hyväksymispäivä arvosana

arvostelija

Otsikko

Arttu Kilpinen

Helsinki 7.12.2016

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

Otsikko

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	7.12.2016	19 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

Johdanto.

2 Symbolinen konekieli ja assemblykääntäjät

Erään määritelmän mukaan assemblykääntäjä on kääntäjä, joka kääntää symbolisella konekielellä kirjoitettuja komentoja konekielisiksi komennoiksi (Salomon, 1993: 1). 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 Ensimmäiset assemblykääntäjät

Koska ennen ensimmäisiä assemblykääntäjiä ei ollut mitään ohjelmointia helpottavia työkaluja, tuli ensimmäiset kääntäjät ohjelmoida suoraan konekielellä. Esimerkiksi yksi ensimmäisistä assemblykääntäjistä, vuonna 1949 valmistunut EDSAC-tietokoneen assemblykääntäjä toteutettiin tällä tavalla. 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: 61).

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 symbolien sekä ohjelmoijan määrittelemien tunnisteiden eron. Esimerkkikoodi on TTK91 virtuaaliprosessorille (Helsingin yliopisto, Tietojenkäsittelytieteen laitos, 1991) tehty ohjelma, joka tulostaa käyttäjälle kokonaisluvut [0, 5]. Keltaisella pohjalla olevat symbolit ovat niin sanottuja tunnisteita, joilla voi olla eri arvo käännöskerrasta ja kääntäjästä riippuen. 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 tarkoitetaan tässä tutkielmassa ohjelmointikieltä, jossa lähdekieli ja siitä käännettävä konekieli ovat selkeästi eri abstraktiotasoilla. Toisin sanoen korkean tason ohjelmointikielellä viitataan kieleen, joka ei ole (symbolinen) konekieli. Korkean tason ohjelmointikielten kääntäminen edellyttää siis muutakin, kuin mekaanista sanojen vaihtamista ennalta määrättyjen sääntöjen perusteella.

3.1 Täsmällisten kuvausjärjestelmien kehitys

Tietojenkäsittelytieteessä on aina koitettu kuvailla ohjelmien suoritusta ja algoritmeja täsmällisesti. Ennen formaaleja merkintätapoja ja korkean tason ohjelmointikieliä ainoa täsmällinen tapa algoritmien kuvaamiseksi oli niiden kirjoittaminen konekielellä. Alan Turingin tunnetussa julkaisussa (Turing, 1937) esitettiin määritelmä laskenta-automaatista, joka paremmin tunnetaan Turingin koneena. Sen yhteydessä määriteltiin myös matemaattinen formalismi, jolla automaatin toimintaa voitiin täsmällisesti kuvailla. Vaikka esitystapa oli vaikea ja esitelty automaatti hypoteettinen, se edusti Alonzo Churchin merkintätavan (Church, 1936) ohella kehittyneintä formaalia kuvausta, 'kieltä', joka siihen aikaan oli olemassa.

Toisen maailmansodan jälkeen vuonna 1945 saksalainen Konrad Zuse aloitti tie-

tokoneohjelmien kuvailuun tarkoitetun kielen Plankalkülin kehittämisen (Knuth and Pardo, 1976: 8). Zusen sanoin Plankalkülin tarkoitus oli luoda puhtaasti formaali esitystapa mille tahansa laskentaongelmalle (Knuth and Pardo, 1976: 10). Plankalkülissa 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, sillä artikkeli julkaistiin vasta vuonna 1972. Vaikka Plankalkülille toteutettiinkin kääntäjä, ei sitä juuri koskaan käytetty, koska silloin oli jo Plankalkülia huomattavasti kehittyneempiä ohjelmointikieliä.

Samoihin aikoihin Zusen kanssa myös Yhdysvaltalaiset Herman Goldstine ja John von Neumann kehittivät laskennallista formalismia. 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 (Knuth and Pardo, 1976: 16). Lohkokaaviot esitellyt artikkeli Goldstine and von Neumann (1947) saavutti suuren lukijakunnan ja sillä oli suuri vaikutus ohjelmointikielten kehitykseen (Knuth and Pardo, 1976: 16) .

Vuonna 1946 Marylandissa työskennellyt Haskell B. Curry kehitti ENIAC-tietokoneelle aikaansa nähden monimutkaista ohjelmaa. Curryn työ ENIAC:n parissa sai hänet ehdottamaan formalismia ohjelmistojen toiminnalle. Hänen formalisminsa perustui uuteen ajatukseen ohjelman suorituksen lohkomaisesta rakenteesta. Näitä itsenäisesti suoritettavia osia hän nimitti divisiooniksi (Curry, 1950: 34). Divisioonat tulisi rakentaa siten, 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 (Knuth and Pardo, 1976: 21). Historiallisesti työ oli kuitenkin merkittävä, sillä se sisälsi algoritmeja joilla kuvauksesta pystyttiin tuottamaan konekieltä. Näitä rekursiivisia — vaikkakin toteuttamatta jääneitä — algoritmeja voidaankin pitää ensimmäisinä koodin generointiin tarkoitettuina algoritmeina.

