä oleva x86 arkkitehtuurilla toimiva c-kääntäjä, jonka kohdekieli on x86 konekieli. Näiden kahden kääntäjän avulla voidaan tuottaa x86 alustalla toimiva ada kääntäjä, jonka kohdekieli on x86. Kahden ensimmäisen kääntäjän yhteistyöllä saadaan siis kolmas kääntäjä. Huomattavaa on, että prosessin alimman tasoinen kääntäjä toimii aina jossakin todellisessa laitteistossa, eikä täten voi olla muu kuin jonkin laitteiston ymmärtämä konekieli.



Kuva 3: c:llä kirjoitetun ada-kääntäjän ja x86:lla toimivan c kääntäjän avulla voidaan tuottaa x86:lla toimiva ada-kääntäjä

T-kaavio oli melko yksinkertainen, joten siitä on kehitetty paranneltuja vaihtoehtoja. Jay Earley ja Howard Sturgis laajensivat Bratmanin kaaviota lisäämällä siihen ominaisuuksia. Alkuperäisen T-kaavion kuvatessa vain kääntäjiä, Earleyn ja Sturgisin kaavioissa pystyi kuvaamaan myös tulkkeja. Lisäksi kaavioon sisällytettiin suoritusta kuvaava osa sekä sovellusohjelmaa kuvaava kaavio. Heidän kaavioissaan määritellään kuvan 4 mukaiset elementit.



Kuva 4: Earleyn ja Sturgisin versio T-kaavioista

Vasemmalta oikealle käytynä ensimmäinen kuva on Bratmanin alkuperäisen T-kaavion vastaava elementti. Se kuvaa kääntäjää joka kääntää kieleltä A kielelle B ja toimii kielellä C. Suorakaiteen muotoinen kahdesta elementistä koostuva kaavio kuvaa tulkkia, joka tulkkaa kieltä C ja toimii kielellä D. Jotta käännös voitaisiin suorittaa, tulee pohjimmaisen kielen olla suoritettavissa jollakin oikealla laitteistolla. Tätä suoritusta kuvaamaan lisättiin yksiosainen kolmiomerkintä. Kolmion sisällä oleva merkintä kertoo mitä konekieltä suoritus ymmärtää, esimerkkikuvassa konekieli on D

[Mogensen, 2010]. Esitettäessä jotakin laskentaongelmaa, joka on ohjelmoitu kielellä D merkitään lampun muotoinen kaavio, jonka sisällä kyseinen kieli ilmoitetaan. Kyseisiä kaavioita voi yhdistellä hyvinkin monimutkaisiksi rakenteiksi. Esimerkkikuvan viimeinen kohta kuvaa yksittäistä käännöstä, jossa A-kielinen ohjelma käännetään B kieliseksi ohjelmaksi. Huomioitava T-kaavioiden käytössä on, että kielten vierekkäisten kaavioiden kielten tulee täsmätä toisiinsa. Kuvan esimerkin käännös ei voi ottaa lähdeohjelmakseen muuta kuin kielellä A kirjoitetun ohjelman.

