Ohjelmakoodin tiivistäminen kielioppiin perustuvalla menetelmällä

Ilkka Salminen

Tampereen yliopisto
Tietojenkäsittelyoppi
Pro gradu –tutkielma
Kesäkuu 1998

Tampereen yliopisto
Tietojenkäsittelyopin laitos
SALMINEN ILKKA: Ohjelmakoodin tiivistäminen kielioppiin perustuvalla menetelmällä
pro gradu –tutkielma, 57 sivua
kesäkuu 1998

Tekstin tiivistäminen on perustunut perinteisesti siihen, miten hyvin voidaan arvata seuraavan merkin esiintyminen aikaisempien merkkien esiintymisten perusteella. Tekstin pohjalla on siis ollut mallina Markovin ketju, jossa merkit ovat riippuvia toisistaan.

Kielioppiin perustuvassa tiedon tiivistyksessä ei tutkita pelkästään yksittäisten merkkien esiintymistodennäköisyyksiä, vaan malliksi otetaan ohjelmakoodin kielioppi. Silloin tiivistäjällä on enemmän informaatiota tiivistettävästä kohteesta kuin pelkkiä merkkejä tutkittaessa. Näin ollen kielioppiin perustuvan tiivistäjän pitäisi pystyä tuottamaan tehokkaampaa tiivistystä, tosin yleisyyden kustannuksella. Kielioppiin perustuva tiivistäjä osaakin tiivistää vain oikein tuntemaansa kielioppia noudattavia ohjelmia.

Tiivistys tapahtuu kahdessa osassa. Ensiksi selaaja etsii lähdekoodista kaikki terminaalit, kuten muuttujien ja aliohjelmien nimet, ja tallentaa ne symbolitauluun. Seuraavaksi tiivistäjä muodostaa jäsennyspuun, jonka lehdissä on osoittimet symbolitaulun alkioihin. Lopulta sekä jäsennyspuu että symbolitaulu esitetään mahdollisimman tehokkaassa muodossa esimerkiksi jollain perinteisellä tiivistysmenetelmällä tiivistettynä.

Tutkimuksen tuloksena kävi ilmi, että kielioppiin perustuva tiivistys ei tuota kilpailukykyisiä tuloksia verrattuna perinteisiin tiivistäjiin. Se on myöskin hyvin hidas ja rajoittunut. Kielioppiin perustuva tiivistys voi kuitenkin toimia esimerkiksi verkossa jaettavien ohjelmien esitysmuotona samaan tapaan kuin Javan p-koodi.

Sisällys:

