Tampereen yliopisto

Ville Nyrhilä

JÄSENTÄMISEN STRATEGIAT

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Joulukuu 2019

TIIVISTELMÄ

Ville Nyrhilä: Jäsentämisen strategiat
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan kandidaattiohjelma
Joulukuu 2019

Ohjelmistokääntäjät lukeutuvat kaiken nykyaikaisen ohjelmoinnin kivijalkoihin. Osa ohjelmointikielistä toimii siten, että ne käännetään annetulta koodikieleltä konekieleksi. Toisaalta jotkut ohjelmointikielet toimivat siten, että lähdekoodia lukee reaaliaikaisesti ohjelmistotulkki, joka itse on konekieleksi käännetty ohjelmisto. Lisäksi on vielä ohjelmointikieliä, joita käännetään jonkinlaiseksi välikieleksi, jota sitten tulkataan konekieleksi. Hyvin harva kuitenkaan on täysin perillä siitä, miten kääntäjät toimivat. Tässä työssä tutkitaan yleisimpiä menetelmiä eräästä tietystä kääntämisen vaiheesta: jäsentämisestä. On haluttu tietää, miten kääntäjät pilkkovat koodia, miten käsittelevät sitä ja millaiseen muotoon se jäsennellään.

Työssä kuvaillaan ensin yksityiskohtaisesti jäsentimien toiminnan periaatteita, teoriaa ja historiaa, minkä jälkeen perehdytään kaikkein eniten käytetyn jäsentimen toimintaan. Aihetta tutkiessa saatiin selville, että jäsentimet jakaantuvat karkeasti kahteen kategoriaan: ylhäältä-alas ja alhaalta-ylös -tyyppeihin. Kumpikin näistä jakautuu edelleen alatyyppeihin, jotka ovat toinen toistaan tehokkaampia. Ne ovat myös entistä monimutkaisempia ja soveltuvat yhä laajemman ohjelmointikielten kirjon kääntämiseen.

Havaittiin, että jäsentimien suunnittelussa on vankka tiede, joka on muodostunut purkamaan korkeamman tason ohjelmoinnin abstraktioita ja vastaamaan asiaan liittyviin teknisiin haasteisiin. Kaikkein kehittyneimmissä jäsentimissä käytetään matemaattisloogisia merkintätapoja kuvailemaan jäsentimien toimintaa.

Avainsanat: kääntäjä, jäsennin

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

Φ	Joukko-opissa tyhjää joukkoa kuvaava merkki.
∈	Tyhjää merkkijonoa merkitsevä symboli [1].
DFA	Deterministic, finite automata; deterministinen, äärellinen automaatti [8]. Äärellinen automaatti on laite, jolla on erilaisia tiloja ja määrättyjä siirtymiä näiden tilojen välillä. Deterministisessä automaatissa kustakin tilasta siirrytään aina samalla tietyllä siirtymällä tiettyyn toiseen tilaan. [1]
GCC	GNU Compiler Collection, GNU C Compiler; GNU-ryhmän avoimen lähdekoodin kääntäjäkokoelma, myöskin lyhenne sen sisältämälle C-ohjelmointikielen kääntäjälle. [4]
GNU	GNU (lyh. Gnu's Not Unix) on ilmainen ja täysin avoimen lähdekoodin käyttöjärjestelmä. Sitä kehittää GNU-ryhmä. [7]
LALR(1)	Nykyaikana laajimmin käytössä oleva jäsennintyyppi. [1][2]
LL(1)	Asiayhteydettömien kielten alajoukko, jolle ominaisia piirteitä ovat, että syötettä luetaan vasemmalta (ensimmäinen L), niiden jäsentämiseen käytetään vasemmanpuolista derivointia (toinen L) ja että niissä tarkastellaan syötettä ennakkoon vain yhden merkin verran. [1][2]
LR	Tietynlainen shift-reduce -tekniikkaan perustuva jäsennintyyppi [1][2]. Tarkemmin selitetty sivulla 9.
NFA	Nondeterministic, finite automata, nondeterministinen äärellinen automaatti. Kts. DFA. Nondeterministisellä automaatilla siirrytään tilanteen mukaan kulloisestakin tilasta eri siirtymillä eri tiloihin. [1]
S	Yleinen merkki kieliopin ensimmäiselle merkille. [1]
w	Merkki, joka kuvaa syötteeksi annettua merkkijonoa. [1]

1. JOHDANTO

Koneiden toiminnan ymmärtäminen on vaikeaa. Ne ymmärtävät ainoastaan konekieltä, joka puolestaan on ihmisille vaikeaa. Ilman perusteellista syventymistä kulloisenkin laitteen dokumentaatioon, olisi mahdotonta saada mitään aikaiseksi. Aivan varhaisimmat käskykannat ovat kuitenkin olleet sen verran suppeita ja yksinkertaisia, ettei niitä välttämättä ollut vaikea oppia. Ohjelmoinnin alkuaikoina konekielen opettelu oli toistuva haaste, joka oli selvitettävä kerta toisensa jälkeen, koska lähes jokaisella prosessorilla on oma, erityinen konekielensä. Aikanaan ohjelmointi oli erittäin erikoistunut taitolaji, johon vain insinöörit kykenivät.

Nämä ongelmat onnistuttiin ratkaisemaan keksimällä ohjelmistokääntäjät ja -tulkit. Kääntäjä on ohjelma, joka kääntää jotain korkean tason ohjelmointikieltä joko konekieleksi tai jonkin asteiseksi välikieleksi [3]. Tulkit puolestaan kääntävät lähdekoodia suorittamisen aikana [3]. Näiden avulla voitiin tehdä siirrettävää koodia, eli sama ohjelma saatiin toimimaan usealla, erilaisella laitteella. Koodi kirjoitetaan tekstitiedostoon, jota kutsutaan lähdekoodiksi. Se annetaan kääntäjälle tai tulkille syötteeksi. Tässä tekstissä tullaan puhumaan lähes yksinomaan kääntäjiä koskevista asioista.

Lähdekoodi on kuvaus jostain algoritmista tai ohjelmasta jollain ongelmaorientoituneella ohjelmointikielellä. Sen vaste eli objektikoodi on sitä vastaava kuvaus koneorientoituneella kielellä. Algoritmi on joukko selkeitä sääntöjä tai ohjeita, joilla ratkaistaan jokin ongelma siten, että käytetään jokin rajallinen määrä askelia. [3]

Kääntäjien suunnittelu on taitona ja taiteenlajina yhtä vanha kuin ohjelmistokääntäjät itse. Tähän päivään mennessä tällä lajilla on monen vuosikymmenen pituiset perinteet. Suunnittelu on oleellinen tekijä kääntäjien soveltamisessa oikeaan ohjelmointityöhön, koska kääntäjät eivät välttämättä tuota kaikkein optimaalisinta koodia. Tämä onkin pitkään ollut yksi estetekijä kääntäjien soveltamiseen joillain alustoilla. Käsintehty konekielinen koodi on mahdollista kirjoittaa niin tehokkaaksi kuin ohjelmoijan asiantuntemus vain sallii.

