TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

KALLE TAMMINEN

KÄÄNTÄJÄN JÄSENNYSMENETELMIEN VERTAILU

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen
Jätetty tarkastettavaksi 29.4.2018

TIIVISTELMÄ

KALLE TAMMINEN: Kääntäjän jäsennysmenetelmien vertailu
Tampereen teknillinen yliopisto
Kandidaatintyö, 22 sivua, 2 liitesivua
Huhtikuu 2018
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: kääntäjä, syntaksianalyysi, jäsennin, top–down, bottom–up

Ohjelmointikielen kääntäjän tehtävä on kääntää jollakin ohjelmointikielellä kirjoitettu ohjelma lähdekoodista konekieliseksi. Ohjelmointikielet sopivat käytettävyydeltään ihmisten kirjoitettavaksi, mutta niitä ei voi sellaisenaan suorittaa laitteistolla. Tämän vuoksi vaaditaan kääntäjä, joka suorittaa koodille käännöksen.

Työssä tutkittavat jäsennysmenetelmät suorittavat syntaksianalyysin, joka on yksi käännösprosessin vaiheista. Syntaksianalyysin tehtävä on analysoida koodista sen rakenne, ja esittää se sopivassa muodossa kääntäjän seuraavalle vaiheelle. Tyypillisesti tämä muoto on puurakenne. Vertailtavat jäsennysmenetelmät, eli top–down- ja bottom–up-jäsennys suorittavat puurakenteen muodostamisen päinvastaisissa järjestyksissä, eli ylhäältä alas tai alhaalta ylös. Lähdekoodin rakenteen tulkitsemisen apuna käytetään niin sanottuja kontekstittomia kielioppeja, joiden käyttö eroaa vertailtavien menetelmien välillä.

Työn tavoitteena oli selvittää minkälaisiin sovelluskohteisiin top–down- ja bottom–up-jäsennysmenetelmät sopivat parhaiten. Tätä pyrittiin arvioimaan vertailemalla menetelmien ominaisuuksia. Vertailun tuloksena todettiin, että top–down-jäsennys voi sopia paremmin pienemmille projekteille sen yksinkertaisemman toteutuksen vuoksi, mutta monimutkaisemmille ohjelmointikielille sen asettamat rajoitteet saattavat olla esteenä. Tämän vuoksi laajemman ohjelmointikielen kääntäjää toteutettaessa bottom–up-jäsennys saattaa olla sopivampi, vaikka se voikin olla toteutukseltaan monimutkaisempi.

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. KÄÄNTÄJÄN TOIMINTA 2
3. SYNTAKSIANALYYSIN PERUSTEET 4
3.1 Kontekstiton kielioppi 4
3.2 Jäsennyspuu ja abstrakti syntaksipuu 5
4. JÄSENNYSMENETELMÄT 8
4.1 Top–down-jäsennys 8
4.2 Bottom–up-jäsennys 11
5. VERTAILU 16
5.1 Vertailuperusteet 16
5.1.1 Vaatimukset kieliopille 16
5.1.2 Suhde generointityökaluihin 16
5.1.3 Toteuttamisen haasteellisuus 17
5.1.4 Virheidenkäsittely 17
5.2 Vertailun suoritus 18
6. YHTEENVETO 21
LÄHTEET 22
LIITE A: ESIMERKKI RDP:STÄ 23

LYHENTEET JA MERKINNÄT

AST	eng. abstract syntax tree, abstrakti syntaksipuu
IC	eng. intermediate code, välikoodi
LL(1)	eng. Left-to-right, leftmost derivation, yhden merkin ennakointi
LL(*k*)	eng. Left-to-right, leftmost derivation, *k*:n merkin ennakointi
LR(1)	eng. Left-to-right, rightmost derivation, yhden merkin ennakointi
LR(*k*)	eng. Left-to-right, rightmost derivation, *k*:n merkin ennakointi
RDP	eng. recursive descent parsing, rekursiivinen laskeva jäsennys

1. JOHDANTO

Kääntäjien rooli ohjelmistokehityksessä on ollut erittäin tärkeä useiden vuosikymmenien ajan. Kääntäjät mahdollistavat korkean tason ohjelmointikielien käytön, sen sijaan että ohjelmia kirjoitettaisiin suoraan konekielelle. Tämä yksinkertaistaa ohjelmointia huomattavasti.

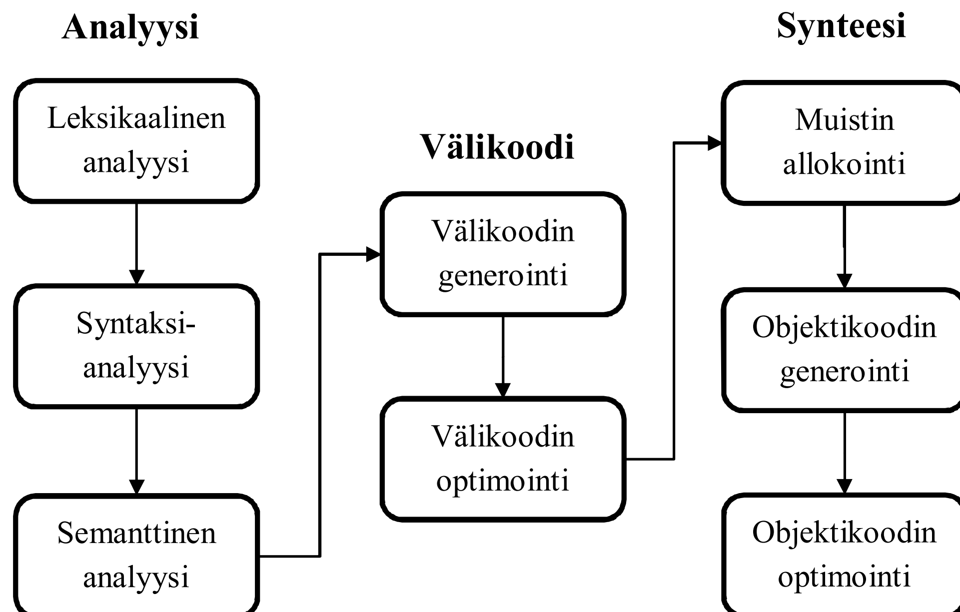
Tässä kandidaatintyössä vertaillaan jäsennysmenetelmiä, joilla syntaksianalyysi, yksi käännösprosessin vaiheista, voidaan toteuttaa. Yleisesti kääntäjissä käytetyt jäsennysmenetelmät jakautuvat kahteen tyyppiin, ylhäältä alas (top–down) ja alhaalta ylös (bottom–up) -metodeihin. Jäsennysmenetelmiä voidaan edelleen jakaa erilaisiin toteutuksiin näiden kahden ryhmän sisällä. Työn tavoitteena on selvittää erityisesti, minkälaisiin tapauksiin erilaiset jäsennysmenetelmät soveltuvat parhaiten. Vastaus pyritään muodostamaan vertailemalla eri jäsennysmenetelmien toteuttamiseen liittyviä vaatimuksia ja etuja. Lisäksi tarkastellaan menetelmien mahdollisuuksia virheiden tunnistamiseen, koska tarkka virheiden paikannus ja kuvaus voi nopeuttaa ohjelmoijan työtä merkittävästi.

Työn 2. luvussa lukijalle kuvataan koko käännösprosessi ja esitellään lyhyesti sen vaiheet. Sen jälkeen 3. luvussa esitellään tarkemmin teoriaa, johon syntaksianalyysi perustuu. 4. luvussa esitellään jäsennysmenetelmät, joita työssä vertaillaan. Varsinainen vertailu suoritetaan 5. luvussa. Työn yhteenveto on viimeisessä eli 6. luvussa.

2. KÄÄNTÄJÄN TOIMINTA

Kääntäjän tehtävä on kääntää yhdellä kielellä kirjoitettu ohjelma toiselle kielelle eli muuttaa lähdekoodi objektikoodiksi (object code, target code) (Aho et al. 1986, p. 1; Grune et al. 2012, p. 1). Tyypillisesti lähdekoodi on ohjelmoijan jollakin ohjelmointikielellä kirjoittamaa, ja objektikoodi on laitteistolla suoritettavaksi soveltuvaa eli konekieltä. Vaikka lähde- ja objektikoodi voisivat periaatteessa olla mitä tahansa kieliä, tehdään osassa lähdekirjallisuuttakin tällainen oletus (Hunter 1999, p. 1).