1. Johdanto ... 1

2. Yleistä tiivistämisestä ... 2

3. Entropia ... 4

4. Koodausmenetelmiä ... 6

4.1 Yksiselitteisesti purettavat koodit ... 6

4.2 Koodisanojen välien koodaus ... 7

5. Perinteiset tiivistysmenetelmät ... 9

5.1 Shannon-Fano ... 9

5.2 Huffmanin-koodit ... 10

5.3 Aritmeettinen koodaus ... 12

6. Staattiset ja mukautuvat mallit ... 14

6.1 Mallin merkitys tiedon tiivistämisessä ... 15

6.2 Mukautuva Huffmanin koodaus ... 16

6.3 Lempelin - Zivin koodaus ... 17

6.4 Mukautuva aritmeettinen koodaus ... 18

7. Kielioppiin perustuva ohjelmatiedoston tiivistys ... 20

7.1 Contlan mentelmä ... 20

7.2 Kieliopin ja jäsentelijän valinta Stonen mukaan ... 26

7.2.1 Johtokoodauksen valinta ... 27

7.2.2 Kieliopin valinta ... 27

7.2.3 Jäsennyskoodaus vai johtokoodaus ... 31

7.2.4 Jäsentelijän valinta ... 32

7.2.5 Kielen entropia ... 33

7.2.6 Stonen ehdotus koodausmenetelmäksi ... 35

7.3 Muotoilutietojen säilyttäminen tiivistyksessä ... 37

8. Kielioppiin pohjautuva informaatiomalli ... 39

9. Oman algoritmin kuvaus ... 41

10. Parannuksia perusalgoritmiin ... 46

10.1 Globaalit ja paikalliset johtonumerot ... 46

10.2 Moniselitteiset kieliopit ... 46

10.3 Mukautuvat kieliopit ... 47

10.4 Symbolitaulun organisoimisesta 48
11. Kokeellisia tuloksia 50
12. Omia tuloksia ja johtopäätelmiä... 53
Lähdeluettelo 56

1. Johdanto

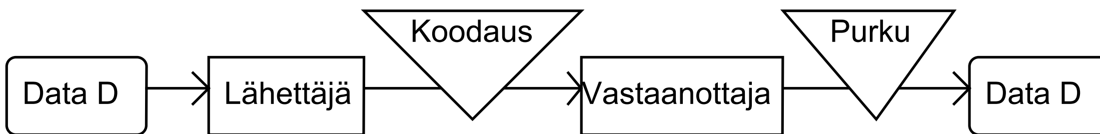
Tämän työn tarkoituksena on tutkia, voidaanko ohjelmakoodia tiivistää paremmin kuin perinteisillä merkkien todennäköisyyteen pohjautuvilla menetelmillä, käyttämällä hyväksi ohjelmointikielen kielioppia tiivistäjän mallinnuksen pohjalla. Tiivistäjä pyrkii löytämään tiivistettävästä lähteestä redundanttia informaatiota, joka poistamalla saadaan sama informaatio esitettyä pienimmällä mahdollisella määrällä bittejä. Perinteisissä tiivistysmenetelmissä redundanttia informaatiota on etsitty ainoastaan yksittäisistä merkeistä tai niiden jonoista. Kielioppiin perustuva tiivistys pyrkii ottamaan huomioon myös tiivistettävän tiedon rakenteen ja näin löytämään vielä paremman tehokkuuden. Haittapuolena tästä on se, että tiivistettävän tiedon on noudatettava tiivistäjän tuntemaa kielioppia melko täydellisesti eli ainoastaan kieliopillisesti oikeita lähdekoodeja voidaan tiivistää tällä menetelmällä. Perinteiset koodausmenetelmäthän ovat yleisiä eivätkä ne välitä koodattavasta materiaalista mitään.

Kielioppiin perustuvaa tiivistystä voidaan kuitenkin ajatella käytettäväksi esimerkiksi tietoverkkojen kautta levitettävissä ohjelmakoodeissa, kuten nykyisin jo Javan tavukoodia levitetään. Kieliopilla tiivistetystä koodista saadaan helposti poistettua kaikki muuttujien, aliohjelmien ja vastaavien nimet ilman, että ohjelman kääntäminen suoritettavaksi koodiksi vaarantuisi. Lopputuloksena on siis tiivistä ja jossain määrin suojattua koodia, jota voidaan levittää verkossa pienin kustannuksin.

Omissa kokeilluissani olen soveltanutkin näitä menetelmiä Java 1.1 lähdekoodien tiivistykseen. Tiivistysohjelma on pyritty tekemään mahdollisimman modulaariseksi käyttämällä kääntäjien ohjelmoimiseen tarkoitettuja perustyökaluja: LEX–yhteensopivaa selaaja Flex- ja YACC–yhteensopivaa jäsentäjäkääntäjää Bison. Näiden avulla on melko helppo muuntaa tai lisätä tiivistäjän ymmärtämiä kielioppeja. Kuten myöhemmin huomataan, tämä tapahtuu tiivistystehokkuuden kustannuksella.