3.2 Kohti ensimmäisiä kääntäjiä

Millekään aiemmin mainituista ohjelmointikielistä ei niiden julkaisun yhteydessä toteutettu kääntäjiä. Nämä kielet toimivat ohjelmoijien käsitteellisenä apuna auttaen ohjelmien suunnittelussa, mutta jättäen toteutuksen ihmisille. Tästä huolimatta ne olivat merkittäviä askeleita kohti parempia ohjelmointikieliä sekä niiden kääntäjiä. Ilman täsmällisiä esitystapoja ei koodin generointi eli varsinainen käännöksen tai tulkkauksen suorittaminen ikinä olisi voinut tulla mahdolliseksi.

Ensimmäinen korkean tason ohjelmointikieli, joka toteutettiin oli Short Code . Sitä kehitti John W. Mauchly vuonna 1949 ja William F. Schmitt toteutti sille tulkin (Knuth and Pardo, 1976: 23). Tulkki toimi alkuun BINAC-tietokoneella mutta se ohjelmoitiin myöhemmin myös UNIVAC:lle. Yksityiskohtia Short Coden toiminnasta ei ikinä julkaistu, joten sen tarkemmasta toiminnasta ei ole tietoa. Vuonna 1955 julkaistussa 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 tulkkasi vastaavat toiminnot ajetulle laitteistolle.

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 vuonna 1952 artikkelin, jossa hän kuvasi hypoteettisen tietokoneen sekä siinä toimivan kääntäjän kehittämälleen ohjelmointikielelle (Knuth and Pardo, 1976: 30) . Julkaisu oli merkittävä, sillä siinä kuvattiin ensimmäistä kertaa menetelmä kääntäjien toteuttamisesta sekä koodin generoinnista (Ibid).

Myös Corrando Böhm kehitti samaan aikaan itsenäisest ohjelmointikieltä sekä tämän kääntäjää. Hänen julkaisunsa oli Rutishauserin julkaisua vieläkin merkittävämpi, sillä hän oli toteuttanut kääntäjän omalla kielellänä (Knuth and Pardo, 1976: 36). 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 puolestaan toimi neliöllisessä ajassa (Knuth and Pardo, 1976: 40). Lisäksi Böhmin kääntäjä hallitsi matemaattisten operaattoreiden sidontajärjestyksen, sekä osasi käsitellä sulkeita aritmeettisissa lausekkeissa. Böhm oli myös ensimmäinen, joka todisti matemaattisesti ohjelmointikielensä voivan laskea minkä tahansa laskettavan funktion (Knuth and Pardo, 1976: 42).

Vaikka Rutishauser ja Böhm olivat kumpikin valmistaneet omat kääntäjänsä, pidetään ensimmäisenä oikeana kääntäjänä silti Alick E. Glennien 1952 valmistamaa AUTOCODE-ohjelmistoa (Knuth and Pardo, 1976: 42). Aiemmista kääntäjistä poiketen AUTOCODE ei toiminut hypoteettisella laitteistolla joten sen tuottama konekieli oli täten suoritettavissa. AUTOCODEa pystyttiin siis käyttämään oikeiden, käyttökelpoisten ohjelmien tekemiseen (Knuth and Pardo, 1976: 42).

Vuoden 1954 alussa John Backus rupesi kehittämään kokoamansa kehittäjäryhmän 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. Ryhmän julkaisu alkoi painottamalla sitä tosiasiaa, että FORTRAN oli tehokas. Aiemmin ohjelmoijien tuli valita helpon ohjelmoinnin ja hitaan suorituksen tai työläämmän ohjelmoinnin ja nopean suorituksen väliltä, mutta julkaisun jälkeen FORTRAN tarjoaisi parhaat puolet molemmista (IBM Corporation, 1954: 1). FORTRAN 0-dokumentissa on myös ensimmäinen yritys esittää ohjelmointikielen kielioppi 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. Se tuotti suhteellisen tehokasta koodia ja kehittäjät sanoivat sen olevan lähes yhtä tehokasta kuin hyvän ohjelmoijan kirjoittama symbolinen konekieli (Backus et al., 1956: 1). 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: 62).

Ensimmäisen julkaisun jälkeen FORTRANissa oli kuitenkin useita ongelmia. Virheitä oli paljon ja eräs FORTRAN:n kehittäjistä, Saul Rosen, sanoikin ettei uskonut FORTRAN:n ikinä tulevan toimimaan (Rosen, 1964: 4????). Vaikeuksista huolimatta FORTRAN:sta tuli hyvin suosittu ja sitä käytettiin enemmän kuin oltiin osattu odottaa.

Taulukko 1 tiivistää aiemmissa kappaleissa esiteltyjen ohjelmointikielten merkittävimmät piirteet. Lisäksi taulukossa esitellään ohjelmointikielten nimet sekä päätekijät.