4.2 bootstrapping

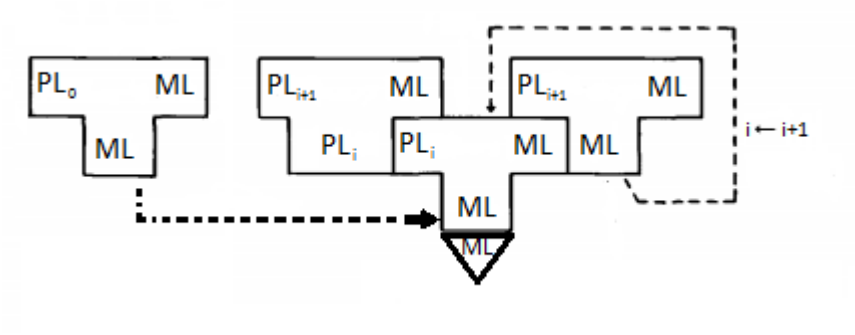
Koska kääntäjien ohjelmointi matalan tason ohjelmointikielillä on erittäin vaivalloista [Mogensen, 2010], suositaan niiden kehittämisessä korkean tason ohjelmointikielten käyttöä. Yksi vaihtoehto on kirjoittaa laitteistolle kääntäjä, jollakin kyseiselle laitteistolle jo olemassa olevalla ohjelmointikielellä. Tämä ei kuitenkaan aina ole mahdollista. Esimerkiksi jos kyseessä on uusi prosessoriarkkitehtuuri, eikä tälle vielä ole kääntäjiä, joudutaan ohjelmointi tekemään jollakin muulla tavalla. Haluttaessa kääntäjän joka sekä kääntää että suorittuu kyseisellä arkkitehtuurilla eräs yleisesti käytetty menetelmä on bootstrapping [Mogensen, 2010].

Bootstrapping on tekniikka jonka ydinajatus on kääntää jokin kääntäjä sillä itsellään [Mogensen, 2010] ja tavanomaisen kääntäjän sanotaan olevan bootsträpätty jos se kääntää itsensä [Appel, 1994].

Perinteinen ongelma kääntäjien kehityksessä on tilanne, jossa kääntäjä pitää rakentaa ilman tai lähes ilman muita työkaluja ja käänäjiä. Iteratiivisessa bootstrapmenetelmässä kääntäjä kehitetään kahdessa päävaiheessa. Ensin kääntäjästä tehdään hyvin suppea versio, joka osaa kääntää tavoitellusta ohjelmointikielestä vain pienen osajoukon [Mogensen, 2010]. Tämä ensimmäisen vaiheen kääntäjä voidaan toteuttaa millä tahansa ohjelmointikielellä, mutta mikäli muiden ohjelmointikielten kääntäjiä ei ole saatavilla se joudutaan kehittämään konekielellä. Toisessa vaiheessa kääntäjä ohjelmoidaan sen itsensä ymmärtämällä kielellä eikä muita ohjelmointikieliä enää tarvita. Tällöin jo olemassa oleva kääntäjä voi kääntää seuraavan version itsestään. Koska kieli on tässä vaiheessa vielä hyvin vajavainen, olisi valmiin ohjelmointikielen toteuttaminen heti ensimmäisen vaiheen jälkeen erittäin työlästä tai mahdotonta. Toista vaihetta suoritetaan useita kertoja siten, että kääntäjän tuntemaa ohjelmointikieltä kasvatetaan ja siitä käännetään uusi kääntäjä jonka avulla kieltä taas kasvatetaan ja niin edelleen.Huomioitavaa prosessissa on se, että kääntäjän uusi versio on kehitettävä aina käyttäen vanhan version tuntemaa ohjelmointi-

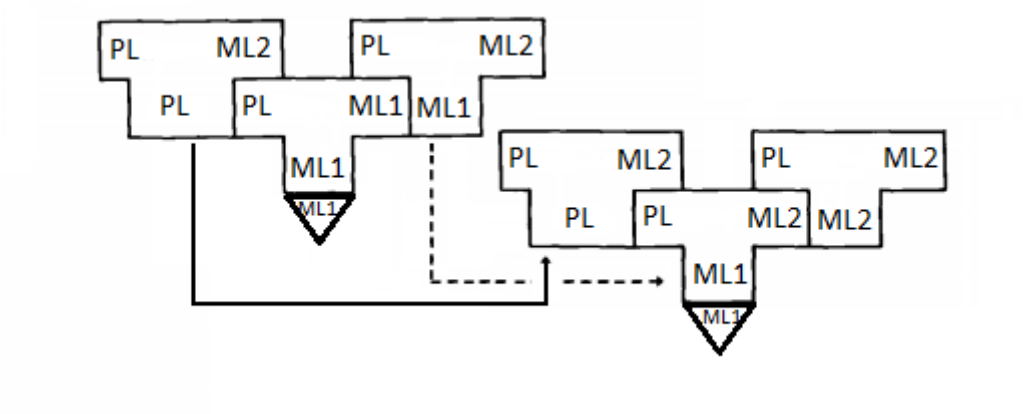
kieltä.