Nykyisin kääntäjät ovat hyvin kehittyneitä, eivätkä niiden toiminnan yksityiskohdat ole ohjelmoijille selviöitä. Kysymys ei sinällään ole oleellinen; ohjelmoijalle riittää, että kääntäjä tekee mitä pitää. Asiaan perehtyminen lisää ohjelmoijan kokonaisvaltaista hallintaa aiheesta ja voi mahdollisesti opettaa kirjoittamaan koodia, joka kääntyy tehokkaaksi konekoodiksi. Näin varsinkin, jos perehtyy eniten käyttämäänsä kääntäjään.

Koska kääntäjän toiminta on moniosainen ja monimutkainen, on rajattava tutkittava osuus kapeaksi. Käytetyt tutkimuskysymykset ovat tässä dokumentissa seuraavat:

1. *Minkälaisiksi alkioiksi lähdekoodi jaetaan?*
2. *Minkälaiseen rakenteeseen nämä alkiot laitetaan?*
3. *Millä tavoin kääntäjät tarkistavat koodin oikeellisuuden?*
4. *Mistä nämä tarkistustavat ovat tulleet?*
5. *Mikä on yleisin jäsennintyyppi?*
6. *Mitä loogisia rakenteita siihen kuuluu?*

Luvussa 2 selitetään ensin kääntäjien koko toiminta kiteyttäen ja vastataan kysymyksiin 1 ja 2. Luvussa 3 paneudutaan yksityiskohtaisemmin tutkimuskysymyksiin 4 ja 5. Luvussa 4 paneudutaan kysymyksiin 6 ja 7.

2. YLEISESTI KÄÄNTÄJISTÄ

Kääntäjien toiminta on monivaiheinen prosessi, jonka ensimmäisissä vaiheissa pilkotaan sisään menevä koodi ja tarkistetaan sen oikeellisuus. [1][2] Kääntäjät pilkkovat lähdekoodia enimmäkseen samankaltaisilla tavoilla tiettyjen, yleisesti käytettyjen kaavojen mukaisesti. Tällä tarkoitetaan, että tapoja on useampi kuin yksi ja ne eroavat toisistaan oleellisin tavoin.

Ensimmäinen vaihe on *leksikaalinen analyysi*. Tämä prosessi ei ota kantaa asiayhteyteen, ts. se on kontekstivapaa eli asiayhteydetön. Siinä lähdekoodi rikotaan lekseemeihin, joiden oikeellisuus arvioidaan. *Lekseemi* on mikä tahansa merkkijono tai yksittäinen merkki, jolla on tai voi olla merkityksellinen tulkinta annetussa ohjelmointikielessä. Useimmiten välilyönnillä jaetut merkkijonot katsotaan omiksi lekseemeikseen, mutta ei kaikissa tapauksissa. Poikkeuksiin lukeutuvat muun muassa moniosaiset avainsanat, joita joissain kielissä tavataan. [1][2] Yksittäiset merkit, kuten puolipiste C++:ssa, katsotaan omiksi lekseemeikseen ja ne huomataan, vaikka niitä ei olisi välilyönneillä rajattu. Lekseemit kerätään jonoon siinä järjestyksessä kuin ne ilmestyvät lähdekoodissa ja annetaan edelleen analyysin seuraaviin vaiheisiin. Tämä vastaa kolmeen ensimmäiseen tutkimuskysymykseen: *”Minkälaisiksi alkioiksi lähdekoodi jaetaan?”* ja *”Minkälaiseen rakenteeseen nämä alkiot laitetaan?”*.

Toinen vaihe tyypillisessä käännösprosessissa on *syntaktinen analyysi*. Myös tässä vaiheessa lekseemejä tarkastellaan kontekstittomasti. Syntaktinen analyysi tapahtuu *jäsentelyksi* kutsutun toiminnon kautta. Siinä lekseemit asetellaan derivointipuurakenteeseen. *Derivointipuu* on tietorakenne, joka jäsentelytavan mukaan lähtee yhdestä alkiosta, jota sanotaan juureksi. Derivointipuu haarautuu kahteen tai hieman useampaan suuntaan, mutta puun rakentaminen voidaan aloittaa myös sen ulommaisilta oksilta. Derivointipuuta ei viedä seuraaviin vaiheisiin, vaan niihin annetaan lekseemijono sellaisenaan kuin se leksikaalisesta vaiheesta tuli ulos. [1][2] Luvussa 3 kuvaillaan syntaktista analyysia yksityiskohtaisesti. Siinä vastataan tutkimuskysymyksiin 3 ja 4: *”Millä tavoin kääntäjät tarkistavat koodin oikeellisuuden? Mistä nämä tarkistustavat ovat tulleet?”*

Kolmantena vaiheena on *semanttinen analyysi* ja tämä on ensimmäinen kontekstitietoinen vaihe, eli sisällöllä ja asiayhteydellä on väliä. Tämä tarkoittaa, että tarkastellaan muun muassa, tuleeko tietotyyppejä tallennettua vääränlaisiin muuttujiin. [1][2] Tämä vaihe omalla tavallaan johtaa yhteen käännettävien kielten etuun tulkattaviin kieliin nähden: on koko joukko virheitä, jotka voidaan havaita jo tässä vaiheessa.

Tämän jälkeen jotkin kääntäjät tuottavat eräänlaista välikoodia. Esimerkiksi Java ja C# toimivat tähän tyyliin siten että ne käännetään omalle välikielelleen, esimerkiksi Javan tapauksessa Java-tavukoodiksi. Kaikki kääntäjät eivät toteuta tätä vaihetta.

Riippumatta siitä, toteutettiinko välikoodin kirjoitus tai ei, seuraava vaihe olisi optimointi [1][2]. Tämäkin vaihe on siinä mielessä ehdollinen, että kaikki kääntäjät eivät sitä tee. Esimerkiksi GCC-kääntäjissä tämä on valinnainen operaatio. GCC on GNU-ryhmän avoimen lähdekoodin kääntäjäkokoelma, sisältäen GNU:n alkuperäisen GCC:n, joka oli C-kielen kääntäjä [4]. GNU-ryhmä on maailmanlaajuinen joukko, jonka päämääränä on tuottaa ilmainen täysin avoimen lähdekoodin käyttöjärjestelmä nimeltään GNU (lyh. Gnu’s Not Unix) [7].

GCC-kokoelman kääntäjillä optimoinnin järeys on säädettävissä [5]. Isoissa projekteissa, kuten kokonaisissa Linux-pohjaisissa käyttöjärjestelmissä, hyvin järeää optimointioperaatiota ei suositella. Kääntäjä oikaisee tässä vaiheessa niin monta mutkaa, että ohjelma ei välttämättä enää tule tehneeksi niitä asioita, joita sen oli tarkoitus. [6]

3. JÄSENTIMIEN ESITTELY

Jäsennin on ohjelma tai algoritmi, joka suorittaa syntaktista analyysia. Se saa syötteekseen jonon tokeneita, jotka on eroteltu lähdekoodista leksikaalisen analyysin aikana. [3] Tokenit ovat symboleja, jotka kuvaavat ohjelmointikielessä esiintyviä ilmaisuja tai muuttujia [1][2]. Leksikaalinen analyysi on syntaktista analyysia välittömästi edeltänyt vaihe. Jäsentimet asettelevat tokenit puurakenteeseen, toisin sanoen ne *jäsentelevät* tokenit.