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

4 Välikielet ja ristiinkääntäminen

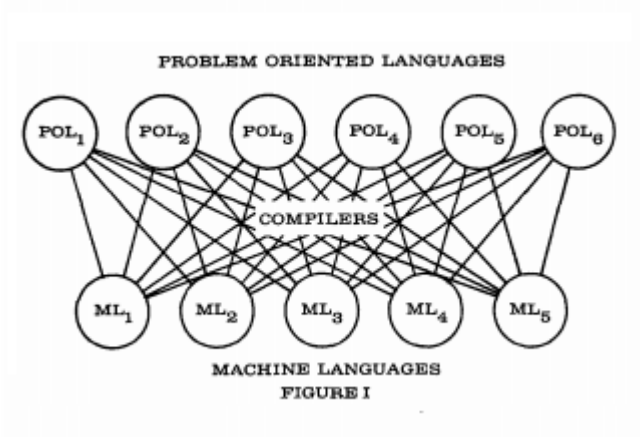
Jo 1950-luvun lopulla sekä käytettävissä olevien ohjelmointikielten että erilaisten laitteistojen määrä oli kasvanut merkittävästi. Korkean tason ohjelmointikielillä ohjelmoinnista oli tullut yleistä ja sen hyödyt olivat laajalti tunnettuja (Strong et al., 1958: 12). Uusien laitteistojen kehitys oli nopeaa ja tyypillisesti laitteistoja uusittiin viimeistään viiden vuoden välein (Ibid). Vaikka korkean tason ohjelmointikielten käyttö vähensi tarvetta sovellusten uudelleenohjelmoinnille, muodostui kääntäjien kehitys merkittäväksi ongelmaksi. Tehokkaan kääntäjän kehittämiseen kuluva aika oli samaa suuruusluokkaa laitteistojen käyttöiän kanssa, joten hyvien kääntäjien valmistuttua oli laitteisto usein jo vanhentunut (Ibid). Ongelmaa koitettiin välttää erilaisilla käännöstekniikoilla. Yleiseksi tavaksi muodostui ristiinkääntäminen uusille laitteistoille.

4.1 Ristiinkääntäminen

Tavallisesti tietokonejärjestelmissä käytettävät kääntäjät tuottavat konekieltä samalle laitteistolle, jossa niitä suoritetaan. Ristiinkääntämisellä (cross compiling) tarkoitetaan sitä, että kohdelaitteisto on jokin muu kuin käännöstä suorittava laitteisto (Free Software Foundation, Inc., 2016). Ristiinkääntämisen mahdollistamiseksi tarvitaan siis kääntäjiä, jotka tuottavat konekieltä uudelle laitteistolle. Nämä ohjelmoidaan vanhalla laitteistolla jo olemassa olevia ohjelmointikieliä käyttäen. Täten saaduilla ristiinkääntäjillä pystytään kääntämään kääntäjiä, tai muita ohjelmia, jotka toimivat uudessa laitteistossa.

Ongelmana edellä mainitussa menetelmässä on suuri sekä kasvava joukko lähde- ja

kohdekieliä. Jo yhden uuden kääntäjän toteuttaminen on varsin työlästä, joten uuden kääntäjän tekeminen kaikille halutuille laitteistoille vaatisi valtavasti aikaa ja resursseja. Oletetaan, että korkean tason ohjelmointikielten määrä on N ja laitteistojen määrä on M. Tällöin tarvittavien kääntäjien määrä on $N \cdot M$ ja se kasvaa neliöllisesti kohde- ja lähdekielien kasvaessa. Kuva 2 esittää tilannetta jossa on kääntäjät kuudelle eri ohjelmointikielelle viiteen eri laitteistoon.



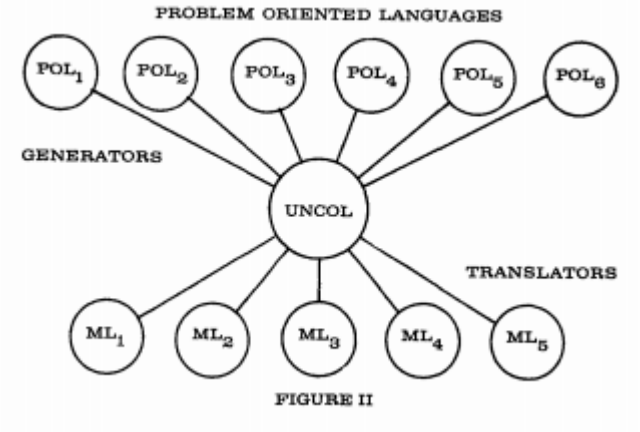
Kuva 2: Ristiinkääntäminen usealta lähdekieleltä usealle kohdekielelle. Lähde: (Steel, Jr., 1961: 378) .

4.2 Välikielet

Mikäli olisi olemassa sellainen ohjelmointikieli, jolle olisi toteutettu kääntäjä kaikille mahdollisille laitteistoille, voitaisiin korkean tason ohjelmointikielet kääntää mille tahansa laitteistolle tätä kieltä apuna käyttäen. Tällaisella menetelmällä tarvittavien kääntäjien määrä vähenisi huomattavasti. Uudet ohjelmointikielet saataisiin toimimaan millä tahansa laitteistolla vain yhden uuden kääntäjän avulla. Uuden ohjelmointikielen kääntäjät kääntäisivät niitä kyseiselle välikielelle, josta se voitaisiin jo olemassa olevien kääntäjien avulla kääntää mille tahansa laitteistolle.

Idea tunnettiin alunperin nimellä UNCOL (UNiversal Compuiter Oriented Language) (Strong et al., 1958: 14). UNCOL ei ollut niinkään suunniteltu ohjelmointikieli, vaan pikemminkin idea välikielestä, jonka avulla kääntäjiä voisi toteuttaa pienemmällä vaivalla. Siitä puhuttiin jo vuonna 1954, eikä konseptin alkuperäistä keksijää tunneta (Ibid).

Mikäli jonkinlainen UNCOL — kaikkien kääntäjien tuntema välikieli — olisi olemassa, vähenisi ristiinkääntämiseen tarvittavien kääntäjien määrä huomattavasti. Jos ohjelmointikielten määrä on N ja laitteistojen määrä M, tarvittaisiin tällaisessa välikieliratkaisussa vain $N + M$ kääntäjää (Speetjens, 1976: 15). Toisin sanoen kääntäjien määrä kasvaisi lineaarisesti ohjelmointikielten ja laitteistojen suhteen. Kuvassa 3 esitetään tilanne välikielen avulla.