2. Yleistä tiivistämisestä

Tiedon tiivistäminen tarkoittaa datan D koodaamista pienempään tilaan Δ(D) siten, että on mahdollista purkaa Δ(D) takaisin dataksi D tai joksikin sitä hyväksyttävän lähellä olevaksi arvioksi datasta D [12]. Kuviossa 2-1 on esitetty tiivistysjärjestelmän osat. Tiedon tiivistämisellä on kaksi yleistä sovellusaluetta: tiedon tiivistäminen tallentamista varten ja tiedon tiivistäminen sen lähettämistä varten.



Kuva 2-1 Tiivistysjärjestelmän osat.

Tallennettaessa dataa massamuistille, kuten levy- tai nauha-asemalle, saadaan tiivistettyä tietoa mahtumaan enemmän samaan tilaan kuin tiivistämätöntä tietoa. Tässä tapauksessa sekä lähettäjä ja vastaanottaja ovat samat. Haettaessa tietoa massamuistista se pitää kuitenkin purkaa ennen kuin se on valmista käytettäväksi.

Lähetettäessä tietoa modeemin välityksellä säästetään rahaa kun varsinainen tieto tiivistettynä vie vähemmän aikaa lähettää. Tällöin vastaanottajalla täytyy olla kyky purkaa viesti alkuperäiseen muotoon.

Silloin kun purettu data on täsmälleen samanlainen kuin tiivistetty data, kutsutaan tiivistysmenetelmää *hukkaamattomaksi tiivistysmenetelmäksi*, ja vastaavasti sellaista tiivistysmenetelmää, joka ei palauta täysin alkuperäistä dataa, kutsutaan *hukkaavaksi* menetelmäksi. Hukkaavia tiivistysmenetelmiä käytetään usein sellaisissa tilanteissa, joissa vaaditaan suurta tiivistyskykyä, mutta menetetyllä tiedolla ei ole suurta merkitystä. Tällaisia tilanteita ovat esimerkiksi valokuvien tallennus, jolloin esimerkiksi sellaisia värejä, joita ihminen ei pysty havaitsemaan, ei tarvitse säilyttää.

Kun sekä lähettäjä että vastaanottaja eivät näe koko dataa etukäteen, vaan lähettäjä koodaa dataa sitä mukaan kun se sitä saa sekä lähettää sen suoraan vastaanottajalle,

on kyseessä *on-line* koodaus. Vastaavasti *off-line* koodauksessa sekä lähettäjä että vastaanottaja käsittelevät dataa yhtenä kokonaisuutena.

Tiedon tiivistäminen riippuu siitä miten hyvin pystytään arvaamaan, mitä seuraavaksi on tulossa [1]. Jos aina pystytään arvaamaan tuleva viesti, saadaan täydellinen tiivistäminen eikä lähettäjän tarvitse lähettää mitään, koska vastaanottaja kykenee arvaamaan viestin. Esimerkiksi tavallisessa, puhutussa kielessä, ei tuottane ongelmia ymmärtää murtaen puhuvaa ulkomaalaista, koska hänen tekemistään virheistä tai puuttuvista sanoista huolimatta tarpeeksi informaatiota välittyy kuulijalle, jotta hän pystyy arvaamaan koko viestin sisällön.

3. Entropia

Intuitiivisesti on selvää, että viesteihin, joilla on suurempi todennäköisyys esiintyä, sisältyy vähemmän informaatiota kuin täysin satunnaisiin viesteihin. Jos viestin todennäköisyys on 1, ei sillä ole minkäänlaista informaatiosisältöä, koska vastaanottaja pystyy sen arvaamaan.

Tietoa tiivistettäessä tarkoituksena on poistaa siitä redundanssia ja jättää jäljelle vain tiedon informaatiosisältö [6]. Jotta voidaan täsmällisesti tutkia, kuinka paljon tietoa voidaan tiivistää, pitää informaatiosisältö pystyä määrittelemään. C. E. Shannon määritteli 1940-luvun loppupuolella entropian käsitteen, jolla hän tarkoitti juuri tätä tiedon informaatiosisältöä[12] .