Havainnollistetaan lähdekoodin pilkkomista ja tokenien keräämistä C-kielen Hello world -ohjelmalla:

```
int main(int argc, char *argv[]) {
      printf(”Hello world\n”);
      return 0;
}
```

Koodi jakautuu tokeneihin, joista merkit (,), [,], { ja } ovat *rajaimia* (engl. *delimiter*); *main, argc* ja *argv* ovat *tunnistimia* (engl. *identifier*) ja *”int”*, *”printf”* sekä *”return”* ovat avainsanoja. *Merkkijonot*, kuten *”Hello world\n”*, ovat oma token-tyyppinsä. Muunlaiset litteraalit, kuten 0, ovat vakioita. Nämä esimerkit perustuvat C-kieleen, mutta eri kielet voivat jaotella tokenit muihin tyyppeihin.

Jäsennin pilkkoo kunkin lauseen token-palasiinsa ja riisuu niistä kaiken asiayhteyden, toisin sanoen ”int” ei ole enää ”int”, vaan ”avainsana”; 0 ei ole enää ”nolla” vaan ”vakio”. Samanlainen riisunta tehdään kaikille tokeneille. Sitten jäsennin vertaa asiayhteydettömäksi muunnettua syötelausetta ohjelmointikielen kielioppiin. Esimerkiksi, onko *”avainsana vakio erityismerkki”* eli *”return 0;”* kelvollinen lause. Jos johdos kielioppisääntöjen ja syötelauseen välillä voidaan johtaa, on syötelause syntaktisesti kelvollinen.

Jäsentimet vertaavat syötelausetta kielen kielioppiin ja tarkistavat, onko lause oikeellinen. Jos koodi on virheellinen, annetaan virheilmoitus. Kääntäjä jatkaa tarkastelua syötteen loppuun asti, mutta ei aloita varsinaista kääntämistä tarkastusprosessin päätteeksi. Kääntämiseen ryhdytään vain, jos missään analyysin vaiheessa ei havaita virheitä.

Jäsentelyssä tuotettua derivointipuuta ei välitetä semanttiseen analyysiin, sillä derivointipuu on asiayhteydetön rakenne. Semanttinen analyysi taas tarkastelee tokeneita juuri siinä asiayhteydessä, jossa ne ovat lähdekoodissa esiintyneet. Tässä työssä ei tarkastella semanttista analyysia tai tapoja toteuttaa sitä.

4. KAKSI TAPAA

Jäsentimille on käytettävissä kaksi karkeasti erilaista jäsentämistapaa: alhaalta ylös ja ylhäältä alas. Molemmat käyttävät syötteenään leksikaalisesta analyysistä saatavaa tokenien jonoa. [15][17]

Luvut 4.1 ja 4.2 tulevat vastaamaan kysymyksiin: *”Millä tavoin kääntäjät tarkistavat koodin oikeellisuuden? Mistä nämä tarkistustavat ovat tulleet?”* Noissa luvuissa kuvaillut kummatkin jäsentelystrategiat yrittävät vetää johdoksen kielen sallittujen lauseiden ja tarkasteltavaksi annetun lauseen välillä.

Yhteistä jäsentelystrategioille on, että ne asettavat tokeneita derivointipuuhun. Tämä on pääpiirteittäin kuin mikä tahansa puurakenne. Puurakenne koostuu solmuista ja lehdistä, jotka ovat edellisen solmun lapsisolmuja. Solmut sisältävät tietoa, derivointipuun tapauksessa tokeneita [15][17]. Solmun käyttämistä algoritmissa kutsutaan *solmussa vierailuksi* [2]. Solmusta sen alipuiden kautta kulkemista kutsutaan *solmussa kulkemiseksi* [2]. *Puussa kulkemiseksi* kutsutaan juurisolmussa kulkemista ja koko siitä laskevassa rakenteessa liikkumista [2]. Tämä vastaa tutkimuskysymykseen: *”Minkälaiseen rakenteeseen alkiot laitetaan?”*.

Kielioppi on oleellinen käsite jäsentämisen ymmärtämiselle. Se käsittää joukon sääntöjä, joita kutsutaan produktioiksi ja joita käytetään jäsentämisessä. Kielioppiin kuuluu myös sanasto, joka jaetaan terminaalisymboleihin, eli *terminaali*, sekä nonterminaalisiin symboleihin, eli *nonterminaali*. Ensin mainittu käsittää kielen omat jakamattomat merkit ja komennot. Niiden merkitys on jo määrätty. Ne ovat lopullisia eli terminaaleja. Nonterminaalit puolestaan kuvaavat muuttujia ja ne korvataan jäsentelyprosessin myötä erinäisillä terminaaleilla tai niiden joukoilla. [1][2]

Kieliopin yhteydessä käytetään yleisesti aloitusmerkin symbolina *S*:ää. Sitä kutsutaan myös lausesymboliksi. Kielen lauseet ovat johdettavissa siitä kieliopin sääntöjen mukaan. Matemaattinen merkki $\in$ tarkoittaa tyhjää merkkijonoa. Merkki *w* tarkoittaa syötteeksi annettua merkkijonoa, ts. kyseessä on yksittäinen lause ohjelmointikielessä. [1][2]

4.1 Ylhäältä alaspäin -jäsentely

Ylhäältä alas -jäsentely saa nimensä siitä, että se on täysin rinnastettavissa jäsentelypuun rakentamiseen juuresta alkaen. Siinä johdoksen vetäminen aloitetaan kielen ensimmäisestä merkistä S. Sitä laajennetaan käyttäen kieliopin produktioita, jotta saadaan kokoon lause, joka vastaa haettua lausetta w. Rekursiivisen jäsentimen tapauksessa ilmaisuja myös poistetaan, kun tarvitsee takautua epäkelvosta jäsentelypuusta.

4.1.1 Rekursiivinen jäsennin

Rekursio on sitä, kun funktio, algoritmi tai muu ratkaisu määritellään viittaamalla siihen itseensä. Esimerkiksi funktio, joka ajonsa aikana kutsuu uuden instanssin itsestään. [3] Rekursiivinen jäsentelymenetelmä aloittaa jäsentelyn puun juuresta S ja laajentaa puuta kokeilemalla yhteensopivia produktioita kieliopista. Sen tavoite on muodostaa merkkijono, joka vastaa syötteen merkkijonoa. Se laittaa solmun lehdiksi ne esineet, jotka ovat sen produktion tuotteita, jonka vasemmalla puolella on juuri syötteessä vastaan tullut terminaali. Sen jälkeen se siirtyy laajentamaan vasenta lehteä kokeillen jotain produktiota, jonka vasemmalla puolella on sama nonterminaali kuin kyseisessä solmussa. [1]

Jos solmussa on terminaali, joka vastaa syötettä samassa kohdassa, ei kyseistä solmua tarvitse työstää enempää, vaan jäsennin siirtyy etsimään vastaavaa syötteen seuraavalle merkille. Jos solmun esine ei vastaa syötettä, on tämä solmu todettava virheelliseksi ja kokeillaan sen sisarsolmua. Jos tämäkään ei toimi, peruutetaan ylempään solmuun ja siitä edelleen sen sisareen. Jos takautuessa palataan juureen, on jäsentäminen todettava epäonnistuneeksi ja annetaan virheilmoitus. [1]