Käännösprosessi koostuu useista vaiheista. Tyypillisesti prosessi jaetaan ensin karkeasti kahteen vaiheeseen, joita kutsutaan analyysi- ja synteesivaiheeksi (analysis stage, synthesis stage) (Wilhelm & Maurer 1995; Hunter 1999), tai front- ja back-endiksi (Aho et al. 1986; Grune et al. 2012). Nämä kaksi vaihetta jaetaan edelleen pienempiin vaiheisiin. Näiden pienempien vaiheiden määrittelyssä on paljon tulkinnallisia eroja, erityisesti välikoodin sijoituksen osalta. Tämän vuoksi kuvaan 1 kootussa esityksessä välikoodi on analyysivaiheen ja synteesivaiheen välissä eikä osana kumpaakaan niistä.



Kuva 1: Kääntäjän rakenne, perustuu lähteisiin (Wilhelm & Maurer 1995, p. 223, Fig 6.1; Hunter 1999, p. 5, Fig. 1.6; Grune et al. 2012, p. 23, Fig. 1.21).

Kääntäjän ensimmäisen vaihe on leksikaalinen eli sanastollinen analyysi (lecical analysis). Kääntäjällä on tässä vaiheessa käytössään vain puhdas tekstimuotoinen tiedosto.

Leksikaalisen analyysin tehtävä on tunnistaa tästä tiedostosta erilleen käytetyn ohjelmointikielen tunnisteet (token). Ohjelmointikielissä tunnisteita voivat olla esimerkiksi merkkijonot, luvut, sulkeet, muuttujien nimet ja avainsanat (esim. for). Leksikaalinen analyysi perustuu yleensä säännöllisten lausekkeiden (regular expression) käyttöön. (Grune et al. 2012)

Seuraava vaihe on tämän kandidaatintyön kohteena oleva syntaksianalyysi (syntax analysis). Syntaksianalyysin suorittamisesta käytetään verbiä jäsentää (parse), ja kääntäjien kontekstissa syntaksianalyysi ja jäsentäminen ovat käytännössä synonyymeja. Näistä jälkimmäinen termi on kuitenkin paremmin taivutettavissa sanana. Syntaksianalyysia suorittavan kääntäjän osan, eli jäsentimen (parser), tehtävä on tunnistaa leksikaalisella analyysilla löydetyistä tunnisteista ohjelman rakenne. Usein tuloksena pyritään muodostamaan abstrakti syntaksipuu (abstract syntax tree, AST). Syntaksianalyysi perustuu kontekstittomien kielioppien käyttöön. (Grune et al. 2012) Syntaksianalyysin teoriaa kuvataan tarkemmin luvussa 3.

Syntaksianalyysi tarkastelee kuitenkin vain ohjelman rakennetta, eikä se esimerkiksi havaitse puuttuvaa muuttujan alustusta tai väärää tietotyyppiä olevan muuttujan käyttöä. Tällaisen kontekstista riippuvan tiedon käsittely tehdään vasta semanttisella analyysilla (semantic analysis). (Hunter 1999, pp. 147–148)

Kun leksikaalinen, syntaksi- ja semanttinen analyysi on suoritettu, generoidaan välikoodi (intermediate code, IC). Välikoodin on tarkoitus olla mahdollisimman riippumatonta käytetystä lähdekoodin ja objektikoodin kielestä. Vähentämällä lähdekoodin ja objektikoodin välistä riippuvuutta kääntäjistä saadaan joustavampia. Tämä pienentää tarvetta luoda täysin uusi kääntäjä jokaiselle erilaiselle lähdekoodin ja objektikoodin yhdistelmälle. (Hunter 1999, p. 9)

Synteesivaihe voi koostua monenlaisista erilaisista alavaiheista. Ennen lopullista objektikoodia saatetaan generoida myös muita välimuotoisia koodeja. Optimointia saatetaan tehdä useassa eri vaiheessa, mutta joissain toteutuksissa sitä ei välttämättä tehdä ollenkaan. Lisäksi yhden generointivaiheen yhteydessä tehdään muistin allokointi. (Hunter 1999, p. 9)

3. SYNTAKSIANALYYSIN PERUSTEET

Kuten edellisessä luvussa todettiin, jäsennin saa käännettävän ohjelman sarjana tunnisteita, jotka leksikaalinen analyysi on tunnistanut. Ohjelman rakenne tunnistetaan tästä sarjasta käyttämällä ohjelmointikielelle määriteltyä kontekstitonta kielioppia. (Grune et al. 2012, p. 115)

3.1 Kontekstiton kielioppi

Kontekstiton kielioppi (context-free grammar) koostuu päätemerkeistä (terminal symbol) ja välikemerkeistä (non-terminal symbol), sekä näistä muodostetuista produktioista (production). Lisäksi yksi välikemerkeistä on myös aloitusmerkki. (Aho et al. 1986, p. 165–166; Grune et al. 2012, p. 35; Kaijanaho 2016) ”Päätemerkki” ja edellisessä luvussa käytetty ”tunniste” ovat käytännössä synonyymeja, ja käytetty termi riippuu kontekstista. Pienenä poikkeuksena tähän on päätemerkki ε, jolla voidaan merkitä merkkijonoa jossa ei ole yhtään tunnistetta (Aho et al. 1986, p. 27). ε ei kuitenkaan kuulu kaikkiin kielioppeihin.

```
expression → digit | ’(’ expression operator expression ’)’
operator → ’+’ | ’*’
digit → ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’
```

Ohjelma 1: Esimerkki kieliopin produktioista (Grune et al. 2012, p. 13, Fig. 1.9).

Ohjelma 1 on esimerkki yksinkertaisen kieliopin produktioista. Produktiot koostuvat vasemmasta ja oikeasta puolesta, jotka on erotettu toisistaan nuolimerkillä. (Grune et al. 2012, p. 35) Kontekstittomissa kieliopeissa produktion vasen puoli on aina tasan yksi välikemerkki (Hunter 1999, p. 25). Oikealla puolella voi olla sekä välikemerkkejä että päätemerkkejä. Ohjelma 1 käyttää merkintätapaa, jossa päätemerkit ovat heittomerkkien sisällä. Oikealla puolella käytetään pystyviivaa erottamaan vaihtoehtoja, joiksi vasemman puolen päätemerkki voidaan johtaa. (Grune et al. 2012, p. 37) Esimerkiksi välikemerkki `operator` voidaan siis johtaa joko plus- tai kertomerkiksi. Johdettava muoto voi myös koostua useasta merkistä, kuten välikemerkin `expression` toinen johto, joka koostuu kahdesta päätemerkistä ja kolmesta välikemerkistä.

Koska vain päätemerkit kuuluvat kieleen, välimerkeistä on johdettava produktioilla lauseita, joissa on vain päätemerkkejä. Esimerkiksi Ohjelma 1:n kieliopilla voitaisiin muodostaa yksinkertaisia matemaattisia lausekkeita, kuten 2, (4+5), tai ((5*5)*(3+9)).

Aho et al. esittää kaksi ehtoa (2006, p. 204), jotka on osoitettava todeksi, jos halutaan todistaa kieliopin G generoivan tietyn kielen L:

1. Kaikki merkkijonot, jotka kieliopilla G on mahdollista generoida, kuuluvat kieleen L.
2. Kaikki merkkijonot, jotka kuuluvat kieleen L, on mahdollista generoida kieliopilla G.

Kieliopin ja ohjelmointikielen on siis vastattava toisiaan täysin. Kieliopilla ei saa pystyä luomaan lauseita, jotka eivät kuulu käytettävään ohjelmointikieleen. Vastaavasti kaikki ohjelmointikielen lauseet on pystyttävä kuvaamaan sille luodulla kieliopilla.

Samalle ohjelmointikielelle saattaa silti pystyä luomaan useita erilaisia kielioppeja, jotka vastaavat sitä täydellisesti. Jos kaksi eri kielioppia kuvaavat tarkalleen saman kielen, niiden sanotaan olevan ekvivalentteja. (Hunter 1999, pp. 23–24)