Kuva 5 esittää edellä mainittua tilannetta. Ensimmäisessä vaiheessa kääntäjän ensimmäinen versio ohjelmoidaan konekielellä (ML) ja tämän jälkeen ohjelmointikieltä (PL) laajennetaan iteratiivisesti kunnes tavoiteltu laajuus on saavutettu.



Kuva 5: Iteratiivinen bootsträppäys ilman muita ohjelmointikieliä.

Bootstrap menetelmää käytetään usein myös yhdessä ristiinkäätämisen (cross compiling) kanssa [Reynolds, 2003]. Tällöin tavoitteena on siirtää jo olemassa oleva kääntäjä toimimaan toisessa laitteistossa. Oletetaan että laitteistolle, joka ymmärtää konekieltä ML1 on toteutettu kielen PL kääntäjä. Haluttaessa kielen PL kääntäjä toimimaan sekä tuottamaan uuden laitteiston kohdekoodia ML2, voidaan kielen PL kääntäjä ohjelmoida uudelle laitteistolle ristiinkääntämistä ja bootstrapingiä käyttäen seuraavalla tavalla: Ensin kielellä PL ohjelmoidaan kielen PL kääntäjä uudelle laitteistolle. Tämä voidaan kääntää alkuperäisellä laitteistolla. Nyt kielelle PL on olemassa uudelle laitteistolle koodia tuottava kääntäjä, mutta sen suoritus tapahtuu edelleen vanhassa laitteistossa (ristiin kääntäminen). Nyt Kääntäjän uudella versiolla, jonka kohdekieli on ML2, käännettäessä oma lähdekoodinsa, saadaan kääntäjä, joka sekä toimii että kääntää kielelle ML2. Kuva 6 havainnollistaa tilannetta. Ensimmäisessä vaiheessa jo olemassa olevalla kääntäjällä käännetään uuden kääntäjän lähdekoodi, jolloin tulokseksi saadaan ristiinkääntäjä. Saadulla uudella kääntäjällä käännetään sen oma lähdekoodi, jolloin tuloksena on haluttu uudella laitteistolla toimiva ja sen konekieltä tuottava PL kielen kääntäjä.



Kuva 6: Kääntäjän porttaus uudelle arkkitehtuurille ristiinkääntämisen ja bootstrappingin avulla.

Edellämainittu esimerkki on toimiva, mutta se edellyttää kääntäjän kirjoittamisen kokonaan alusta loppuun asti sen itsensä ymmärtämällä ohjelmointikielellä. Koska korkean tason ohjelmointikielten kääntäjät voivat olla hyvinkin monimutkaisia ja koostua suuresta määrästä lähdekoodia, on tämänkaltainen prosessi varsin työläs. Vastaavan prosessin voi toteuttaa huomattavasti pienemmällä vaivalla, mikäli alkuperäisen kääntäjän rakenne olisi eri tavalla toteutettu.