Rekursiivinen ylhäältä-alaspäin -jäsennin on yksi vanhimmista malleista. Sitä kehitti H. D. Huskey 1950-luvulla, jolloin se tosin oli ainut kaltaisensa. Se oli myös jäsentimien vähemmistössä ylhäältä-alaspäin -jäsentimenä. Se tuli kuitenkin nauttimaan suosiota kehittäjien keskuudessa yksinkertaisen ja suoraviivaisen toimintaperiaatteensa takia. Tällainen jäsennin on helppo suunnitella ja ohjelmoida. [13]

Koska jäsentimen toteutus perustuu rekursioon, voi tällaisen jäsentimen todennäköisesti kirjoittaa suppealla määrällä koodia, ainakin muihin jäsennintyyppeihin verrattuna. Takautumisen vuoksi se on kuitenkin paljon aikaa tuhlaava menetelmä, ja todennäköisesti myös muistisyöppö. Tällaisen ohjelmoiminen on helppoa, mikäli ei koe rekursiota vaikeaksi konseptiksi. Grune et al. [2] huomauttavat, että se ei ole kovin hyvä palautumaan virheistä. Se on hyvin yksinkertainen jäsennin toteutettavaksi, mutta hyödyllisyydeltään rajallinen. Mikäli hallitsee jonkin muun jäsennintyypin toimintaperiaatteen, lienee parempi käyttää sitä.

4.1.2 Ennakoiva ylhäältä-alaspäin jäsennin

Rekursiivisten jäsentimien toiminnasta näkyy, että samalle syötteen nonterminaalille valitaan johdonmukaisesti sama terminaali [2]. Takautuminen aiheuttaa paljon ajanhukkaa ja turhaa työtä. Siksi on mielekästä suunnitella jäsennin, joka suoraan valitsee kyseisen terminaalin. Tätä tyyppiä kutsutaan ennakoivaksi *ylhäältä-alaspäin -jäsentimeksi*. Ennakoivan ylhäältä-alaspäin -tyypin keksi ja kehitti A. G. Oettinger 1950-luvulla samaan aikaan, kun Huskey kehitti rekursiivista jäsennintä [13].

Ennakoiva algoritmi toteutetaan laatimalla taulu, johon on ennakoitu kutakin nonterminaalia tyypillisesti seuraava terminaali. Jäsentelytaulun rakentamiseksi pitää tuntea tiettyjä funktioita ja algoritmeja. Algoritmi taulun muodostamiseen kuvaillaan myöhemmin. Ensin on tunnettava FIRST-algoritmi, joka on sen komponentti. Algoritmi on seuraavanlainen [1][2]:

$$FIRST(\alpha) = joukko\ niistä\ terminaaleista,$$
$$joilla\ \alpha{:}sta\ johdettavat\ merkkijonot\ alkavat.$$

Toisin sanoen FIRST(α) on α:n jokaisen produktion ensimmäisten terminaalien joukko. Jos α esimerkiksi käsittää merkit XYZ, algoritmi käsittelee sen ensimmäisen merkin X. Jos X on terminaali, se lisätään tauluun. Jos X on nonterminaali, se voidaan silti lisätä tauluun, mikäli se ei sisällä tyhjiä merkkijonoja. Jos X on tyhjä merkkijono, valitaan tauluun sellainen X:n merkki, joka ei ole tyhjä merkkijono tai ei ole Y:n tai Z:n merkkijono.

Itse jäsentelytaulu muodostetaan kieliopista seuraavalla tavalla [1]:

$$Jokaista\ produktiota\ A \rightarrow \alpha\ kohti,$$
$$jokaista\ terminaalia\ a\ joukossa\ FIRST(\alpha)\ kohti,$$
$$TABLE[A, a] = A \rightarrow \alpha$$

Algoritmi on suoraviivainen, mutta se vaatii monta kierrosta. Jokaista kielen produktiota kohti, käydään läpi tuon produktion tuotteen FIRST-joukko. Jokaisella FIRST-joukon terminaalilla, lisätään kyseinen produktio tauluun. Rivin määrää produktion vasen puoli ja pylvään määrää oikea puoli.

Kakde [1] sanoo, että rajatuissa tapauksissa tauluun voidaan lisätä tyhjän merkkijonon tuottava produktio, mikäli syötteen seuraava merkki on terminaali, joka löytyy tarkastelun alla olevan nonterminaalin FOLLOW-joukosta. Ne ovat terminaaleja, jotka esiintyvät produktioiden toiseksi ensimmäisinä tuotteina.

Jäsennin käyttää taulua siten, että se katsoo syötteen nonterminaalin perusteella, mitä riviä tulee katsoa taulusta. Sitten se katsoo syötteen seuraavaa merkkiä ja valitsee sen perusteella pylvään. Näin se valitsee, millä produktiolla se laajentaa jäsentelypuuta. Koko prosessin päätteeksi saadaan joko täysin syötettä vastaava merkkijono tai virheilmoitus. [1]

Jos jollain nonterminaalilla on kaksi tai useampi vaihtoehtoinen produktio, on välttämätöntä, että niiden FIRST-joukot eivät sisällä ainoatakaan samaa merkkiä. Jos kieliopissa ei ole lainkaan tyhjiä merkkijonoja, on tämä ainoa vaatimus. Jos tyhjiä merkkijonoja on, vaatimuksia on lisää. Vain yksi kahdesta samaa nonterminaalia kuvaavasta produktiosta voi tuottaa tyhjän merkkijonon. Minkään produktion FIRST-joukolla ei saa olla samoja merkkejä kuin sen FOLLOW-joukolla. [1]

Vain tietynlaiset kieliopit ovat yhteensopivia. Kuten aiemmin vihjattiin, tyhjät merkkijonot aiheuttavat ongelmia eikä niitä saisi olla. Niitä voi toki olla, mutta siinä tapauksessa niitä ei lisätä tauluun. Kielioppeja, jotka toimivat näin, kutsutaan LL(1)-kieliksi. Kun ne jäsentävät syötemerkkijonoa, ne katsovat sitä yhden merkin verran ennakkoon eivätkä tuota jäsennyskonflikteja. [1] Kirjallisuudessa usein puhutaan juuri LL(1)-tyypistä, kun puhutaan ennakoivasta ylhäältä-alaspäin -jäsentimestä.

4.1.3 LL(1)-tyyppi

Lyhenne LL(1) jakautuu palasiin. Ensimmäinen L tarkoittaa syötteen lukemista vasemmalta oikealle. Toinen L tarkoittaa vasemmanpuolimmaisinta derivointia, ts. valitaan aina ensin vasemmanpuolimmaisin saatavilla oleva produktio. Merkki (1) tarkoittaa, että seuraavaa syötteen symbolia käytetään seuraavassa jäsentelyprosessissa. Syötettä katsotaan siis vain yhden askeleen päähän. [1]

LL(1)-tyyppi näyttäisi olevan tätä nykyä vanha ja hyvin pitkään tunnettu malli, sillä jo vuonna 1974 Griffiths kuvailee LL(1)-tyypin käyttötekniikoita ja toteaa sellaisia käytetyn moniin kieliin jo vuodesta 1966. Hän mainitsee J. M. Fosterin keksineen ko. tekniikat 1968 ja että näiden teorian kehitti D.E. Knuth vuonna 1971. [12] Bauer sanoo, että Foster keksi LL(1)-menetelmän jo vuonna 1965 ja että niiden julkaiseminen vain viivästyi vuoteen 1968 [13]. Tämä näyttäisi tukevan ajatusta, että ylhäältä-alaspäin-jäsentimet keksittiin ja kehitettiin hyvin varhain.