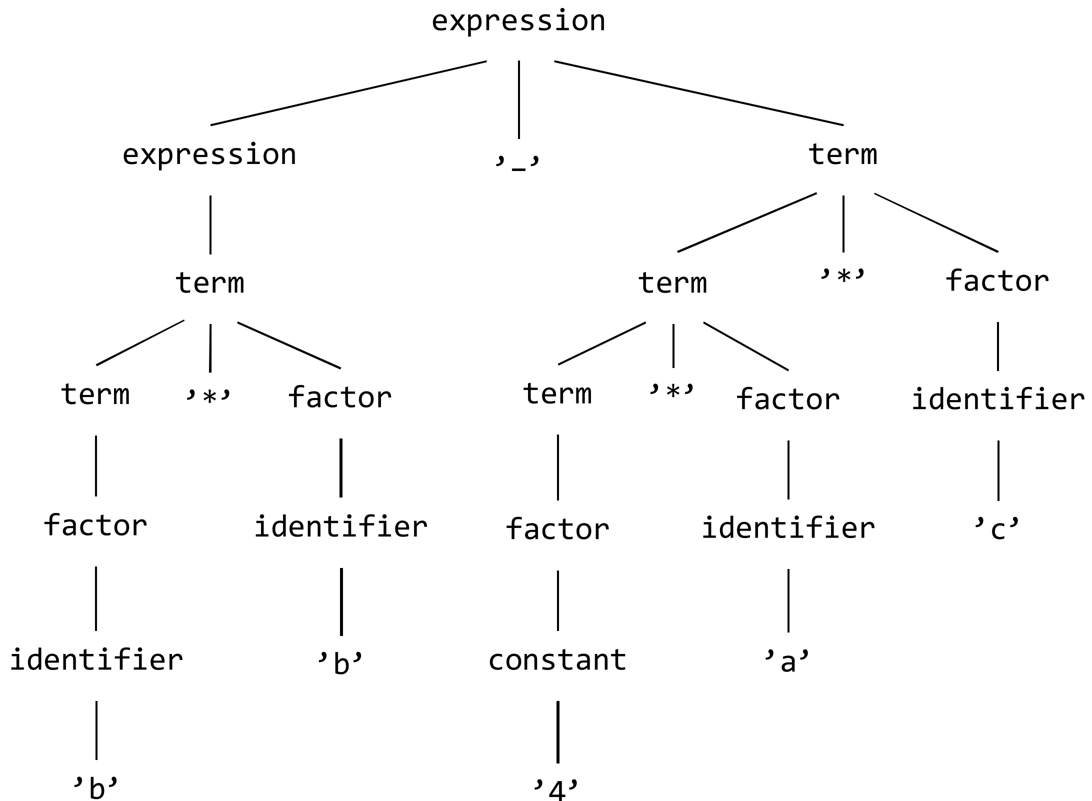
3.2 Jäsennyspuu ja abstrakti syntaksipuu

Syntaksianalyysin tavoitteena on ohjelman rakenteen tunnistaminen, joten sen lopputulos tallennetaan tyypillisesti johonkin tietorakenteeseen. Lähdekirjallisuuden esimerkeissä tähän käytetään pääasiassa erilaisia puurakenteita (Aho et al. 1986, p. 6; Wilhelm & Maurer 1995, p. 266; Hunter 1999, p. 7). Muita tietorakenteita, kuten linkitetty lista, mainitaan lähinnä ohimennen (Grune et al. 2012, p. 9). Tämän lähdekirjallisuudessa olevan painotuksen vuoksi myös tässä työssä keskitytään vain puurakenteisiin.

```
expression → expression ’+’ term | expression ’-’ term | term
term → term ’*’ factor | term ’/’ factor | factor
factor → identifier | constant | ’(’ expression ’)’
```

Ohjelma 2: Esimerkki kieliopin produktioista (Grune et al. 2012, p. 10, Fig. 1.5).

Ohjelma 2 on esimerkki hieman Ohjelma 1:tä monimutkaisemmasta kieliopista. Oleellisena erona kielioppiin on lisätty muuttujien nimet (`identifier`) ja monimutkaisempia matemaattisia operaattoreita.



Kuva 2: Jäsennyspuu, perustuu lähteeseen (Grune et al. 2012, p. 11, Fig. 1.5).

Jäsentämisen tuloksena saadaan ensin jäsennyspuu (parse tree). Kuva 2 on esimerkki jäsennyspuusta, kun kielioppina on Ohjelma 2 ja lauseena on b*b–4*a*c. Jos jäsennyspuuta tarkastelee aloittaen sen juurisolmusta, voidaan seurata produktioita, joilla tunnistesarja saadaan johdettua. Esimerkiksi johdettaessa lausetta b*b–4*a*c aloitetaan siis produktiolla `expression → expression ’-’ term`.

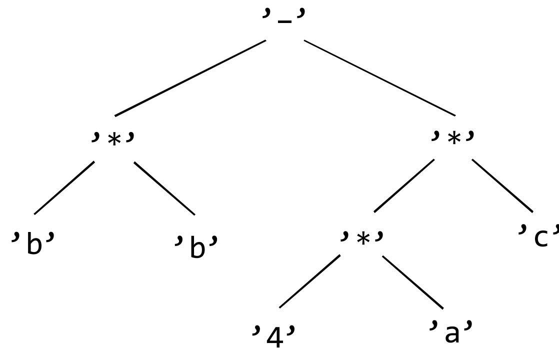
Jäsennyspuun rakenteen on noudatettava seuraavia sääntöjä:

1. Kaikissa lehtisolmuissa on päätemerkki (eli jokin tunniste tai ε) ja kaikissa sisäsolmuissa on jokin käytetyn kieliopin välikemerkki.
2. Lehtisolmujen päätemerkit, sekä järjestys jossa ne ovat, vastaavat syötteen tunnisteita (poikkeuksena ε, jota ei luonnollisesti löydy tunnistesarjasta, mutta joka voi olla lehtisolmussa.).
3. Sisäsolmu (esim. A) ja sen lapsisolmut (esim. x_1, x_2, … x_n) vastaavat jotakin produktiota (esim. $A \rightarrow x_1, x_2, \ldots x_n$).
4. Puun juurisolmussa on kieliopille määrätty aloitusmerkki.

(Aho et al. 1986, p. 29; Wilhelm & Maurer 1995, p. 273; Grune et al. 2012, p. 117; Kaijanaho 2016, s. 19)

Jäsennyspuiden ja kontekstittomien kielioppien väliseen yhteyteen liittyvät oleellisesti

myös termit moniselitteisyys ja yksiselitteisyys. Jos kielioppi on moniselitteinen, se sisältää lauseita, joita vastaa useampi kuin yksi jäsennyspuu. Vastaavasti yksiselitteisessä kieliopissa jokaista lausetta voi vastata vain yksi jäsennyspuu. (Wilhelm & Maurer 1995, p. 273) Osa työssä myöhemmin esiteltävistä jäsennysmenetelmistä toimii vain yksiselitteisillä kieliopeilla.



Kuva 3: Abstrakti syntaksipuu, perustuu lähteeseen (Grune et al. 2012, p. 11, Fig. 1.6).

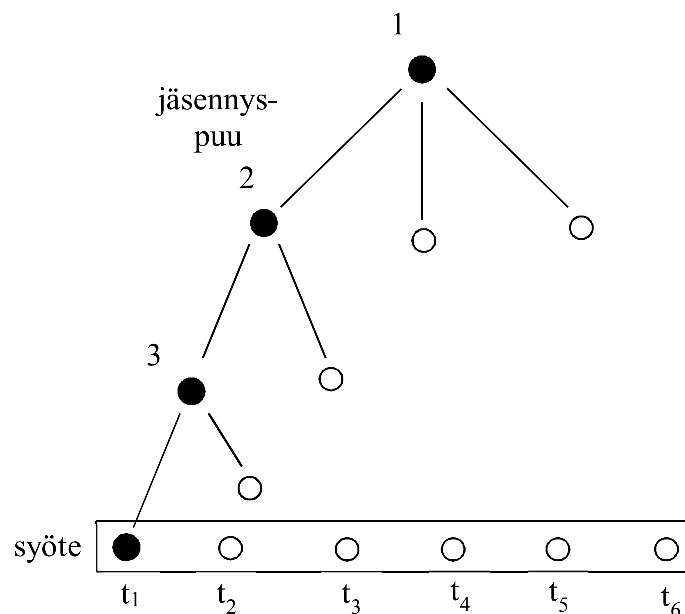
Kun jäsentäminen on suoritettu, kieliopin välikemerkit eivät ole enää oleellisia, joten jäsennyspuu muokataan tyypillisesti abstraktiksi syntaksipuuksi (abstract syntax tree, AST) (Grune et al. 2012, pp. 10–11). Kuva 3 on esimerkki abstraktista syntaksipuusta, kun lauseena on aikaisemmin käytetty b*b−4*a*c.

4. JÄSENNYSMENETELMÄT

Tässä työssä vertailtavat syntaksianalyysin jäsennysmenetelmät jaetaan top–down- ja bottom–up-menetelmiin. Top–down eli ylhäältä alas -jäsennys aloittaa kieliopin aloitusmerkistä ja generoi jäsennyspuun solmut juuresta alaspäin, kohti syötelauseen tunnisteita. Bottom–up eli alhaalta ylös -jäsennys puolestaan aloittaa syötelauseesta, ja redusoi sen kohti jäsennyspuun juurta ja kieliopin aloitusmerkkiä. (Grune et al. 2012, p. 117)

4.1 Top–down-jäsennys

Kuten Grune at al. (2012, p. 119) ilmaisee, voidaan mieltää, että top–down-jäsentimen tehtävä on valita produktiosta sopiva vaihtoehto jokaiselle kohdattavalle välikemerkille.



Kuva 4: Top–down-jäsennys aloitusmerkistä ensimmäiseen tunnisteeseen, perustuu lähteeseen (Grune et al. 2012, p. 119, Fig. 3.2).