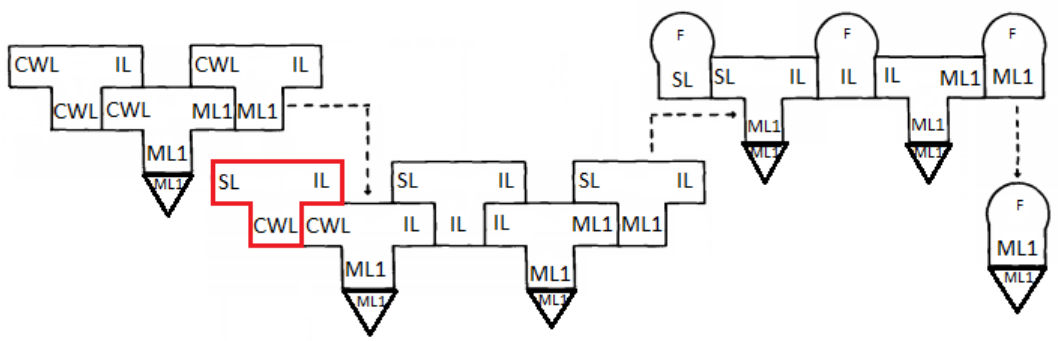
Vielä 1970-luvulla kääntäjät kirjoitettiin hyvin tarkasti tiettyä lähdekieltä ja laitteistoa varten [Guilan et al., 2002]. 1980-luvulla ruvettiin kehittämään enenevässä määrin kääntäjäkokonaisuuksia jotka kykenevät kääntämään useita eri lähdekieliä. Tällaisissa systeemeissä kääntäjän sisäinen rakenne oli jaettu lähdekielestä riippuvaiseen etuosaan (frontend) sekä näiden jakamaan takaosaan (backend) [Guilan et al., 2002]. Tämän kaltainen rakenne vähensi uuden koodin kirjoittamista ja näinollen pienensi kääntäjien kehityksestä koituvia kustannuksia. 80-luvun lopulla oli jo hyväksytty ajatus siitä, että kääntäjien kehitys usealle ohjelmointikielelle ja/tai laitteistolle on tehokkaampaa sekä kilpailukykyisempää.

Eräs tyypillinen tapa kääntäjien ohjelmoinnissa onkin jakaa sen sisäistä rakennetta sekä suorittaa käännös useassa eri vaiheessa. Apuna käytetään usein jotakin niinsanottua välikieltä (intermediata language) [Speetjens, 1976]. Näin haluttu lähdekieli voidaan kääntää helpommin eri laitteistoille.

Oletettaen, että jollekin välikielelle ja jollekin korkean tason ohjelmointikielelle on olemassa tietyllä laitteistolla toimivat kääntäjät, voidaan halutun lähdekielen kään-

nös suorittaa kaksivaiheisesti toteuttamalla korkean tason ohjelmointikielellä kääntäjä lähdekieleltä välikielelle [Earley and Sturgis, 1970]. Uuden lähdekielen toteutus sisältää siis vain korkean tason ohjelmointikielellä tehdyn kääntäjän käytetylle välikielelle, eikä koodin generointivaihetta tarvitse kirjoittaa uudestaan.

Kuva 7 demonstroi usein käytettyä rakennetta [Earley and Sturgis, 1970], jolla kääntäjän jatkokehitystä voidaan helpottaa huomattavasti. Uuden ohjelmointikielen toteuttaminen kyseisen kääntäjän avulla on mahdollista korvaamalla toisen vaiheen ensimmäinen kääntäjä (punaisella) siten, että uusi lähdekieli (SL2) kyetään kääntämään välikielelle (IL) samalla kielellä, jolla kääntäjäkompleksi on suurimmaksi osaksi kirjoitettu (CWL). Esimerkiksi haluttaessa kyseisellä järjestelmällä toteuttaa konekielellä (ML1) suoritettava C-kääntäjä, tulee ohjelmoida vain sellainen osa, joka on kirjoitettu kielellä CWL ja kääntää C-kieltä välikielellä IL. Sama pätee muihinkin toteutettaviin ohjelmointikieliin.