Bauer kuitenkin osoittaa tekstissään totuuden olleen enimmäkseen päinvastainen, kun hän toteaa ylhäältä-alaspäin -jäsentimien olleen vähemmistöä varhaisten jäsentimien joukossa. 1960-luvun alkuun mennessä lähes kaikki luodut kääntäjät olivat alhaalta-ylöspäin -tyyppisiä, vaikkakin kyseistä termiä ei silloin ollut vielä keksitty [13].

Alatyyppejä on olemassa monta, ja niiden toteutukselliset erot ovat hienovaraisia, mutta oleellisia. Tyypillinen toteutus on ns. *vahva LL(1)-jäsennin*, mutta on myös tyyppi nimeltä *täysi LL(1)*. Jälkimmäinen on periaatteessa vahvempi, mutta käytännössä ei ole olemassa vahvoja LL(1)-jäsentimiä, jotka eivät olisi myös täysiä LL(1)-jäsentimiä. On olemassa kuitenkin hienovarainen ero: täysi LL(1)-jäsennin voi palautua virheistä paremmin, koska se kerää enemmän tietoa. Tätä ei juuri kuitenkaan käytetä jäsentimissä. [2]

Jäsenninten kehitys ei ole vieläkään loppunut ja edelleen keksitään uusia innovaatioita. Boštjan Slivnik kuvaili vuonna 2016 menetelmän, joka yhdistää LL(1)-tyypin myöhemmin tässä tekstissä kuvailtavan LR-tyyppin kanssa. Jäsennin toteutetaan muutoin tyypillisenä LL-jäsentimenä, mutta LL-konfliktien esiintyessä ne ratkaistaan käyttäen jäsentimeen upotettua LR-jäsennintä. [10] LR-jäsentimistä puhutaan enemmän osiossa 3.3.4. Silläkin on omat vahvuutensa, mutta se ei välttämättä ole yleisesti parempi kuin

jotkin muut variantit. Vuotta myöhemmässä tekstissään Slivnik esitteleekin tavan tehdä yleistetty toteutus LLLR-jäsentimestään [11].

Kun kieliopitkin ovat niin isoja kokonaisuuksia, herää kysymys: *”Miten kielioppien pätevyyttä voi testata?”* Tähän ongelmaan Paracha ja Franek [14] kuvailivat vuonna 2009 menetelmän, jossa he käyttivät Purdomin algoritmia testatakseen LL(1)-kieliopin kieltä. Purdomin algoritmi luo automaattisesti ohjelman, jossa jokaista kieliopin terminaalia, nonterminaalia ja produktiota käytetään vähintään kerran. Aluksi kirjoitetaan tulevaan vasteeseen ainoastaan aloitusmerkki S. Jokaisella siitä seuraavalla kierroksellaan Purdomin algoritmi vaihtaa yhden nonterminaalin tulevasta vasteesta sellaisen produktion oikeanpuoleisiin merkkeihin, missä kyseinen nonterminaali oli vasemmalla puolella. Algoritmi päättyy, kun kaikki merkit ovat nonterminaaleja. Vaikka Paracha ja Franek toteuttivat tämän LL(1)-tyypin kieliopille, Purdomin algoritmissa ei ole mitään sellaista, etteikö sitä voisi soveltaa minkä tahansa muunkin kielioppityypin testaamiseen.

4.2 Alhaalta ylöspäin -jäsentely

Alhaalta ylöspäin on monin tavoin edellisen käänteinen strategia. Siinä missä aiempi johtaa aloitusmerkistä S kieliopin produktioiden kautta syötettä vastaavan lauseen, alhaalta-ylöspäin-jäsennin löytää polun S:ään yrittämällä vähentää syötelauseen S:ksi. Aiempi toimii laventamalla, tämä toimii supistamalla.

4.2.1 Käsitteet

S*ententiaalimuoto* on nimitys mille tahansa derivaation symboleiden merkkijonolle. Lauseeksi sanotaan sellaista sententiaalimuotoa, jossa on pelkästään terminaaleja. *Oikeasententiaalimuodoksi* kutsutaan sellaista sententiaalimuotoa, joka syntyy jonkin lauseen oikeanpuoleisesta derivoinnista. [1][12][16]

Kahva on produktio, jonka perusteella alimerkkijono voidaan palauttaa johonkin abstraktiin muotoon, joka on lähempänä juurta. Tätä kutsutaan *redusoimiseksi* [2]. Oleellisin ero erityyppisten alhaalta ylöspäin -tyylisten jäsentimien välillä onkin siinä, miten ne etsivät kahvan [2]. Kahvaksi sanotaan toisaalta itse produktiota, mutta myös kyseisen produktion tuotetta [1]. Kun syötemerkkijonossa tunnistetaan tuotekahva, se redusoidaan sille kuuluvan produktion vasemmanpuoleiseksi symboliksi.

4.2.2 Shift-reduce-tekniikka

Shift-reduce tarkoittaa ”siirtoa ja vähennystä”, mikä on täysin kuvaava nimi. Tällaiseen jäsentimeen kuuluu kaksi komponenttia: pino ja syötepuskuri. Syötepuskuri pitää syötemerkkijonoa ja jäsennin siirtää siitä yksitellen merkkejä pinoon. Jokaisen siirron jälkeen jäsennin tarkastelee pinoa ja yrittää tunnistaa siitä kahvaa. Kun kahva tunnistetaan, se redusoidaan ja pinon sisältö työnnetään takaisin syötepuskuriin. Tätä

toistetaan, kunnes syötemerkkijono on täysin palautettu kieliopin alkumerkkiin S tai kunnes todetaan virhe. Jos syötepuskuri tulee työnnetyksi kokonaisuudessaan pinoon ilman että syötettä saadaan redusoitua S:ksi, jäsennys todetaan epäonnistuneeksi, syöte virheelliseksi ja annetaan virheilmoitus. [1][2] Shift-reduce on käytännössä vain toinen nimi alhaalta-ylöspäin -jäsentimelle [3].

Tämän tekniikan periaate on vanha ja se oli käytössä jäsentimien enemmistössä 1960-luvun alkuun mennessä. Varhaisia alhaalta-ylöspäin -kääntäjiä kehittivät mm. P.C. Fischer, M. P. Schützenberger, M. E. Conway ja D. Gries. Ainoastaan A. G. Oettinger ja H. D. Huskey kehittivät kumpikin ylhäältä-alaspäin -jäsentimiä, joista Oettingerin oma oli jopa ennakoivaa tyyppiä. [13]

Grune et al. [2] mainitsevat seuraavia alatyyppejä. *Precedence parsing,* joka on heikko ja todella yksinkertainen toteutus. Se on edelleen käytössä yksinkertaisissa jäsentimissä, joissa sillä etsitään tyypillisiä aritmeettisiä ilmauksia. *BC(k, m)* eli *bounded context* tarkoittaa rajattua asiayhteyttä, jossa kontekstia rajaa vasemmalta *k* määrä tokeneita ja oikealta *m* määrä tokeneita. Se on kohtuullisen tehokas jäsennin ja oli muodissa 1970-luvulla.