Kuva 4 on hieman yksinkertaistettu versio Grune et al. antamasta esimerkistä top–down-jäsennyksen rakennusjärjestykselle (2012, p. 119). Kuvassa puun mustat solmut ovat käsiteltyjä välikemerkkejä ja valkoiset ovat tunnettuja, mutta toistaiseksi käsittelemättömiä solmuja. Puun rakenteesta ei vielä tiedetä muuta. Kuvassa 4 syötteen ensimmäinen tunniste on käsitelty, eli solmu 3:ssa olevan välikemerkin produktio tuottaa sen. Luonnollisesti myös muut syötteen tunnisteet tiedetään, vaikka välikemerkkejä joista ne johdetaan ei tiedetäkään.

Yksinkertainen top–down-jäsennysmenetelmä on rekursiivinen laskeva jäsennys (recursive descent parsing, RDP). Tällöin jäsennin etenee suoraviivaisesti kohti käsiteltävänä olevaa tunnistetta kokeilemalla vaihtoehtoisia produktiota. Vaikka jäsennettäessä ei vielä tiedetäkään mille jäsennyspuun pitäisi lopuksi näyttää, puu voidaan tunnistaa virheelliseksi, jos se ei vastaa syötteen tunnistesarjaa. Jos puolestaan saadaan tuotettua tunnisteita vastaava puu, se voidaan olettaa oikeaksi, kunhan käytetty kielioppi on yksiselitteinen. Algoritmi hyväksyy ensimmäisen sopivan sarjan produktioita, joilla päästään oikeaan tunnisteeseen asti. Jos algoritmi päätyy väärään tunnisteeseen, se palaa taaksepäin ja kokeilee läpi muut vaihtoehdot. (Grune et al. 2012, p. 122–124)

Yksi RDP:n etu on kieliopin ja jäsentimen koodin välinen suora yhteys. Kieliopin produktiot kääntyvät helposti kääntäjän funktioiksi, jotka palauttavat totuusarvon sen mukaan löytyykö syötesarjasta vastaava tunniste. Tästä hyvä esimerkki on liitteessä A, joka sisältää Grune et al. esittämän yksinkertaisen kieliopin, ja sitä vastaavan RDP:n (2012, pp. 123–124).

Liitteen A Ohjelma 1 kuvaa esimerkin kieliopin, Ohjelma 2 kuvaa varsinaisen jäsentimen ja Ohjelma 3 niin sanotun ajurin, joka vastaa jäsentimen kutsumisesta sekä muista kieliopista riippumattomista toiminnoista. Liite A:n RDP-toteutus nojaa paljolti `require()`-funktion sekä Boolen operaattoreiden && (AND) ja || (OR) käyttöön. Jäsennin aloittaa kutsumalla `require()`-funktiota parametrilla `input()`. `Input()` vastaa merkitykseltään kieliopin ensimmäistä produktiota, joka johtaa kieliopin aloitusmerkistä `input` välikemerkin `expression` sekä tiedoston loppua merkitsevän `EoF`:n. Tämän vuoksi `input()` palauttaa totuusarvon 1, kun funktio `expression()` palauttaa totuusarvon 1 ja `require()` parametrilla `token(EoF)` palauttaa totuusarvon 1. Jotta `expression()`-funktion palauttama arvo saadaan selville, pitää edelleen tarkastella sen kutsumia funktioita, jotka puolestaan vastaavat muita kieliopin merkkejä. Syötteen tunnisteiden tarkastaminen ja seuraavan tunnisteen hakeminen tehdään `token()`-funktiolla.

Tämän vähäisen koodimäärän vuoksi RDP on siis nopea tuottaa käsin, ilman että vaadittaisiin muita työkaluja jäsentimen generointiin (Hunter 1999, p. 80). RDP:ssä on kuitenkin rajoituksensa. Usein se ei tuota toimivaa jäsennintä, ja sen virheiden käsittely on rajoittunutta (Grune et al. 2012, p. 124). Esimerkin RDP-toteutus ei myöskään osaa tehdä minkäänlaista älykästä tunnisteiden tarkastelua, vaan se kokeilee läpi jokaisen mahdollisen vaihtoehdon, kunnes sopiva tunniste löytyy. Lisäksi se vaatii LL(1)-kieliopin (Hunter 1999, p. 80), joista on lisää seuraavana LL(1)-jäsentimien yhteydessä.

Merkintä LL(1) tarkoittaa syötteentunnisteiden lukemista vasemmalta oikealle (left-to-right, L), vasemmanpuolimmaisimman välikemerkin johtamista (leftmost derivation, L) ja yhden merkin ennakointia (one symbol lookahead, (1)). LL-jäsennystä voidaan myös tehdä suuremmalla ennakoinnilla, jolloin käytetään merkintää LL(k), missä k on ennakoitavien merkkien määrä. (Hunter 1999, p. 78; Grune et al. 2012, p. 129)

LL-jäsentimet käyttävät apunaan ns. FIRST- ja FOLLOW-joukkoja. Ilmaisulla FIRST(α) merkitty joukko, missä α on jokin pääte- ja välikemerkeistä muodostuva merkkijono, sisältää kaikki mahdolliset päätemerkit, joilla α voi alkaa. Jos α alkaa päätemerkillä, FIRST(α) sisältää vain kyseisen päätemerkin. Jos α alkaa välikemerkillä, FIRST(α) sisältää kyseisen välikemerkin vaihtoehtoisten produktioiden FIRST-joukot. Jos α voi tuottaa tyhjän merkkijonon ε, myös se kuuluu FIRST(α)-joukkoon. (Grune et al. 2012, p. 126–127) Jo pelkkä FIRST-joukko tuntemalla voidaan kehittää RDP:tä edistyneempi ennakoiva jäsennin (predictive parser, predictive recursive descent parser) (Grune et al. 2012, p. 129).

FOLLOW(α) -joukko puolestaan sisältää kaikki päätemerkit, jotka voivat esiintyä heti α:n oikealla puolella. Jos esimerkiksi merkkijonoa α seuraa merkkijono β, missä myös β voi koostua pääte- ja välikemerkeistä, FOLLOW(α) sisältää kaikki merkit joilla β voi alkaa. Jos β:n ensimmäinen päätemerkki voi olla tyhjän merkkijono ε, pitää tarkastelua jatkaa vielä sitä mahdollisesti seuraaviin merkkeihin. (Aho et al. 2006, pp. 221–222)

```
A → C B
B → ’+’ C B | ε
C → E D
D → ’*’ F D | ε
E → ’(’ A ’)’ | ’id’
```

Ohjelma 3: Kielioppi FIRST- ja FOLLOW-joukkojen esimerkkiä varten, perustuu lähteeseen (Aho et al. 2006, p. 217).

```
FIRST(E) = FIRST(C) = FIRST(A) = { ‘(‘, ‘id’ }
FIRST(B) = { ‘+’, ε }
FIRST(D) = { ‘*’, ε }

FOLLOW(A) = FOLLOW(B) = { ‘)’, EoF }
FOLLOW(C) = FOLLOW(D) = { ‘+’, ‘)’ , EoF }
FOLLOW(E) = { ‘+’, ‘*’, ‘)’, EoF }
```

Ohjelma 4: Esimerkki FIRST- ja FOLLOW-joukoista, perustuu lähteeseen (Aho et al. 2006, p. 222).

Ohjelma 4 on esimerkki Ohjelma 3:lle luodoista FIRST- ja FOLLOW-joukoista. Kieliopin aloitusmerkki on `A`. Muut välikemerkit ovat `B`, `C`, `D` ja `F`. Kaikki päätemerkit ovat heittomerkkien sisällä. Lisäksi `EoF` merkitsee tiedoston loppua (eng. End of File) ja ε tyhjää merkkijonoa. Ohjelma 4:ssä esimerkiksi `A`:n, `C`:n ja `E`:n FIRST-joukot ovat samat, koska `A` johtaa ensimmäisenä aina `C`:n, `C` johtaa ensimmäisenä aina `E`:n ja `E`:n mahdolliset johdot alkavat kaarisulkeella ja `id`:llä.

Aho et al. mukaan (2006, p. 223) kielioppi on LL(1) jos ja vain jos se täyttää seuraavat rajoitteet, kun kieliopilla on produktiot A → α | β, missä A on välikemerkki ja α ja β ovat joitakin pääte- ja välikemerkeistä koostuvia merkkijonoja:

1. α ja β eivät voi johtaa mitään samalla päätemerkillä alkavia merkkijonoja, eli FIRST(α) ∩ FIRST(β) = Ø.
2. α ja β eivät saa molemmat pystyä johtamaan tyhjää merkkijonoa. Eli joko vain α

tai β voi johtaa tyhjän merkkijonon, tai kumpikaan ei niistä ei voi.