Kuva 3: Ristiinkääntäminen välikielillä. Lähde: (Steel, Jr., 1961: 378).

UNCOL:ksi on ehdotettu useita eri välikieliä, mutta yhdestäkään ei ole tullut niin käytettyä, että sitä voitaisiin sanoa universaaliksi. Eräs UNCOL:ksi ehdotettu kieli oli Melvin Conwayn 1958 julkaisema välikieli. Conwayn julkaisussa *Proposal for an UNCOL* määritellään matalan tason välikieli, jota hän kutsuu SML:ksi (Simple Machine Language) (Conway, 1958: 5).

Kaikilla laitteilla toimivan välikielen tulisi olla täysin laitteistoriippumaton. Laitteistoriippumattomalla kielellä tarkoitetaan, että se voidaan kääntää tehokkaasti mille tahansa laitteistolle (Brown, 1972: 1060). Koska erilaisia laitteistoja on hyvin suuri määrä, ei mikään kieli ole täysin laitteistoriippumaton (Ibid). Näin ollen kaikille laitteistoille käännettävän välikielen toteutus on melko utopistista. Useat laitteistot ovat kuitenkin tarpeeksi samankaltaisia tehokkaan välikielen kehittämiseksi, mikäli tavoiteltujen kohdelaitteistojen määrää pienennetään. Suurin osa eniten käytetyistä tietokoneista toimii niin samankaltaisesti, että hyvä välikieli kykenee kattamaan niistä suurimman osan (Ibid).

Vaikka välikielet ovat usein laitteistoläheisiä ja ne käännetään suoraan jollekin kone-

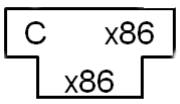
kielelle, voisi välikieli teoriassa olla hyvinkin korkeatasoinen. Jos jollekin laitteistolle on olemassa hyvä jonkin korkean tason ohjelmointikielen kääntäjä, voisi välikieli olla myös korkean tason ohjelmointikieli, jonka kohdekielenä ei olisi mikään konekieli vaan kyseinen toinen korkean tason ohjelmointikieli (Brown, 1972: 1060). Koska tällainen tekniikka tuottaa käytännössä aina huonotasoista koodia, käännetään välikielet lähes poikkeuksetta, vaikka olisivatkin korkean tasoisia ohjelmointikieliä, suoraan konekieliksi (Ibid).

5 T-kaaviot

Kääntäjien suunnittelussa ja mallintamisessa on käytetty useita erilaisia merkintätapoja. Käytetyin ja tunnetuin lienee Harvey Bratmanin 1961 ehdottama T-kaavio. Hän kehitti sen (Strong et al., 1958)-julkaisussa eisteltyn UNCOL-kaavion korvaajaksi (Bratman, 1961).

5.1 Alkuperäinen Bratman-kaavio

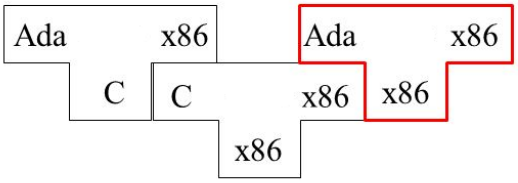
Harvey Bratmanin esittelemä kaavio kuvaa yksittäistä kääntäjää. Siitä käy ilmi kääntäjän ymmärtämä kohde- ja lähdekieli sekä kieli, jolla kääntäjä toimii. Kaaviota kutsutaan Bratman-kaavioksi tai T-kaavioksi. Jälkimmäinen nimi tulee kaavion muodosta, jossa T-kirjaimen muotoisessa alueessa vasempaan päätyyn merkitään lähdekieli, oikeaan päätyyn kohdekieli ja alaosaan kieli jolla kääntäjä toimii.



Kuva 4: T-kaavio, joka kuvaa x86-arkkitehtuurilla toimivaa C-kääntäjää, jonka kohdekieli on x86-konekieli.

Kaavioita toisiinsa liittämällä voidaan havainnollistaa monimutkaisiakin kääntäjillä suoritettavia toimintaketjuja. Kuvassa 5 oletetaan, että käytössä on C-kielinen kääntäjä, joka kääntää Ada-kieltä x86-konekielelle. Lisäksi käytössä on kuvan 4 x86-arkkitehtuurilla toimiva C-kääntäjä, jonka kohdekieli on x86-konekieli. Näiden kah-

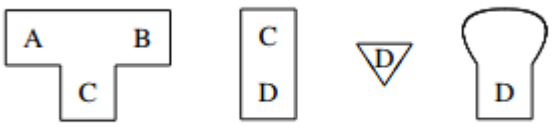
den kääntäjän avulla voidaan tuottaa x86-alustalla suoritettava Ada kääntäjä, jonka kohdekieli on x86. Kahden ensimmäisen kääntäjän yhteistyöllä saadaan siis kolmas kääntäjä. Huomattavaa on, että kaavion alin kääntäjä toimii aina jossakin todellisessa laitteistossa, eikä täten voi olla muu kuin laitteiston ymmärtämä konekieli.