4.2.3 LR-jäsennin

LR on yleisin jäsennintyyppi ja Shift-Reduce-jäsentimien alajoukko [1][2]. Sen nimi tulee ilmaisusta *"Left-to-right Rightmost derivation sequence"* [3]. Tämä jäsennintyyppi työntää tokeneita syötepuskurista pinoon alkaen syötemerkkijonon vasemmasta päästä, ts. sen alusta [1][2]. Sen toteuttaminen on monimutkainen työ ja vaatii jäsennystoimintotaulun ja GOTO-taulun. Niitä varten on määriteltävä DFA, eli deterministinen, äärellinen automaatti. Edelleen DFA:ta varten on muodostettava laajennettu kielioppi. Nämä jäsentimet ovat valitettavan työläitä toteuttaa, jos sellainen tehdään tavanomaisille kieliopeille. On kuitenkin olemassa useita automaattisia LR-jäsenningeneraattoreita [1].

Jäsenningeneraattori on ohjelma, joka ottaa syötteeksi kuvauksen ohjelmointikielen syntaksista ja luo sille ohjelmointikielelle jäsentimelle [3]. On myös kääntäjägeneraattoreita, jotka luovat kokonaisen kääntäjän. Ne ottavat syötteekseen kuvauksen sekä kielen syntaksista että sen semantiikasta [3]. Jäsenningeneraattori on tällaisen ohjelman komponentti. Tunnettu esimerkki kääntäjägeneraattorista on UNIX:n yacc [3] ja jäsenningeneraattorista GNU Bison, joka on yacc:in kanssa yhteensopiva [19].

Tehokkuutensa ja laajan käytettävyytensä puolesta LR-jäsentely on jäsenninten parhaimmistoa. Sillä voidaan jäsennellä kutakuinkin kaikki ohjelmointikielirakenteet, joille voidaan kirjoittaa asiayhteysriippumaton kielioppi. LL(1)-jäsentimillä voidaan käsitellä vain osa niistä kielistä, jotka LR-tyyppisillä voidaan käsitellä. Tutkiessaan syötettä vasemmalta oikealle, LR pystyy löytämään kielioppivirheitä nopeasti. [1]

LR(0) on teorian kannalta merkittävä, mutta liian heikko käytännön sovellutuksiin. *SLR(1)*, eli Simple LR [3], on edellisestä paranneltu versio, mutta edelleen liian heikko. *LR(1)* on kuin LR(0), mutta hyvin tehokas vaikkakin muistisyöppö. *LALR(1)* on LR(1):stä hieman heikennetty versio. Siitä huolimatta se on erittäin tehokas ja hyvin monikäyttöinen. Siksi se onkin yleisin käytössä oleva jäsennintyyppi. [2]

Kakden [1] sekä Brunen ja kumppaneiden [2] materiaalissa alhaalta-ylöspäin -jäsentimet vaikuttavat teoriansa puolesta monimutkaisilta, mutta niiden toimintaperiaate on todellisuudessa hyvin suoraviivainen. Laitetaan syötemerkki kerrallaan pinoon ja redusoidaan, kun tavataan tunnistettava kielioppiproduktio. Bauerin mukaan varhaiset kehittäjät puhuivat pinon sijaan ”kellarista” tai ”laarista” [13]. LR-jäsennintyyppien määrittelevä ero useimmiten onkin se, minkälaisella mekanismilla ne tunnistavat kahvat pinosta.

Kaikkien etujensa puolesta on ilmiselvää, miksi LR-jäsentimistä on tullut eniten käytettyjä tyyppejä. Niiden isoimpana ongelmana on niiden toteuttamiseen tarvittava ennakkotyö, mutta tätäkin on onnistuttu helpottamaan automatisoimalla, ainakin osittain. Tuntuu oudolta, että muunlaisia jäsennintyyppejä on edes vaivauduttu kehittämään, varsinkin kun LR-tyyppi oli jo varhain hyvin suosittu. Kenties ylhäältä-alaspäin-tyyppi on viehättänyt kehittäjiä juuri riittävästi niiden suoraviivaisen toteutuksen vuoksi. Ehkä sen vuoksi niillä on ollut paljon helpompi tehdä prototyyppejä kuin LR-jäsentimillä.

5. LR-JÄSENTIMEN LUOMINEN

Aiemmin tässä tekstissä mainittiin, että LR-jäsennin on kaikista yleisin tyyppi. Tässä osiossa käydään läpi sellaisen luominen yleisesti ja avataan niitä käsitteitä, jotka kuuluvat LR-jäsentimen luomiseen.

5.1 DFA ja NFA

Deterministinen, äärellinen automaatti (lyh. DFA, deterministic, finite automata) on mekanismi, jolla LR-jäsennin tunnistaa kahvat pinosta. Deterministisellä automaatilla kustakin tilasta siirrytään johdonmukaisesti aina samalla siirtymällä tiettyyn toiseen tilaan. [3] LR-jäsentimen on myös valvottava DFA:n tiloja. Tällaisen jäsentimen pinossa onkin kahdenlaisia symboleita: DFA:n tiloja kuvaavia tilasymboleita ja kielioppisymboleita [1].

Jäsennin aloittaa pinossa DFA:n alkutilasta. Se tutkii seuraavaa syötesymbolia ja pinon päällä olevaa tilasymbolia. Jos DFA:ssa on siirtymä kyseisestä tilasta kyseisen siirtymän kautta johonkin toiseen tilaan, siirretään syötesymboli pinon päälle ja sen päälle sitä toista tilaa vastaava tilasymboli. Jos seuraavaa syötemerkkiä ei vastaa mikään yhteensopiva DFA:n siirtymä, mutta sitä redusoiva kahva löytyy, laitetaan kahvan vasemman puolen nonterminaali pinoon, mutta syötteen merkki pysyy paikallaan. [1]

On myös nondeterministisiä automaatteja. Nondeterministisellä automaatilla (lyh. NFA) voi yhdestä tilasta olla useampi vaihtoehtoinen siirtymä eri tiloihin, ja kulloinkin valittava siirtymä määräytyy asiayhteyden mukaan [1][3]. Sen käyttöä ei erityisemmin kuvaile Kakde [1] eikä Grune et al. [2], mikä antaa ymmärtää, että NFA:n käyttö LR-jäsentimissä on enintään harvinaista.