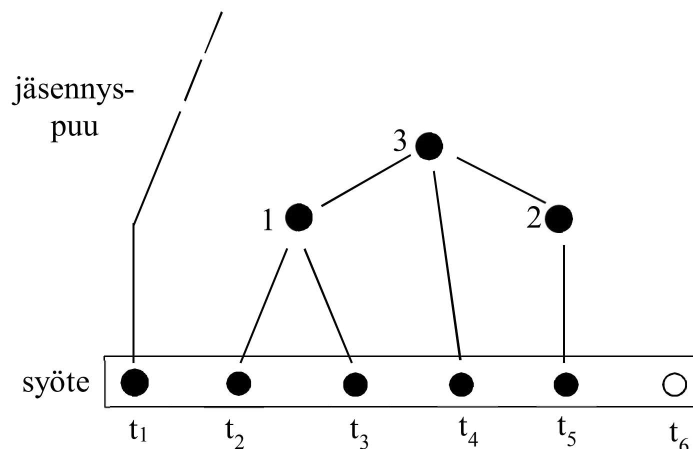
3. Jos β voi johtaa tyhjän merkkijonon, α ei voi johtaa mitään merkkijonoa, joka alkaa päätemerkillä, joka on joukossa FOLLOW(A). Vastaavasti, jos α voi johtaa tyhjän merkkijonon, β ei voi johtaa mitään merkkijonoa, joka alkaa päätemerkillä, joka on joukossa FOLLOW(A).

LL(1) ei siten esimerkiksi voi sisältää vasenta rekursiota ja se on oltava yksiselitteinen. Myös Grune et al. (2012, p. 133) esittää vastaavat kolme sääntöä.

Jos kielioppi ei ole LL(1), vaihtoehtoina on joko käyttää jäsennysmenetelmää, joka ei vaadi sitä tai muokata kielioppi sellaiseen muotoon että se on LL(1). Grune et al. (2012, p. 133) kehottaa ensin mainittuun, koska tällöin kielioppi voidaan pitää alkuperäisessä muodossaan.

4.2 Bottom–up-jäsennys

Bottom–up-jäsennys etenee päinvastoin verrattuna top–down-jäsennykseen, eli jäsennyspuun solmut rakennetaan aloittaen lähimpänä tunnisteita olevaista. Bottom–up-jäsennin rakentaa puun solmun, kun kaikki sen lapsisolmut on käsitelty (Grune et al. 2012, p. 119).



Kuva 5: Bottom–up-jäsennys, perustuu lähteeseen (Grune et al. 2012, p. 120, Fig. 3.3).

Kuva 5 on hieman yksinkertaistettu versio Grune et al. antamasta esimerkistä bottom–up-jäsennyksen solmujen rakennusjärjestykselle (2012, p. 120). Kuten numeroitujen solmujen rakennusjärjestyksestä huomataan, solmu voidaan rakentaa vasta kun kaikki sen lapsisolmut on käsitelty. Tämän vuoksi solmu 3 rakennetaan solmun 2 jälkeen, vaikka se vasemmalta oikealle edetessä kohdataan ensin.

Puurakenteessa ylöspäin eteneminen vastaa merkkijonojen tasolla sitä, että produktioita sovelletaan oikealta vasemmalle. Bottom–up-jäsennys siis etenee tunnistamalla produktioiden oikealla puolelle olevia välike- ja päätemerkkejä, ja johtamalla niitä vasemman

puolen välikemerkeiksi. Tästä käytetään termiä reduktio (reduction). (Aho et al. 2006, p. 234)

Yleinen bottom–up-jäsennyksen muoto on shift–reduce-jäsennys, joka nimensä mukaisesti perustuu shiftaus- ja reduktio-operaatioiden käyttöön. Shift–reduce-jäsennyksessä käytetään apuna pinoa, johon saadusta syötteestä siirretään eli shiftataan tunnisteita. Lisäksi käytetään virhe-operaatiota, jos jäsentäessä päädytään syntaksivirheeseen, ja hyväksymis-operaatiota, jos jäsennys suoritetaan onnistuneesti. (Aho et al. 2006, pp. 236–237)

```
E → E '+' T | T
T → T '*' F | F
F → '(' E ')' | id
```

Ohjelma 5: Yksinkertainen kielioppi shift–reduce-jäsennyksen esimerkkiä varten, perustuu lähteeseen (Aho et al. 2006, p. 193).

Ohjelma 5:n kielioppi on määritetty shift–reduce-jäsennyksen esimerkkiä varten. Merkinnät vastaavat aikaisemminkin käytettyjä, eli päätemerkit ovat heittomerkeissä tai tummennetulla tekstillä, ja välikemerkkejä merkitään isoilla kirjaimilla.

Taulukko 1: Shift–reduce-jäsennys, perustuu lähteeseen (Aho et al. 2006, p. 237, Fig. 4.28).

Rivi	Pino	Syöte	Operaatio
(1)	$	$\mathbf{id}_1$ '*' $\mathbf{id}_2$ $	Shiftaus
(2)	$ $\mathbf{id}_1$	'*' $\mathbf{id}_2$ $	Reduktio F → **id**
(3)	$ F	'*' $\mathbf{id}_2$ $	Reduktio T → F
(4)	$ T	'*' $\mathbf{id}_2$ $	Shiftaus
(5)	$ T '*'	$\mathbf{id}_2$ $	Shiftaus
(6)	$ T '*' $\mathbf{id}_2$	$	Reduktio F → **id**
(7)	$ T '*' F	$	Reduktio T → T '*' F
(8)	$ T	$	Reduktio E → T
(9)	$ E	$	Hyväksyminen

Taulukko 1 havainnollistaa shift–reduce-jäsennystä, kun kielioppi on Ohjelma 5:n mukainen. $-merkkiä käytetään pinossa merkitsemään sen pohjaa, sekä syötteelle merkitsemään sen loppua. Ensimmäisellä rivillä pino on siis tyhjä, ja syötteenä saatu ohjelma, jota ollaan jäsentämässä, on $\mathbf{id}_1$ '*' $\mathbf{id}_2$. Koska pino on tyhjä, ensimmäisenä on pakko suorittaa shift-operaatio, joka siirtää ensimmäisen tunnisteen $\mathbf{id}_1$ syötteestä pinoon. Seuraavana operaationa $\mathbf{id}_1$ voidaan redusoida muotoon F käyttämällä produktiota F → **id**. Myös seuraava operaatio on reduktio.

Shift–reduce-jäsennys siis etenee valitsemalla joko shift- tai reduce-operaation. Kuten top–down-jäsennykselle annetuissa esimerkeissä, myös tällä bottom–up-jäsennyksen muodolla on tiettyjä rajoitteita. Shift–reduce-jäsennys voi päätyä konfliktiin, eli tilaan, jossa ei ole varmuutta mikä operaatio pitäisi suorittaa seuraavaksi. Näistä ovat shift/reduce-konflikti, jossa ei tiedetä kumpi operaatioista on suoritettava seuraavaksi, ja reduce/reduce-konflikti, jossa ei tiedetä mitä useista mahdollisista produktioista pitäisi käyttää redusointiin. (Aho et al. 2006, pp. 238–239)

Myös Taulukko 1:ssä voi huomata, ettei jäsentämistä välttämättä voi suorittaa pelkän kieliopin, pinon ja syötteen avulla. Kuten Aho et al. huomauttaa (2006, p. 242), Ohjelma 5:n kuvaaman kieliopin perusteella neljäntenä operaationa voitaisiin suorittaa reduktio `E → T`. Tällöin kuitenkin päädyttäisiin suoraan kieliopin aloitusmerkkiin. Koska `E '*'` muotoista merkkijonoa ei voida jäsentää, päädyttäisiin virheeseen.

Yleisimmät bottom–up-jäsennyksen menetelmät ovat LR(k)-menetelmiä. Termi LR(k) muodostuu vastaavalla tavalla kuin top–down-jäsentimien LL(k), mutta sillä erotuksella, että R merkitsee oikeanpuolimmaisimman välikemerkin johtamista. (Aho et al. 2006, p. 241) LR-menetelmät ratkaisevat shift–reduce-jäsennyksen ongelmia käyttämällä jäsentämisen apuna kohtia (item), jotka kuvaavat miten pitkälle tietty produktio on tunnistettu ja mitä merkkejä seuraavaksi pitäisi löytyä. Kohtien avulla voidaan seurata jäsentimen tilaa. Tilojen käsittelemiseksi luodaan jäsennystauluja tai pinoautomaatteja (push-down automata). (Grune et al. 2012, p. 159–170) Seuraavassa esimerkissä kuvataan jäsentäminen jäsennystaulun avulla.

```
(1) E → E '+' T    (4) T → F
(2) E → T          (5) F → '(' E ')'
(3) T → T '*' F    (6) F → id
```

Ohjelma 6: Numeroitu kielioppi jäsennystaulua varten, perustuu lähteeseen (Aho et al. 2006, p. 251).