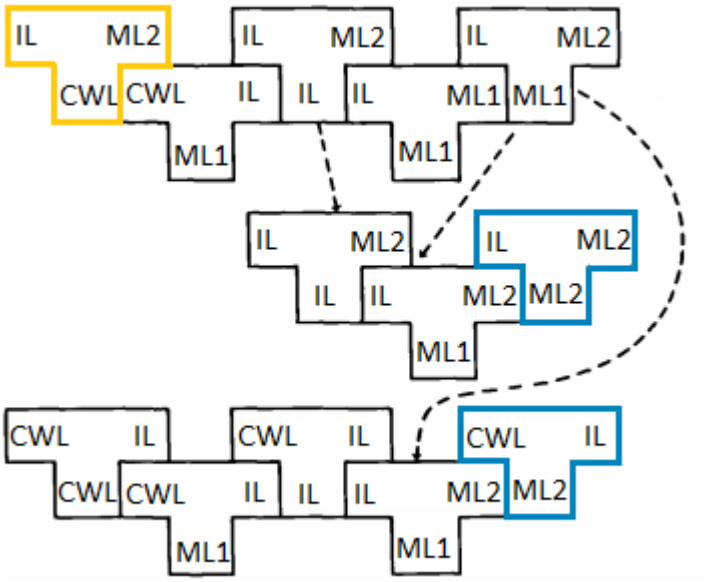
Kuva 7: Käänätäkokonaisuus.

Pelkkää välikieltä apuna käyttäen, uuden ohjelmointikielen toteutus ei vaatisi näin monimutkaista rakennetta. Kyseinen rakenne mahdollistaa kuitenkin sekä tehokkaan ristiinkääntämisen, että itsensä bootsträppäämisen uudelle laitteistolle. Jos kääntäjäkokonaisuuden halutaan kääntävän ymmärtämäänsä lähdekieltä uudelle laitteistolle, jonka konekieli on ML2, joudutaan siihen edelleen kirjoittamaan vain yksi uusi osa. Tämä osa kirjoitetaan kielellä CWL ja se kääntää välikieltä IL uuden laitteiston ymmärtämälle konekielelle ML2. Kun kyseinen osa käännteään jo olemassa

olevalla kääntäjällä, saadaan tulokseksi samassa laitteistossa toimiva kääntäjä välikieleltä uudelle konekielelle ML2. Saadulla ristiinkääntäjällä on siis mahdollista kääntää lähdekieltä konekielelle ML2, mutta sen suoritus tapahtuu edelleen kielen alkuperäistä konekieltä ML1 ymmärtävässä laitteistossa.

Ristiinkääntämisen mahdollistavan uuden osan avulla myös koko kääntäjärakenteen bootsträppääminen uuteen laitteistoon on mahdollista ilman lisäohjelmointia [Earley and Sturgis, 1970]. Kuva 8 havainnollistaa tätä prosessia. Ensimmäisessä vaiheessa jo olemassaolevien komponenttien sekä uuden CWL:llä kirjoitetun välikielikääntäjän (keltainen) avulla tuotetaan seuraavissa vaiheissa tarvittavat komponentit.

Näiden komponenttien avulla voidaan toisessa vaiheessa tuottaa ML2 konekielellä toimiva ja siihen kääntävä välikielikääntäjä. Kolmannessa vaiheessa tuotetaan ensimmäisessä vaiheessa saadun välikielikääntäjän avulla ML2 kielellä toimiva CWL kääntäjä, joka tuottaa välikieltä.



Kuva 8: Käänätäkokonaisuus bootsträp.

Toisessa ja kolmannessa vaiheessa saadut kääntäjät (sininen) sijoittamalla alkuperäiseen ohjelmistoon, saadaan sekä ML2 kielellä toimivat, että sille kääntävät komponentit.

Käsitelty kääntäjäkokonaisuus on monimutkainen mutta myös tehokas. Sekä uuden lähdekielen toteutus alkuperäisellä laitteistolla, että koko kääntäjäohjelmiston siirtämisen uuteen laitteistoon voidaan saavuttaa vain pienen osan uudelleenohjelmoinnilla. Lisäksi uudelleenohjelmointi voidaan suorittaa jo aiemmin käytetyllä korkean tason ohjelmointikielellä CWL:llä [Earley and Sturgis, 1970].