5.2 DFA:n luominen ja laajennettu kielioppi

DFA:n luomisessa voidaan käyttää esineitä (englanniksi *items*), jotka ovat kyseisen kieliopin produktioiden osia. Tällaista esinettä kutsutaan produktion LR(1)-esineeksi, ja se on produktio, jossa on piste sijoitettuna johonkin kohtaan oikealle puolelleen. [1]

Ensin otetaan käytössä oleva kielioppi. Sitten laajennetaan sitä lisäämällä sen alkuun uusi aloitussymboli S. Jos S on jo kuvailtu, voidaan käyttää jotain muuta symbolia. Tämän jälkeen pilkotaan erillisiksi produktioiksi jokainen produktio, jolla on useampi vaihtoehtoinen tuote. Nyt jokaisen tuotteen pitäisi olla kuvattuna omassa produktiossaan. Tämä ei vielä riitä, vaan jokainen produktio on eriteltävä laittamalla piste tuotteiden kunkin merkin viereen. [1] Esimerkiksi produktio ”X→A+B” tuottaa nyt kahvat ”X→.A+B”, ”X→A.+B”, ”X→A+.B” ja ”X→A+B.”.

Piste kuvaa sitä, kuinka suurta osaa osista katsotaan kulloisessakin jäsentämisen prosessissa. Pisteen vasemmalla puolella olevia merkkejä on jo tarkasteltu. Kun piste on X:n oikealla puolella, on X:stä juontuvat merkkijonot jo nähty. Pisteen oikealla puolella olevia merkkejä on tarkoitus tarkastella seuraavaksi. Tällä tavoin saatuja esineitä on aina yksi enemmän kuin alkuperäisessä produktiossa on osia.

Tällaisen laajennetun kieliopin tarkoitus on tehdä jäsentimelle yksiselitteisen selväksi, milloin merkkijono voidaan hyväksyä. Jäsentäminen loppuu, kun jäsennin suorittaa reduktion kieliopin aloitusmerkkiin. Näin saadun äärellisen automaatin tilat vastaavat laajennetun kieliopin produktioita. [1]

Jokainen DFA:n tila on joukko, joka sisältää yhden tai useamman esineen. Joukkoa, joka käsittää vain käypiä prefiksejä, kutsutaan *”kaanoniseksi kokoelmaksi”* [1][2]. DFA:n muodostaminen LR(1) joukoille vaatii kaanonisen kokoelman löytämistä.

Olkoon C kaanoninen kokoelma LR(1) esineiden joukoille. Syötteeksi tulee laajennettu kielioppi. Vasteena tulee C. [18] Kuvaus algoritmista kaanonisen kokoelman muodostamiseen LR(1)-esineille. Perustuu lähteeseen [18]:

```
CC0 = closure({[S’=•S, eof]})
CC = {CC0}
while (new sets are still being added to CC)
      for each unmarked set CCj in CC
            mark CCj as processed
            for each x following a • in an item in CCj
            temp = goto(CCj, x)
            if temp not in CC
                  then CC = CC U {temp}
            record transition from CCj to temp on x
```

Algoritmissa esiintyvä muuttuja CC0 on joukko, joka sisältää sen produktion tai niiden produktioiden, joiden vasemmalla puolella on DFA:n päätesymboli. ”eof” on merkki, josta algoritmi tietää tulleensa syötteen loppuun. ”•” on paikanpitäjämerkki, ts. eri kierroksilla sen sisältö voi olla erilainen. [18]

Ensin alustetaan CC0, minkä jälkeen järjestelmällisesti lisätään kaanoniseen kokoelmaan CC kaikki sellaiset siirtymät, joissa siirrytään CC:n sisältämästä tilasta sellaiseen tilaan, jota ei vielä ole CC:ssä. Tämä tehdään testaamalla jokaista ehdotettua tilaa temp-muuttujassa. Jos muuttuja on uusi, tallennetaan siirtymä siihen. [18]

Algoritmi käyttää kahta muuta algoritmia osana toimintaansa, *closure()* ja *goto()*. Closure tarkoittaa sulkeumaa. Sulkeuma on avoin joukko, joka sisällyttää reunapisteensä muiden pisteidensä lisäksi [9]. Avoin joukko on joukko pisteitä, jonka kunkin ympärillä on avoin pallo [9]. Avoin pallo on joukko, jonka jokaisen pisteen etäisyys johonkin tiettyyn määrättyyn pisteeseen on pienempi kuin jokin tietty, ilmoitettu arvo [9]. Goto()-algoritmi on seuraavanlainen [18]:

```
goto(s, x)
```

```
      moved ← 0
      for each item i in s
            if the form of I is [α→•xδ, a] then
                  moved ← moved U {[α→βx•δ, a]}
      return closure(moved)
```

Cooperin ja Torczonin [18] selitys omalla suomennoksella: ”Goto-proseduuri ottaa kaksi argumenttia: LR(1)-esineiden joukko s ja kielioppisymboli x. Se käy läpi kaikki s:n alkiot. Kun se löytää sellaisen, jossa • välittömästi edeltää x:ää, se luo uuden alkion siirtämällä •:ää oikealle x:n toiselle puolelle. Tämä uusi alkio esittää tilaa, joka seuraa x:n tunnistamisesta. Goto siirtää nämä uudet alkiot joukkoon moved ja palauttaa closure(moved).”

Closure()-algoritmi on seuraavanlainen [18]:

```
closure(s)
      while (s is still changing)
            for each item [A→β•Cδ, a] in s
                  for each production C→γ in P
                        for each b in FIRST(δa)
                              s ← s U {[C→•γ, b]}
      return s
```

Cooperin ja Torczonin [18] selitys omalla suomennoksella: ”Closure-proseduuri käy läpi kaikki alkiot joukossa s. Jos paikanpitäjämerkki • jossain alkiossa edeltää välittömästi jotain nonterminaalia C, closure():n on lisättävä yksi tai useampi produktio, josta C saadaan. Näillä esineillä on alussa • produktion oikealla puolella. -- Jokaisella kolmoissilmukan kierroksella s:ään joko lisätään esineitä tai sille ei tehdä mitään. Alkioita ei ikinä poisteta s:stä. Kolmoissilmukka näyttää raskaalta, mutta todellisuudessa uloin silmukka käsittelee kunkin alkion vain kerran, sillä uudet alkiot lisätään joukon loppuun. Keskisilmukka käsittelee yhden nonterminaalin vaihtoehtoiset produktiotuotteet. Sisin silmukka käy läpi vain joukon FIRST(δa).”

5.3 Jäsennystoimintotaulu ja GOTO-taulu

Jäsennin tarvitsee taulukon, jolla päätellä, mitä tehdä työnsä eri vaiheissa. Tätä varten luodaan *jäsennystoimintotaulu*. Sen rivit ja pylväät määräytyvät DFA:n tilan ja syötemerkin mukaan. Jäsennystoimintotaulun soluissa on yksi neljästä mahdollisesta toiminnosta: *”siirrä”*, *”redusoi”*, *”hyväksy”* ja *”virhe”*. Jos jäsennystoimintotaulun koordinaateista saadaan toiminto ”hyväksy”, hyväksyy jäsennin merkkijonon. Jos koordinaateista tuleekin ”virhe”, suoritetaan tilanteeseen sopiva virheestäpalautumisrutiini. [1]

Suorittaessaan sellaisen reduktion, jossa syötteen merkkiä ei siirretä pinoon, jäsentimen on tiedettävä, mihin tilaan DFA menee tästä nonterminaalista. Siksi tarvitaan taulu, joka parittaa DFA:n tilat ja nonterminaalit johtaen johonkin tilaan. Tätä nonterminaalien siirtojen taulua kutsutaan *GOTO-tauluksi*. [1][2]

Jäsentämisen alussa pinossa on ainoastaan DFA:n alkutila I_0 ja syötepuskurissa w. Jäsennystaulun rakentaminen edellyttää, että DFA:n on oltava sellainen, joka tunnistaa oikeasententiaalimuotojen oikeelliset prefiksit käyttäen annettua kielioppia ja sitten kartoittaa nämä jäsennystoiminto- ja GOTO-tauluihin. [1]

6. YHTEENVETO

Tässä työssä otettiin selvitettäväksi, miten ohjelmistokääntäjät pilkkovat lähdekoodia, miten ne käsittelevät sitä ja minkälaiseen rakenteeseen lähdekoodi tämän prosessin aikana asetellaan. Havaittiin, että pieni osa näiden asioiden kulusta tapahtuu leksikaalisessa analyysissa ja suuri osa syntaktisessa analyysissa. Opittiin myös paljon varsinkin syntaktisen analyysin toteutustavoista.