Jotta jäsennystaulun luettavuus olisi parempi, esitetään kielioppi muodossa, jossa jokainen yksittäinen produktio esitetään erikseen ja numeroituna. Tässä muodossa jokaiseen produktioon voi viitata yksiselitteisesti käyttämällä vain sen numeroa. Ohjelma 6 vastaa edelleen merkitykseltään Ohjelma 5:ää.

Taulukko 2: Jäsennystaulu, perustuu lähteeseen (Aho et al. 2006, p. 252, Fig. 4.37).

Tila	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2	8	2	3
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Taulukko 2 on itse jäsennystaulu Ohjelma 6:n kuvaamaa kielioppia varten. Taulukossa ACTION-puolella oleva merkintä `s`i, missä i on jokin mahdollisista tiloista (Taulukko 2:ssa luku väliltä 0–11), tarkoittaa yhden tunnisteen shiftausta ja siirtymistä tilaan i. Merkintä `r`j, missä j on jokin mahdollisten produktioiden numeroista (Taulukko 2:ssa luku väliltä 1–6), tarkoittaa redusointia kyseisellä produktiolla. Merkintä `acc` tarkoittaa hyväk-

symistä, eli jäsentämisen päättymistä onnistuneesti. Taulukossa tyhjään soluun päätyminen tarkoittaa virhettä. Taulukon GOTO-puolella olevat numerot tarkoittavat siirtymistä numeron osoittamaan tilaan. (Aho et al. 2006, p. 251) Jäsennystaulun käyttö kuvataan seuraavan jäsennysesimerkin yhteydessä.

Taulukko 3: LR-jäsennys, perustuu lähteisiin (Aho et al. 2006, p. 247, Fig. 4.34 p. 253, Fig. 4.38).

Rivi	Pino	Symbolit	Syöte	Operaatio
(1)	0		id '*' id $	Shiftaus
(2)	0 5	id	'*' id $	Reduktio F → id
(3)	0 3	F	'*' id $	Reduktio T → F
(4)	0 2	T	'*' id $	Shiftaus
(5)	0 2 7	T '*'	id $	Shiftaus
(6)	0 2 7 5	T '*' id	$	Reduktio F → id
(7)	0 2 7 10	T '*' F	$	Reduktio T → T '*' F
(8)	0 2	T	$	Reduktio E → T
(9)	0 1	E	$	Hyväksyminen

Taulukko 3 kuvaa LR-jäsennyksen jäsennystaulun avulla. Taulukossa on edelleen mukana Symbolit-sarake, mutta todellisuudessa tätä ei tarvita jäsennyksen suorittamiseen, vaan tilan pitäminen pinossa riittää. Viimeisimmän reduktion vasemman puolen välikemerkki on kuitenkin pidettävä muistissa. (Aho et al. 2006, p. 250) Jäsennyksen alkaessa jäsentimen tila on pinon päällimmäisen arvon mukaisesti 0 ja syöte on vasemmanpuolimmaisimman tunnisteen mukaisesti `id`. Suoritettava operaatio valitaan tällöin jäsennystaulun ACTION-puolelta riviltä 0 ja sarakkeesta `id`. Seuraava operaatio on siis `s5` eli siirtyminen tilaan 5. Tämä tarkoittaa numeron 5 lisäämistä pinoon ja syötteen ensimmäisen tunnisteen poistamista. Seuraavalla rivillä pinoon on siis lisätty arvo 5, mikä vastaa symbolina olevaa tunnistetta `id`. (Aho et al. 2006, p. 252) Kuten jo todettiin, tätä symbolia ei todellisuudessa tarvita säilyttää, mutta se helpottaa jäsentämisprosessin havainnollistamista.

Seuraava operaatio määräytyy jälleen taulukon ACTION-puolen tilan 5 ja syötteen ensimmäisen tunnisteen `'*'` mukaisesti. Tästä jäsennystaulun solusta löytyy operaatio `r6`, eli reduktio produktiolla 6. Tämä siis vastaa `id`:n redusoimista `F`:ksi. Reduktion yhteydessä suoritettavia vaiheita on kolme: pinon päällimmäinen arvo poistetaan, reduktioon käytetty produktio annetaan ulostulona jäsennyspuun rakentamista varten, ja tehdään tilasiirtymä taulukon GOTO-puolen perusteella. Tilasiirtymä tehdään reduktioon käytetyn produktion vasemman puolen ja pinosta nyt löytyvän tilan perusteella. Rivillä 2 siis tapahtuvat seuraavat vaiheet:

1. Etsitään jäsennystaulun ACTION-puolelta suoritettava operaatio (reduktio) ja tämän jälkeen produktio, jota sen mukaan kuuluu käyttää redusointiin (`F → id`).
2. Poistetaan pinon päällimmäinen arvo, joka on 5. Tila on tämän jälkeen pinon mukaisesti 0.
3. Annetaan produktio ulostulona tai tallennetaan se jossain muodossa, jotta jäsennyspuu voidaan rakentaa.

4. Suoritetaan tilasiirtymä käytetyn produktion vasemman puolen (`F`) ja uuden tilan (0) perusteella. Tästä taulukon GOTO-puolen solusta löytyy arvo 3. Tämä arvo lisätään pinoon, eli jäsentimen uusi tila on 3.

(Aho et al. 2006, p. 252)

Myös seuraava jäsennyksen vaihe on reduktio, joten se suoritetaan vastaavalla prosessilla. Koska tila on nyt toinen, redusoiva produktio katsotaan toisesta taulukon ACTION-puolen solusta. Vastaavasti produktion vasemman puolen arvo on eri, joten taulukon GOTO-puolen osoittama tilasiirtymä katsotaan toisesta solusta. Jäsentäminen jatkuu kunnes päädytään jäsennystaulun `acc`-soluun, tai tyhjään soluun eli virheeseen.

LR-jäsentimiä on useaa tyyppiä ja kirjallisuudessa niistä tyypillisesti mainitaan LR(0), LR(1), SLR(1) ja LALR(1) (Wilhelm & Maurer 1995, pp. 356–364; Hunter 1999, p. 122; Aho et al. 2006, pp. 241–266; Su & Yan 2011, pp. 185–194). Hunter kehottaa valitsemaan LR-jäsennysmenetelmän kokeilemalla niitä järjestyksessä (1999, p. 122):

1. LR(0)
2. SLR(1) (simple LR)
3. LALR(1) (lookahead LR)
4. LR(1)

Ideana on tällöin löytää yksinkertaisin algoritmi, jolla jäsennys voidaan toteuttaa.

Kaikki LR-jäsentimet toimivat samalla periaatteella, mutta niiden jäsennystaulu on erilainen (Aho et al. 2006, p. 250). Usein käytännön sovelluksissa valitaan muistinkäytön takia LALR, koska sille luodut jäsennystaulut ovat huomattavan paljon pienempiä kuin LR:n. SLR:n jäsennystaulut ovat kooltaan lähellä LALR:ää, mutta SLR ei pysty kuvaamaan osaa tyypillisistä ohjelmointikielissä käytettävistä rakenteista. (Aho et al. 2006, p. 266)

5. VERTAILU

Tämän työn tavoitteena on suorittaa vertailu, joka selvittää top–down- ja bottom–up-jäsennysmenetelmien soveltuvuutta erilaisiin käyttötarkoituksiin. Tähän liittyy olennaisesti arvioida, miten työlästä jäsentimen toteutus on. Vertailu tarkastelee myös menetelmien virheidenkäsittelyä, koska se voi olla ohjelmoijille erittäin hyödyllinen työkalu.

5.1 Vertailuperusteet

Vertailuperusteita on neljältä osa-alueelta. Seuraavissa luvuissa käsitellään, miten osa-alueet liittyvät työn tavoitteeseen. Lisäksi määritellään, mitä ominaisuuksia osa-alueissa arvioidaan.

5.1.1 Vaatimukset kieliopille

Käytetty kielioppi liittyy vahvasti käännettävän ohjelmointikielen rakenteeseen. Syntaksianalyysia suoritettaessa se myös määrää jäsentämisen toimintalogiikkaa. Käännettävän ohjelmointikielen näkökulmasta olisi siis hyödyllistä, jos kielioppiin ei tarvitsisi tehdä muunnoksia. Erityisesti koska muunnettua kielioppia käytettäessä on riski, että ohjelmakielen semantiikka saattaa muuttua. Toisaalta jäsennys nojaa niin vahvasti kielioppiin, että jotkin jäsennysmenetelmät vaativat kieliopilta tiettyjä rajoitteita. Jäsentimen luomisen näkökulmasta olisi siis helpointa, jos kieliopille ei olisi mitään vaatimuksia.