Oletetaan joukko mahdollisia tapahtumia, joilla on esiintymistodennäköisyydet p_1, p_2, ..., p_n, joiden summa on 1. Koska todennäköisyyksien summa on 1, niin ainakin jokin tapahtumista tapahtuu. Nyt entropian $E(p_1, p_2, ..., p_n)$ täytyy noudattaa seuraavia ominaisuuksia:

- E on p_i:n jatkuva funktio.
- Jos jokainen tapahtuma on yhtä todennäköinen, täytyy E:n olla aidosti kasvava funktio n:n suhteen.
- Jos valinta tehdään useissa osissa, täytyy E:n olla osien entropioiden summa painotettuna jokaisen osan todennäköisyydellä.

Funktio, joka toteuttaa nämä ehdot, on

$$H_r(S) = \sum_{i=1}^{n} p_i \log_r\left(\frac{1}{p_i}\right),$$

jossa vakio r on logaritmin kantaluku. Normaalisti r on 2 eli yksikkönä on bitti ja logaritmit otetaan 2-kantaisina. Tällöin r voidaan jättää pois [1].

Olkoon S viesti aakkostossa Σ ja olkoon Γ viestin koodauksessa käytettävä aakkosto, jossa merkkien lukumäärä r on suurempi kuin yksi. Tällöin vaaditaan keskimäärin

$H_r(s)$ aakkoston Γ merkkiä jokaista Σ:n merkkiä kohden. Lisäksi jokaista reaalilukua $\varepsilon > 0$ kohti on olemassa koodausmenetelmä, joka käyttää keskimäärin $H_r(s) + \varepsilon$ aakkoston Γ merkkiä jokaista Σ:n merkkiä kohden. [12]

Edellä on oletettu, että kaikki tapahtumien todennäköisyydet ovat toisistaan riippumattomia, mutta käytännössä näin on hyvin harvoin. Esimerkiksi jos suomenkielisen viestin vastaanottaja on jo saanut kirjaimet "K", "i", "s" ja "s", on todennäköisyys, että seuraava kirjain on "a", huomattavasti suurempi kuin se oli viestin alussa.

Entropian käsitteen avulla kuitenkin voidaan ymmärtää, milloin ei enää voida saada aikaiseksi jo saavutettua suurempaa tiivistystä. Tiivistysmenetelmä onkin optimaalinen silloin, kun se saavuttaa asymptoottisesti entropian. Tästä optimaalisesta tuloksesta käytetään nimitystä *informaatioteoreettinen alaraja*. Lisäksi entropian käsitteestä voidaan johtaa seuraavat seikat:

- Satunnaista dataa ei voida tiivistää.
- Dataa, joka on jo tiivistetty optimaalisella tiivistysmenetelmällä, ei voida tiivistää enempää.
- Ei voida taata, että tiivistysohjelma saavuttaa mitään tiettyä tulosta kaikilla syötteillä.
- Yksitäisen viestin p_i informaatiosisältö saadaan kaavalla $-\log p_i$. Tästä käy hyvin ilmi, kuinka täysin varman viestin informaatiosisältö on 0.

4. Koodausmenetelmiä

Lähes kaikki tiivistysmenetelmät tarvitsevat jonkinlaisen tavan koodata merkkejä. Kaikkein yleisin tapa koodata on käyttää kiinteän pituisia koodisanoja. Siinä jokainen n-alkioisen joukon S alkio kuvataan $\lceil \log_k(n) \rceil$ pituiseksi merkkijonoksi. Tässä k on kohdemerkkijonon eri merkkien lukumäärä; tavallisesti kyseessä on binaariaakkosto, jolloin k on 2. Esimerkkinä tällaisesta koodauksesta on ASCII koodit. ASCII-koodiston 128 merkkiä kuvataan 8 bittisellä merkkijonolla, jossa eniten merkitsevä bitti on aina 0. Tosin usein myöskin 8. bitti on käytössä, jolloin voidaan esittää 256 eri merkkiä.