Kuva 5: C-kielisen Ada-kääntäjän ja x86:lla toimivan C kääntäjän avulla voidaan tuottaa x86:lla toimiva ada-kääntäjä

5.2 Earleyn ja Sturginsin merkintätapa

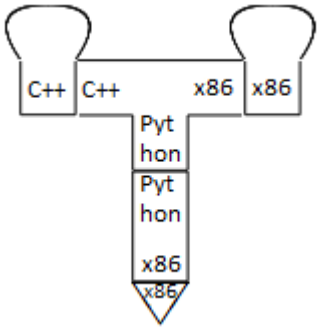
T-kaavio on melko yksinkertainen, joten siitä on kehitetty paranneltuja vaihtoehtoja. Jay Earley ja Howard Sturgis laajensivat Bratmanin kaaviota lisäämällä siihen ominaisuuksia (Earley and Sturgis, 1970: 607-616). Alkuperäisen T-kaavion kuvatessa vain kääntäjiä, Earleyn ja Sturgisin kaavioissa pystyy kuvaamaan myös tulkkeja. Lisäksi kaavioon sisällytettiin suoritusta kuvaava osa sekä sovellusohjelmaa kuvaava kaavio. Heidän kaavioissaan määritellään kuvan 6 mukaiset elementit.



Kuva 6: Earleyn ja Sturgisin T-kaavioelementit. Kuvan lähde (Mogensen, 2010: 282).

Kuvan vasemman puoleisin elementti vastaa Earleyn ja Sturgisin merkintätavassa alkuperäistä T-kaaviota. Se kuvaa C-kielistä kääntäjää joka kääntää kieleltä A kielelle B. Suorakaiteen muotoinen kahdesta osasta 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 kuvaa yksiosainen kolmiomerkintä. Kolmion sisällä lukee millä konekielellä suoritus tapahtuu. Esimerkkikuvassa konekieli on D. Viimeinen kaavio tarkoittaa jotain sovellusohjelmaa tai määrittelemätöntä laskentaa. Kaavion sisällä lukee ohjelmointikieli, jolla sovellus on ohjelmoitu tai jolle se on käännetty. Esimerkkikuvassa kieli on D (Mogensen, 2010: 282).

Kyseisiä kaavioita voi yhdistellä hyvinkin monimutkaisiksi rakenteiksi. Kuvan 7 esimerkissä on kaavio, joka kuvaa C++-kielisen ohjelman kääntämistä x86-konekielelle. Suoritus tapahtuu x86-laitteistolla ja käännöksen tekee Pythonilla toimiva kääntäjä, joka tulkataan x86-laitteistossa. Huomioitava T-kaavioiden käytössä on, että kaavioiden vierekkäisten osien kielten tulee täsmätä toisiinsa. Kuvan esimerkin käännös ei voi ottaa lähdeohjelmakseen muuta kuin C++:llä kirjoitetun ohjelman.



Kuva 7: Esimerkki Earleyn ja Sturgisin kaavioiden yhdistelemisestä.

6 Kääntäjien rakenne ja ketjutus

Koska kääntäjien ohjelmointi matalan tason ohjelmointikielillä on erittäin vaivalloista (Mogensen, 2010: 281), suositaan niiden kehittämisessä korkean tason ohjelmointikielten käyttöä. Yksi vaihtoehto on ohjelmoida laitteistolle kääntäjä jollakin kyseiselle laitteistolle jo olemassa olevalla ohjelmointikielellä. Tämä ei kuitenkaan aina ole mahdollista. Esimerkiksi, jos kyseessä on uusi prosessoriarkkitehtuuri, eikä

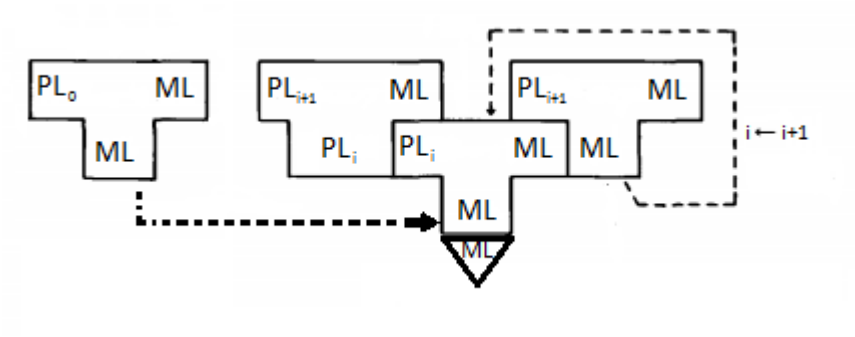
tälle vielä ole kääntäjiä, on ohjelmointi tehtävä jollakin muulla tavalla. Kääntäjä, joka sekä kääntää että on suoritettavissa uudella laitteistolla voidaan toteuttaa ketjutukseksi (bootstrapping) kutsutulla tekniikalla (Mogensen, 2010: 281).

Ketjutus on tekniikka, jonka ydinajatus on kääntää jokin kääntäjä sillä itsellään (Mogensen, 2010: 281). Tavanomaisen kääntäjän sanotaan olevan ketjutettu jos se kääntää itsensä (Appel, 1994).