1.a. Vaatimukset kieliopille:

> Tarkastellaan, onko jäsentäjällä joitakin vaatimuksia kieliopille. Jos on, tarkastellaan miten ehdottomia ne ovat ja arvioidaan karkeasti täyttääkö moni kielioppi ne alkuperäisessä muodossaan.

1.b. Kieliopin muuntaminen:

> Jos jäsentäjällä on joitakin vaatimuksia kieliopille, tarkastellaan, miten yksinkertaista on tehdä kielioppiin muutoksia, joilla vaatimukset saadaan täytettyä.

5.1.2 Suhde generointityökaluihin

Erilaisia generointityökaluja voidaan käyttää jäsentimien luonnin automatisoimiseen. Ne siis voivat helpottaa kääntäjän luomista. Toisaalta myös mahdollisuus luoda jäsennin täysin manuaalisesti, ilman generointityökaluja, saattaa olla hyödyllinen. On myös mahdollista, että tuettuja työkaluja on vain vähän tai niiden käyttöön liittyy joitain vaikeuksia.

2.a. Generointityökalujen tuki

Tarkastellaan, miten paljon ja minkälaisia menetelmää tukevia työkaluja on saatavilla.

2.b. Manuaalinen toteutus

Tarkastellaan, onko manuaalinen toteutus, eli jäsentimen kirjoittaminen mahdollista tai työmäärän suhteen realistista.

5.1.3 Toteuttamisen haasteellisuus

Jäsentimen toteutuksen haasteellisuus riippuu muiden vaatimuksien yhteisvaikutuksesta.

3.a. Muut vaatimukset ja edut

Tarkastellaan, onko menetelmällä joitakin vaatimuksia tai etuja, joita ei olla käsitelty muissa osa-alueissa.

3.b. Toteuttaminen

Arvioidaan yleisellä tasolla, miten monimutkainen prosessi menetelmää käyttävän jäsentimen luominen on, kun otetaan huomioon sen asettamat vaatimukset.

5.1.4 Virheidenkäsittely

Edistynyt virheenkäsittely on ohjelmoijien kannalta erittäin hyödyllinen kääntäjän ominaisuus. Koska työssä käsitellään syntaksianalyysia, virheidenkäsittelyä tarkastellaan vain syntaksivirheiden osalta. Osa virheistä saatetaan käsitellä esimerkiksi leksikaalisessa tai semanttisessa analyysissä.

4.a. Virheen paikannus

Tarkastellaan, onko virheen sijainnin tarkka paikannus mahdollista.

4.b. Virheen kuvaus

Tarkastellaan, onko mahdollista antaa käyttäjälle tarkka kuvaus virheestä, sen sijaan, että se esimerkiksi vain tunnistettaisiin syntaksivirheeksi. Tarkastellaan myös menetelmän mahdollisuuksia korjausehdotusten antamiseen.

5.2 Vertailun suoritus

Vertailu suoritettiin edellisessä alaluvussa määriteltyjen ominaisuuksien mukaisesti. Tehdyt havainnot löytyvät Taulukko 4:stä.

Taulukko 4: Jäsennysmenetelmien vertailu.

Ominaisuus	Top–down-menetelmät	Bottom–up-menetelmät
1.a. Vaatimukset kieliopille	• Top–down-jäsennysmenetelmät vaativat LL(1)-kieliopin (Grune et al. 2012, p. 126). • On mahdollista tarkistaa onko kielioppi LL(1), mutta ei suoraa tapaa tarkistaa onko jollekin ohjelmointikielelle olemassa LL(1)-kielioppi. • Ei voida tunnistaa onko kieliopille, joka ei ole LL(1), olemassa ekvivalentti LL(1)-kielioppi. • Ohjelmointikielten määrittelyssä annetut kieliopit ovat harvoin LL(1), joten muunnoksien tekeminen on usein pakollista, jos halutaan käyttää top–down-jäsennystä. (Hunter 1999, pp. 78–79)	• On mahdollista tarkistaa, onko kielioppi LR(*k*) (Hunter 1999, p. 122). • Kaikki ohjelmointikielet, jotka ovat LR(*k*) jollakin *k*:lla ovat myös LR(1). (Hunter 1999, p. 122) • Käytännössä kaikille kontekstivapaita kielioppeja tukeville ohjelmointikielille voidaan löytää LR-kielioppi (Hunter 1999; Aho et al. 2006, p. 242). • Koska LR-kieliopista on useita muunnelmia, voidaan kokeilla useampaa ja käyttää niistä sopivinta (Hunter 1999, p. 122). • LR-kieliopit ovat aina yksiselitteisiä, mutta joillakin shift–reduce-jäsennysmenetelmillä voidaan myös jäsentää tiettyjä moniselitteisiä kielioppeja. (Aho et al. 2006, p, 239)
1.b. Kieliopin muuntaminen	• Kieliopin muuntaminen on mahdollista yleensä, mutta ei aina (Hunter 1999, p. 93; Grune et al. 2012, p. 136). • Kielioppia muuntaessa ei välttämättä ole varmuutta, ovatko tehtävät muunnokset riittäviä (Hunter 1999, p. 93). • Kieliopin muunnoksien tekemiseen on olemassa työkaluja (Hunter 1999, p. 94).	• LR-jäsentimien vaatimat muunnokset ovat yleensä vähäisiä (Hunter 1999, p. 124). • Monet LR-jäsenningeneraattorit osaavat selvittää LR-konflikteja, mikä vähentää tarvetta tehdä muutoksia kielioppiin (Grune et al. 2012, p. 179).
2.a. Generointityökalujen tuki	• LL(1)-jäsenningeneraattoreita on olemassa kymmenittäin (Comparison of parser generators). • Lisäksi esimerkiksi LLgen (Grune et al. 2012, p. 148).	• Työkalut tukevat LR-jäsennystä hyvin (Hunter 1999, p. 124).

		• Erityisesti LALR(1)-jäsennin-generaattoreita on olemassa paljon. Myös muille LR-jäsennintyypeille löytyy generaattoreita. (Comparison of parser generators)
2.b. Manuaalinen toteutus	• RDP:n toteutus käsin on yksinkertaista (Hunter 1999, p. 80).	• Bottom–up-jäsentimen toteuttaminen käsin on liian työlästä (Aho et al. 2006, p. 243; Grune et al. 2012, pp. 158–159).
3.a. Muut vaatimukset ja edut	• LL(1):llä on pienemmät muistikäyttövaatimukset kuin LR(1):llä (Su & Yan 2011, pp. 203–204).	• LR-jäsennys on lineaariaikaista, eli suoritusaika on suoraan verrannollinen syötteen pituuteen (Hunter 1999, p. 124).
3.b Toteuttaminen	• Top–down-jäsentimen toteuttamisen pääasiallisena haasteena ovat kieliopin vaatimukset. • Käsin tehty toteutus voi olla hyvinkin yksinkertainen, mutta myös generointityökaluja on tarjolla paljon.	• Kielioppien tuki on hyvä, mutta toteutus on pakko tehdä generointityökalujen avulla.
4.a. Virheen paikannus	• Syntaksivirhe voidaan tunnistaa heti kun löydetään ensimmäinen sopimaton merkki. (Hunter 1999, p. 120).	• LR-jäsennin voi tunnistaa virheen heti kun se on edennyt syötteessä siihen asti (Hunter 1999, p. 124; Aho et al. 2006, p. 242).
4.b. Virheen kuvaus	• Ei merkittäviä tuloksia	• Ei merkittäviä tuloksia

Taulukko 4:n vertailussa huomataan kieliopin vaatimuksien olevan potentiaalisesti isokin ongelma top–down-jäsennyksessä. Harvan ohjelmointikielen määrittely täyttää niitä luonnostaan, ja kieliopin muuntaminen saattaa olla haastavaa. Bottom–up-jäsennys on siis näiltä osin joustavampi. Bottom–up tarjoaa myös valinnanvaraa eri tyyppisillä LR-jäsennyksen menetelmillä.

Top–down-jäsennyksen etuna on kuitenkin yksinkertaisempi kääntäjän rakenne, joka mahdollistaa myös jäsentimen toteuttamisen käsin. Bottom–up-jäsennys vaatii aina generointityökalujen käyttöä.