5 Yhteenveto

Tietojenkäsittelytieteessä kääntäjä tarkoittaa ohjelmaa joka kääntää lähdekielisen ohjelmakoodin kohdekieliseksi ohjelmakoodiksi. Koska ennen muun kuin konekielten kehitystä ohjelmointi tapahtui suoraan laitearkkitehtuurin ymmärtämällä muodolla, ei tarvetta kääntäjille ollut. Koska konekoodin ohjelmointi oli varsin työlästä, kehitettiin avuksi symbolisia konekieliä, joissa tietyt binäärijonot oli korvattu paremmin muistettavilla tekstuaalisilla symboleilla.

Korkeamman tason ohjelmointikielet kehittyivät ohjelmoijien tarpeesta kuvata ohjelmistojen toimintaa korkeammilla abstraktiotasoilla. Täsmälliset kuvausjärjestelmät kehitettiin alunperin ilman ajatusta kääntäjistä taikka automaattisesta koodin generoinnista. Vaikka kuvausjärjestelmien sekä koodia generoivien algoritmien kehitys oli alkuun vain teoreettista tutkimista, huomattiin potentiaali niiden tehokkaaseen käyttöön varsin pian. Tämän jälkeen ohjelmointikieliä ruvettiin kehittämään varta vasten automaattisen koodin generoinnin takia ja ensimmäiset oikeasti hyödylliset kielet sekä niiden kääntäjät kehitettiin.

Taulukko 1: Yhteenveto ohjelmointikielten ja kuvausjärjestelmien kehityksestä

Kieli	Kehittäjä	Ensimmäinen
Plankalkül	Zuse	Ohjelmointikieli, Hierarkkinen data
Virtauskaaviot	Goldstine, Von Neumann	Hyväksytty ohjelmointimetodologia
Short Code	Mauchly	Toteutettu korkean tason ohjelmointikieli
Formules	Böhm	Samalla kielellä kirjoitettu kääntäjä
AUTOCODE	Glennie	Käyttökelpoinen kääntäjä
FORTRAN I	Backus	I/O formaatti, kommentit, globaali optimointi

Taulukko 1 tiivistää kappaleessa 2 esitettyjen ohjelmointikielten merkittävimmät piirteet. Lisäksi taulukossa esitetään ohjelmointikielten nimet sekä päätekijät.

Lähteet

[Ttk, 1991] (1991). Ttk91 reference. `https://www.cs.helsinki.fi/group/titokone/ttk91_ref_en.html`. Accessed: 2016-10-21.

[Appel, 1994] Appel, A. (1994). Axiomatic bootstrapping: A guide for compiler hackers.

[Bauer, 1974] Bauer, F. L. (1974). *Compiler Construction – An Advanced Course*. SpringerÂVerslag, Berlin Heidelberg GmbH.

[Bratman, 1961] Bratman, H. (1961). An alternative form of the 'uncol' diagram.

[Earley and Sturgis, 1970] Earley, J. and Sturgis, H. (1970). A formalism for translator interactions.

[Goldstine and Von Neumann, 1947] Goldstine, H. and Von Neumann, J. (1947). Planning and coding problems for an electronic computing instrument.

[Guilan et al., 2002] Guilan, D., Jinlan, T., Suqin, Z., Weidu, J., and Jun, D. (2002). Retargetable cross compilation techniques – comparison and analysis of gcc and zephyr.

[IBM, 1954] IBM (1954). Specifications fot the ibm mathematical formula translating system, fortran Programming Research Group, I.B.M Applied Science Div.

[Knuth and Pardo, 1976] Knuth, D. and Pardo, L. (1976). *The early development of programming languages*. Stanford University.

[Mogensen, 2010] Mogensen, T. (2010). *Basics of Compiler Design*. lulu.

[Reynolds, 2003] Reynolds, J. (2003). Bootstrapping a self compiling compiler from machine x to machine y.

[Rosen, 1964] Rosen, S. (1964). Programming systems and languages, a historical survey.

[Salomon, 1993] Salomon, D. (1993). *Assemblers and Loaders*. Ellis Horwood Ltd.

[Speetjens, 1976] Speetjens, J. (1976). Intermediate language for minicomputer cross-compilation.