6.1 Iteratiivinen ketjutus

Perinteinen ongelma kääntäjien kehityksessä on tilanne, jossa kääntäjä pitää ohjelmoida ilman muita työkaluja ja kääntäjiä. Iteratiivisessa ketjutusmenetelmässä kääntäjä kehitetään kahdessa osassa. Ensin kääntäjästä tehdään hyvin suppea versio, joka osaa kääntää tavoitellusta ohjelmointikielestä vain pienen osajoukon (Mogensen, 2010: 287). Tämä ensimmäisen vaiheen kääntäjä voidaan ohjelmoida millä tahansa ohjelmointikielellä, mutta mikäli muiden ohjelmointikielten kääntäjiä ei ole saatavilla, se joudutaan tekemään konekielellä. Toisessa osassa 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. Toista vaihetta suoritetaan useita kertoja siten, että kääntäjän tuntemaa ohjelmointikieltä kasvatetaan ja siitä käännetään uusi kääntäjä. Saadun kääntäjän avulla voidaan seuraava versio toteuttaa käyttäen uusia ominaisuuksia. Seuraavan version avulla tehdään jälleen seuraava versio ja niin edelleen. Huomioitavaa prosessissa on se, että kääntäjän uusi versio on kehitettävä aina käyttäen vanhan version tuntemaa ohjelmointikieltä.

Kuva 8 esittää edellä mainittua tilannetta. Ensimmäisessä vaiheessa kääntäjän ensimmäinen versio ohjelmoidaan konekielellä (ML, "Machine Language") ja tämän jälkeen ohjelmointikieltä (PL, "Programming Language") laajennetaan iteratiivisesti, kunnes tavoiteltu kieli on saavutettu.

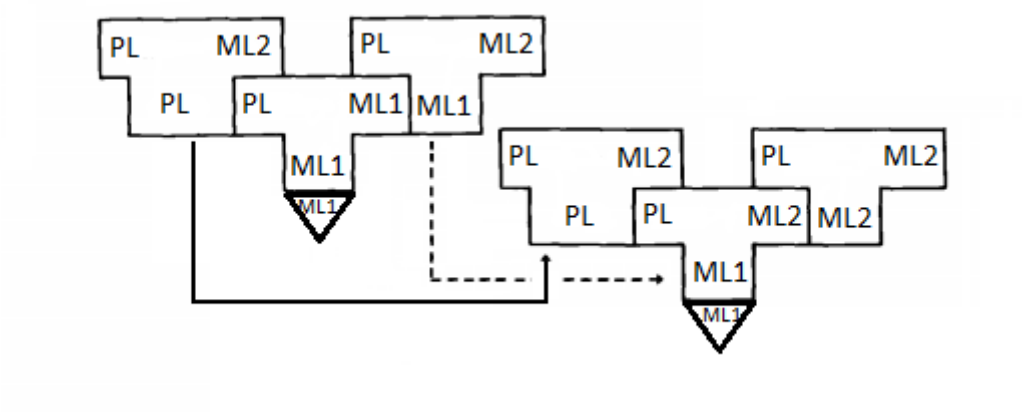


Kuva 8: Iteratiivinen bootsträppäys ilman muita ohjelmointikieliä.

Eräs ongelma edellä kuvatussa prosessissa on kielen kehitykseen soveltuvan sopivan osajoukon määritteleminen (Reynolds, 2003: 176). Mikäli ohjelmointikieltä laajennetaan vain vähän, on päivitys lähes hyödytön. Liian monimutkaisen kielen toteuttaminen yksinkertaisella osajoukolla on puolestaan erittäin työlästä.

6.2 Ketjutus olemassa olevien kääntäjien avulla

Ketjutusmenetelmää käytetään usein myös yhdessä ristiinkääntämisen kanssa (Reynolds, 2003: 175). Tällöin tavoitteena on siirtää jo olemassa oleva kääntäjä toimimaan toisessa laitteistossa. Oletetaan, että laitteistolle joka suorittaa konekieltä ML1 on toteutettu kielen PL kääntäjä. Mikäli kielen PL kääntäjä halutaan toimimaan sekä tuottamaan uuden laitteiston konekieltä ML2, voidaan kielen PL kääntäjä ohjelmoida uudelle laitteistolle ristiinkääntämistä ja ketjuttamista käyttäen seuraavalla tavalla. Ensin kielellä PL ohjelmoidaan sen itsensä 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 (ristiinkääntäminen). Nyt Kääntäjän uudella versiolla, jonka kohdekieli on ML2, käännettäessä oma lähdekoodinsa, saadaan kääntäjä, joka sekä toimii että kääntää kielelle ML2. Kuva 9 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 9: Kääntäjän siirtäminen uudelle laitteistolle ristiinkääntämisen ja ketjutuksen avulla.

Edellämainittu esimerkki on toimiva, mutta se edellyttää kääntäjän kirjoittamisen kokonaan alusta loppuun asti sen omalla lähdekielellä. Koska korkean tason ohjelmointikielten kääntäjät voivat olla todella 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.

6.3 Ketjuttamista helpottava kääntäjärakenne

Vielä 1970-luvulla kääntäjät kirjoitettiin suurimmaksi osaksi yhtä lähdekieltä ja laitteistoa varten (Guilan et al., 2002: 38). 1980-luvulla ruvettiin kehittämään enenevässä määrin kääntäjäkokonaisuuksia, jotka kykenevät kääntämään useita eri lähdekieliä (Ibid). Tällaisissa systeemeissä kääntäjän sisäinen rakenne oli jaettu lähdekielestä riippuvaiseen etuosaan (frontend) sekä näiden jakamaan takaosaan (backend). Tämän kaltainen rakenne vähensi tarvetta kirjoittaa uutta koodia ja näin ollen 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 (Ibid).