Lähdekoodin pilkkominen on karkeasti suoraviivainen prosessi. *Minkälaisiksi alkioiksi lähdekoodi jaetaan? Minkälaiseen rakenteeseen nämä alkiot laitetaan? Miten alkiot asetetaan siihen rakenteeseen?* Leksikaalinen analyysi tunnistaa lähdekoodista yksittäiset lekseemit, jotka annetaan jonona eteenpäin syntaktiseen analyysiin. Merkit lähtevät siinä järjestyksessä kuin ne koodissa esiintyvät.

Kysymykseen: *”Millä tavoin kääntäjät tarkistavat koodin oikeellisuuden? Mistä nämä tarkistustavat ovat tulleet?”* selvisi, että syntaktisessa analyysissä ne asetellaan puurakenteeseen joko ylhäältä alaspäin tai alhaalta ylöspäin -tyyliin. Kääntäjien jäsentimien suunnittelussa on tarkka ja hyvin kehitetty tiede. Jäsenninten karkeat tyypit muodostuivat jo 1960-luvun alkupuolella ja ovat siitä asti jatkaneet hienostumistaan.

Yleisin jäsennintyyppi on LR-jäsennin, tarkalleen ottaen LALR(1)-tyyppi. Katsomalla kysymykseen *”-- mitä loogisia rakenteita kuuluu siihen?”*, selvitettiin DFA, laajennettu kielioppi, GOTO-taulu ja jäsennystoimintotaulu.

Tätä työtä voitaneen käyttää tiivistettynä suomenkielisenä lähteenä jäsentimien toiminnasta ja ponnahduslautana oman jäsentimen, vaikkapa LALR(1)-tyypisen, toteuttamiseen. Itsessään tämä dokumentti ei tietenkään riitä, vaan on perehdyttävä käyttämänsä jäsenningeneraattorin dokumentaatioon. Vastaisuudessa voisi olla perehtymisen arvoista perehtyä ohjelmistokääntämisen prosessin muihin vaiheisiin. Oman ohjelmointikielen luominen tai oman kääntäjän luominen jo olemassa olevalle ohjelmointikielelle voisivat olla mielenkiintoisia aiheita. Ne saattavat kuitenkin olla liian työläitä kandidaantintyöhön.

LÄHTEET

[1] O.G. Kakde, Compiler Design, 2014, Laxmi Publishing.

[2] D. Grune, K. van Reeuwijk, H.E. Bal, C.J.H. Jacobs, K. Langendoen; Modern Compiler Design, 2012; Springer New York

[3] A. Butterfield, G. Ekembe Ngondi, A Dictionary of Computer Science (7 ed.), 2016; Oxford University Press

[4] GCC, the GNU Compiler Collection, GCC:n kotisivu ja dokumentaatio. Saatavissa (viitattu 13.1.): https://gcc.gnu.org

[5] Optimization Levels, Gnu GCC Documentation, osio optimoinnin tasoista. Saatavissa (10.1.): https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html

[6] Compiling GNU/Linux with -O3 optimization, StackExchange: Unix & Linux. Saatavissa (viitattu 16.1.): https://unix.stackexchange.com/questions/1597/compiling-gnu-linux-with-o3-optimization

[7] GNU:n kotisivu. Saatavissa (viitattu 16.1.): https://www.gnu.org

[8] Tieteen termipankki, automaton-artikkeli. Saatavissa (viitattu 26.2.): http://tieteentermipankki.fi/wiki/Language_Technology:automaton

[9] J. Nicholson, The Concise Oxford Dictionary of Mathematics (5 ed.), Oxford University Press, 2014. Saatavissa (viitattu 15.3.): http://www.oxfordreference.com.libproxy.tut.fi/view/10.1093/acref/9780199679591.001.0001/acref-9780199679591

[10] B. Slivnik, LLLR parsing: A combination of LL and LR parsing, OpenAccess Series in Informatics (Vol. 51), 2016. Saatavissa (viitattu 1.6.): https://doi.org/10.4230/OASIcs.SLATE.2016.5

[11] B. Slivnik, On different LL and LR parsers used in LLLR parsing. Computer Languages, Systems & Structures Volume 50, December 2017, Pages 108-126, 2017. Saatavissa (viitattu 1.6.): https://www.sciencedirect.com/science/article/pii/S1477842416301853

[12] M. Griffiths, LL(1) Grammars and Analysers, teoksessa: F.L. Bauer, J. Eickel (eds) Compiler Construction. Lecture Notes in Computer Science, vol 21. Springer, Berlin, Heidelberg, 1974.

[13] F. L Bauer, Historical Remarks on Compiler Construction, Bauer F.L., Eickel J. (eds) Compiler Construction, Lecture Notes in Computer Science, vol 21., Springer, Berlin, Heidelberg, 1974.

[14] A. Paracha, F. Franek, Testing Grammars For Top-Down Parsers, teoksessa: Sobh T. (eds) Innovations and Advances in Computer Sciences and Engineering. Springer, Dordrecht, 2010.

[15] A. Kedawat, Parsing techniques and conflict in LR parsers, International Journal of Advanced Research in Computer Science, 6(1), 2015, Saatavissa (viitattu 4.7.): https://lib-proxy.tuni.fi/login?url=https://search.proquest.com/docview/1674899862?accountid=14242

[16] C. Liang, A Deterministic Shift-Reduce Parser Generator for a Logic Programming Language, teoksessa: Lloyd J. et al. (eds) Computational Logic — CL 2000, CL 2000, Lecture Notes in Computer Science, vol 1861. Springer, Berlin, Heidelberg, 2000.

[17] R. Wilhelm, H. Seidl, S. Hack, Compiler Design: Syntactic and Semantic Analysis, Springer, 2013.

[18] K.D. Cooper, L. Torczon, Engineering a Compiler, Morgan Kaufmann, 2004, Saatavissa (viitattu 14.8.): http://search.ebscohost.com/login.aspx?direct=true&AuthType=cookie,ip,uid&db=nlebk&AN=189683&site=ehost-live&scope=site

[19] GNU Bison 3.4 Documentation, Free Software Foundation Inc., 2019, Saatavissa (siteerattu 19.8.2019): https://www.gnu.org/software/bison/manual/bison.html