Generointityökalujen saatavuus on toisaalta erittäin hyvä molemmille. Bottom–up-menetelmistä erityisesti LALR(1):lle löytyy hyvin paljon tukea. Generointityökalujen tuen arvioimisen apuna käytettiin Wikipediaa, jonka soveltuvuus tieteelliseen kirjoittamiseen tyypillisesti kyseenalainen. Käytetty listamuotoinen artikkeli tarjoaa kuitenkin viittaushetkellä hyvän kokoelman erilaisia olemassa olevia työkaluja. Wikipediaa ei siis käytetty

minkäänlaisena tieteellisenä lähteenä laadullisen analyysin toteuttamiseen, vaan pelkästään työkalujen löytämiseen. Listalta löytyvien työkalujen laadun arviointi jää tämän tutkimuksen ulkopuolelle. (Comparison of parser generators)

Jäsennysmenetelmää valitessa varsin paljon painoa olisi siis käännettävän ohjelmointikielen luonteella. Jos käännettävä kieli on rakenteiltaan rajoittuneempi, tai jos jäsentämisen haasteet on otettu huomioon jo kieltä määritettäessä, saattaa top–down-jäsennys olla hyvä vaihtoehto jäsentimen yksinkertaisemman toteutuksen vuoksi. Kielen LL(1)-konfliktit eivät myöskään välttämättä ole mahdoton este top–down-jäsentimen toteuttamiselle. Kieliopin muuntamisen monimutkaisuus pitäisi luonnollisesti arvioida toteutuskohtaisesti.

Jos kielen muuntaminen LL(1)-yhteensopivaksi on niin monimutkaista, tai kieli on niin laaja, että se tekee muuntamisen haasteelliseksi, saattaa bottom–up-menetelmän toteutus olla sopivampi vaihtoehto. Jäsentimen toteutuksen yksinkertaisuus ei varmastikaan ole tärkein prioriteetti kaikissa toteutuksissa. Kielen semantiikan muuttumattomuus saattaa olla sen arvoista, että jäsentimen toteutus monimutkaistuu.

Syntaksivirheiden haasteena on jäsennysmenetelmästä riippumatta se, että jäsennin ei voi tietää mitä ohjelmoija on tarkoittanut. Esimerkiksi epätasapainossa olevat sulut voivat yhtä hyvin tarkoittaa puuttuvia sulkuja kuin ylimääräisiä sulkuja. (Wilhelm & Maurer 1995, p. 268; Grune et al. 2012, p. 121) Sekä LL(1)-, että LR(1)-jäsentimet tunnistavat ensimmäisen sopimattomassa paikasa olevan merkin, mutta koska syntaksivirheet ovat juuri kuvatulla tavalla monitulkintaisia, tämä ei välttämättä tarkoita todellisen virheen sijainnin tunnistamista.

Sekä top–down-jäsennyksen, (Wilhelm & Maurer 1995, pp. 326–339; Grune et al. 2012, pp. 143–148) että bottom–up-jäsennyksen virheidenkäsittelystä (Wilhelm & Maurer 1995, pp. 364–371; Grune et al. 2012, pp. 188–191) löytyy melko tarkkojakin kuvauksia, mutta nämä eivät juurikaan tarjonneet vertailukelpoista tietoa tutkittavien ominaisuuksien osalta. On mahdollista, että virheidenkäsittelyn laatu on niin toteutuskohtainen kysymys, ettei näin abstraktilla tasolla tehtävä vertailu tuota juurikaan tulosta.

6. YHTEENVETO

Työssä vertailtiin ohjelmointikielen kääntäjän jäsennysmenetelmien, eli top–down- ja bottom–up-jäsentimien eroja. Tutkimuskysymyksen tavoitteena oli selvittää, mihin sovelluskohteisiin vertaillut jäsennysmenetelmät sopisivat parhaiten.

Tuloksien perusteella minkäänlaisen tarkan vastauksen antaminen ei ole mahdollista, mikä oli työn teoreettisen luonteen vuoksi odotettavissa. Voidaan kuitenkin todeta, että top–down-jäsennys soveltuu yksinkertaisemman toteutuksensa vuoksi pienempiin projekteihin paremmin kuin bottom–up-jäsennys. Jos esimerkiksi tavoitteena olisi toteuttaa harjoitusmielessä jonkinlainen yksinkertainen kääntäjä, top–down-jäsennys soveltuisi sen syntaksianalyysin suorittamiseen hyvin.

Bottom–up-jäsennyksen etuna olevat vähäisemmät kieliopin vaatimukset tarkoittavat, että se tukee laajemmin erilaisia ohjelmointikielistä löytyviä rakenteita. Bottom–up-jäsennintä luodessa on todennäköisempää, että käytettävään ohjelmointikielen generoivaan kontekstittomaan kielioppiin ei tarvitse tehdä muutoksia. Tällöin on helpompi välttää ohjelmointikielen semantiikan muuttuminen. Voidaan siis arvioida, että bottom–up-jäsennys soveltuu käytettäväksi paremmin, kun tavoitteena on luoda kääntäjä laajemmalle tai monimutkaisemmalle ohjelmointikielelle.

Työn aiheessa olisi huomattava määrä potentiaalia syvällisemmälle vertailulle. Työssä tutustuttiin hyvin jäsennysmenetelmien teoriaan, mutta vertailun toteuttamisen kannalta tämän olisi pitänyt olla vieläkin kattavampaa. Vertailun suorittamisessa jouduttiin nojaamaan hyvin paljon lähdemateriaalissa jo tehtyihin huomioihin sen sijaan, että perusteluina olisi käytetty sekä lähteitä, että itse teoriaan perustuvia huomioita.

Generointityökalujen vertailu mahdollistaisi hyvin paljon käytännön tutkimusta omana aiheenaan. Esimerkiksi jonkin ohjelmointikielen jäsentämisen toteuttaminen muutamalla eri työkalulla muodostaisi helposti selkeän rakenteen tutkimukselle.

Myöskään vertailuun mukaan otetun virheidenkäsittelyn osalta ei löydetty selkeitä eroja kahden jäsennysmenetelmän välillä. Kuten generointityökalut, myös virheidenkäsittely mahdollistaisi hyvin paljon täysin sille omistettua tutkimusta. Aiheessa olisi paljon sekä teoreettisempia, että käytännönläheisempiä näkökulmia.

LÄHTEET

Aho, A.V., Sethi, R. & Ullman, J.D. (1986). Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading.

Aho, A.V., Lam, M.S., Sethi, R. & Ullman, J.D. (2006). Compilers: Principles Techniques and Tools (2nd Edition), 2nd ed. Pearson Education, Inc.

Comparison of parser generators, Wikipedia, verkkosivu. Saatavissa (viitattu 27.4.2018): https://en.wikipedia.org/wiki/Comparison_of_parser_generators#Deterministic_context-free_languages.

Grune, D., van Reeuwijk, K., Bal, H.E., Jacobs, C.J.H. & Langendoen, K. (2012). Modern Compiler Design, 2nd 2012 ed. Springer New York, New York.

Hunter, R. (1999). The Essence of Compilers, Prentice Hall, London.

Kaijanaho, A. Kontekstittomat kieliopit, TIEA241 Automaatit ja kieliopit , Jyväskylän yliopisto, Tietotekniikan laitos. Saatavissa (viitattu 27.1.2018): http://users.jyu.fi/~antkaij/opetus/auki/2016/matskut/II-cfg.pdf.

Su, Y. & Yan, S.Y. (2011). Principles of Compilers: A New Approach to Compilers Including the Algebraic Method, 1. Edition ed. Springer-Verlag, Berlin, Heidelberg.

Wilhelm, R. & Maurer, D. (1995). Compiler Design, Addison-Wesley, Wokingham.

LIITE A: ESIMERKKI RDP:STÄ

```
input → expression EoF
expression → term rest_expression
term → IDENTIFIER | parenthesized_expression
parenthesized_expression → ’(’ expression ’)’
rest_expression → ’+’ expression | ε
```

Ohjelma 1. *Kielioppi RDP-esimerkkiä varten (Grune et al. 2012, p. 123, Fig. 3.4).*

```
#include "tokennumbers.h"
/* PARSER */
int input(void) {
      return expression() && require(token(EoF));
}

int expression(void) {
      return term() && require(rest_expression());
}

int term(void) {
      return token(IDENTIFIER) || parenthesized_expression();
}

int parenthesized_expression(void) {
      return token(’ ( ’ ) && require(expression()) && require(token(’) ’ ));
}

int rest_expression(void) {
      return token(’+’) && require(expression()) || 1;
}

int token(int tk ) {
      if ( tk != Token.class) return 0;
      get_next_token(); return 1;
}

int require( int found) {
      if (! found) error ();
      return 1;
}
```

Ohjelma 2. *Esimerkki RDP:n toteutuksesta (Grune et al. 2012, p. 123, Fig. 3.5).*

```
#include "lex .h" /* for start_lex (), get_next_token(), Token */
/* DRIVER */
int main(void) {
      start_lex (); get_next_token();
      require(input ());
      return 0;
}
```

```
void error(void) {
      printf ("Error in expression\n"); exit (1);
}
```

Ohjelma 3. *RPD-esimerkin main- ja error-funktioiden toteutukset (Grune et al. 2012, p. 124, Fig. 3.6).*