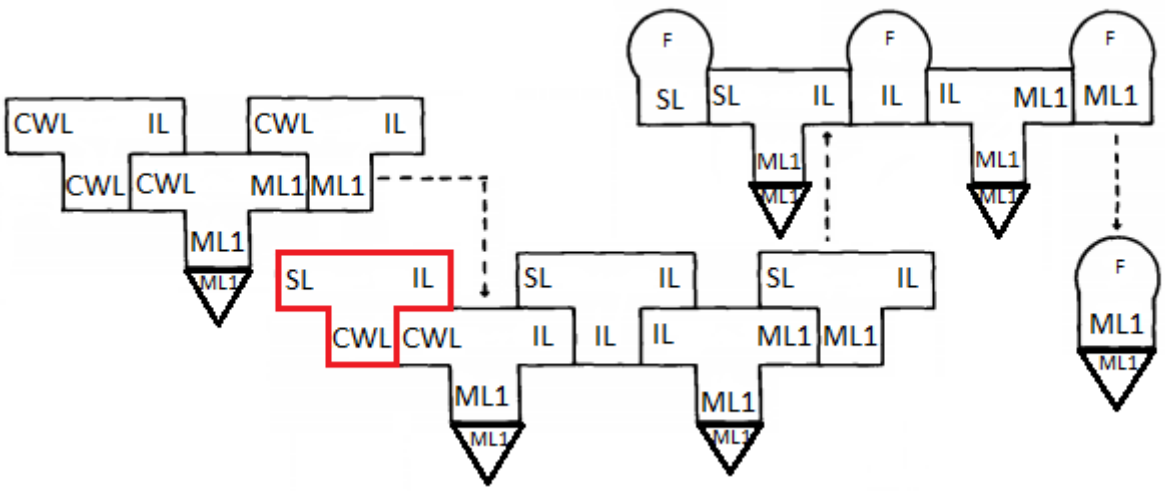
Eräs tyypillinen tapa kääntäjien ohjelmoinnissa onkin jakaa sen sisäistä rakennetta sekä suorittaa käännös useassa eri vaiheessa. Apuna käytetään usein jotakin välikieltä (Speetjens, 1976: 15). Näin haluttu lähdekieli voidaan kääntää helpommin eri

laitteistoille.

Olettaen, että jollekin välikielelle IL ("Intermediate Language") ja jollekin korkean tason ohjelmointikielelle on olemassa tietyllä laitteistolla toimivat kääntäjät, voidaan halutun lähdekielen käännös suorittaa kaksivaiheisesti toteuttamalla korkean tason ohjelmointikielellä kääntäjä lähdekieleltä välikielelle (Earley and Sturgis, 1970: 610). Uuden lähdekielen toteutus sisältää siis vain korkean tason ohjelmointikielellä tehdyn kääntäjän käytetylle välikielelle, eikä koodin generointivaihetta tarvitse kirjoittaa uudestaan.

Kuva 10 demonstroi usein käytettyä rakennetta (Earley and Sturgis, 1970: 610), jolla kääntäjän jatkokehitystä voidaan helpottaa huomattavasti.

Uuden ohjelmointikielen kääntäjän toteuttaminen kyseisen kääntäjän avulla on mahdollista vain yhden komponentin (punainen) uudelleenohjelmoinnilla. Vaihtamalla toisen vaiheen ensimmäinen kääntäjä kääntämään uudelta lähdekieleltä (SL, "Source Language") välikielelle, vaihtuu myös seuraavissa komponenteissa olevat lähdekielet uuteen kieleen. Uusi kääntäjä tulee toteuttaa samalla ohjelmointikielellä, jolla aiempi komponentti on toteutettu (CWL, "Compiler Writing Language"). Esimerkiksi jos kyseisellä järjestelmällä halutaan toteuttaa konekielellä ML1 suoritettava C-kääntäjä, tulee ohjelmoida vain sellainen osa, joka on kirjoitettu kielellä CWL ja kääntää C-kieltä välikielelle IL. Sama pätee muihinkin toteutettaviin ohjelmointikieliin.

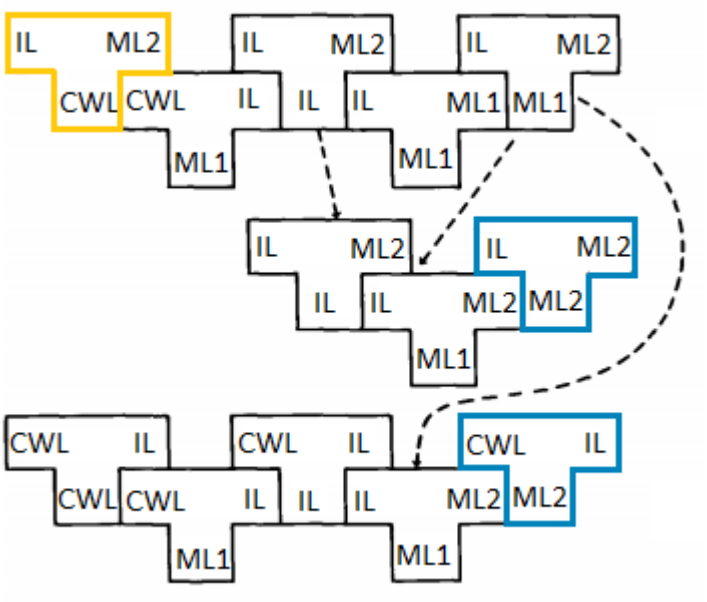


Kuva 10: Kääntäjäkokonaisuus.

Pelkkää välikieltä apuna käyttäen uuden ohjelmointikielen toteutus ei vaatisi näin monimutkaista rakennetta. Kyseinen rakenne mahdollistaa kuitenkin sekä tehokkaan ristiinkääntämisen, että itsensä ketjuttamisen uudelle laitteistolle. Jos kääntäjäkokonaisuuden halutaan kääntävän ymmärtämäänsä lähdekieltä uudelle laitteistolle jonka konekieli on ML2, joudutaan siihen edelleen kirjoittamaan vain yksi uusi komponentti. Tämä osa kirjoitetaan kielellä CWL ja se kääntää välikieltä IL uuden laitteiston ymmärtämälle konekielelle ML2. Kun kyseinen osa käännetään jo olemassa olevalla kääntäjällä, saadaan tulokseksi samassa laitteistossa toimiva kääntäjä välikieleltä uudelle konekielelle. Saadulla ristiinkääntäjällä on siis mahdollista kääntää lähdekieltä SL konekielelle ML2, mutta sen suoritus tapahtuu edelleen kielen alkuperäistä konekieltä ML1 ymmärtävässä laitteistossa.

Ristiinkääntämisen mahdollistavan uuden osan avulla myös koko kääntäjärakenteen ketjuttaminen uuteen laitteistoon on mahdollista ilman lisäohjelmointia (Earley and Sturgis, 1970: 610). Kuva 11 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 11: Käänätäkokonaisuuden ketjutus uudelle laitteistolle.

Sijoittamalla toisessa ja kolmannessa vaiheessa saadut kääntäjät (sininen) alkuperäiseen ohjelmistoon, saadaan sekä ML2-kielellä toimivat, että sille kääntävät komponentit. Ainoa komponentti jota ei saada suoraan sijoittamalla on ensimmäisen vaiheen toinen komponentti. Uudella konekielellä toimivat kääntäjät CWL → IL ja IL → ML2 vastaavat kuitenkin yhdessä tätä komponenttia.

Käsitelty kääntäjäkokonaisuus on monimutkainen, mutta myös tehokas. Sekä uuden lähdekielen toteutus alkuperäisellä laitteistolla että koko kääntäjäohjelmiston siirtäminen uuteen laitteistoon voidaan saavuttaa vain pienen osan uudelleenohjelmoinnilla. Lisäksi uudelleenohjelmointi voidaan suorittaa jo aiemmin käytetyllä korkean tason ohjelmointikielellä (Earley and Sturgis, 1970: 610).

7 Yhteenveto

Yhteenveto.

Lähteet

Appel, A. W. (1994). Axiomatic bootstrapping: A guide for compiler hackers. *ACM Trans. Program. Lang. Syst.*, 16(6):1699–1718.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Herrick, H. L., Hughes, R. A., Mitchell, L. B., Nelson, R. A., Nutt, R., Sayre, D., Sheridan, P. B., Stern, H., and Ziller, I. (1956). *The Fortran automatic coding system for the IBM 704 EDPM*. IBM Corporation.

Bratman, H. (1961). An alternate form of the UNCOL Diagram. *Commun. ACM*, 4(3):142.

Brown, P. J. (1972). Levels of language for portable software. *Commun. ACM*, 15(12):1059–1062.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363.

Conway, M. E. (1958). Proposal for an UNCOL. *Commun. ACM*, 1(10):5–8.

Curry, H. (1950). *A program composition technique as applied to inverse interpolation*. U.S. Naval Ordnance Laboratory.

Earley, J. and Sturgis, H. (1970). A formalism for translator interactions. *Commun. ACM*, 13(10):607–617.

Free Software Foundation, Inc. (2016). Cross-compilation. `https://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html`. Accessed: 2016-12-02.

Goldstine, H. H. and von Neumann, J. (1947). *Planning and coding problems for an electronic computing instrument*. Institute for Advanced Study, Princeton, New Jersey.

Guilan, D., Jinlan, T., Suqin, Z., Weidu, J., and Jun, D. (2002). Retargetable cross compilation techniques: Comparison and analysis of GCC and Zephyr. *SIGPLAN Not.*, 37(6):38–44.

Helsingin yliopisto, Tietojenkäsittelytieteen laitos (1991). Ttk91 reference. `https://www.cs.helsinki.fi/group/titokone/ttk91_ref_en.html`. Accessed: 2016-10-21.

IBM Corporation (1954). *Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*. IBM Corporation.

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

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

Reynolds, J. H. (2003). Bootstrapping a self-compiling compiler from machine X to machine Y. *J. Comput. Sci. Coll.*, 19(2):175–181.

Rosen, S. (1964). Programming systems and languages: A historical survey. In *Proc. AFIPS '64 (Spring)*, pages 1–15. ACM.

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

Speetjens, J. K. (1976). Intermediate language for minicomputer cross-compilation. In *Proc. ACM-SE 14*, pages 15–18. ACM.

Steel, Jr., T. B. (1961). A first version of UNCOL. In *Papers Presented at IRE-AIEE-ACM '61 (Western)*, pages 371–378. ACM.

Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T. (1958). The problem of programming communication with changing machines: A proposed solution. *Commun. ACM*, 1(8):12–18.

Turing, A. M. (1936–1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.