4.1 Yksiselitteisesti purettavat koodit

Funktion, joka toteuttaa merkkijonon koodauksen, täytyy olla injektio, kun halutaan hukkaamatonta tiivistystä. Se tarkoittaa, että funktio ei saa kuvata kahta eri merkkijonoa samalle koodisanalle. Yksiselitteisesti purettava koodi määritelläänkin seuraavasti. Olkoon f funktio $f:S \to \Sigma$. Merkkijono α, joka koostuu joukon Σ merkeistä on *yksiselitteisesti purettavissa* funktion f suhteen, jos on olemassa enintään yksi sellainen joukon S alkioista koostuva merkkijono L, että $f(L)= \alpha$. Funktio f on yksiselitteisesti purettavissa, jos kaikki merkkijonot, jotka koostuvat joukon Σ alkioista, voidaan yksikäsitteisesti purkaa funktion f suhteen. [12]

Tarkastellaan esimerkkinä tilannetta, jossa $S=\{a, b, c, d, e\}$, $\Sigma = \{0, 1\}$ ja f on seuraavanlainen:

f(a) = 00,

f(b) = 01,

f(c) = 10,

f(d) = 11,

f(e) = 100.

Funktion f soveltaminen merkkijonoon tapahtuu homomorfismina siten, että jokainen merkki kuvataan erikseen, jonka jälkeen kuvaukset konkatenoidaan. Nyt f(a,b,a,d)=000100100, joka on yksikäsitteisesti purettavissa funktion f suhteen. Koska mikään koodi ei ole yhden merkin pituinen, täytyy kaksi ensimmäistä bittiä (00) olla a. Seuraavaksi, samasta syystä kuin edelläkin, seuraavat kaksi bittiä (01) on oltava b. Seuraavat kaksi bittiä (00) ovat taas a, jonka jälkeen jää kolme bittiä. Koska yhtään yhden bitin koodia ei ole, täytyy lopuksi olla kolmen bitin koodisana eli 100, joka on koodisana merkille e.

Kaikki joukon Σ merkeistä koostuvat merkkijonot eivät ole kuitenkaan yksikäsitteisiä funktion f suhteen. Esimerkiksi f(cba) = f(ee) = 100100. Määritellään funktio f seuraavasti: f(a) = 01, f(b) = 001, f(c) = 0001, f(d) = 00001 ja f(d) = 000001. Nyt kaikki joukon Σ merkeistä koostuvat merkkijonot ovat yksikäsitteisesti purettavissa, koska tarvitsee ainoastaan lukea merkkien 0 määrä joukon S alkion selvillesaamiseksi.

4.2 Koodisanojen välien koodaus

Edellisestä esimerkistä käy ilmi, kuinka yksikäsitteisesti purettavat koodisanat voidaan muodostaa merkitsemällä koodisanojen välit erityisellä merkillä. Esimerkissä erottimena toimi merkki ‘1’.

Koodisanojen eteen voidaan lisätä sanan pituus, jolloin voidaan helposti saada selville itse koodisana lukemalla oikea määrä bittejä. Tästä syntyy uusi ongelma: miten erottaa koodisanan pituutta osoittava alku itse koodisanasta? Ongelma voidaan ratkaista kasautuvan pituuden tekniikalla, jossa koodisanan pituutta osoittavan osan eteen liitetään taas uutta pituutta osoittava koodi ja näin jatketaan, kunnes eteen lisätään enää vain yksi bitti. Koska aina tiedetään, että koodisana ja sen pituutta kuvaava osa ovat suurempia kuin 2, niin voidaan joka kerralla lisätä kaksi bittiä pienempää pituutta osoittava koodi. Lisäksi koodisanan loppuun lisätään vielä loppumerkki a_0. Jokainen koodisanan koostuu siten yhden bitin pituusmerkinnästä, pidemmästä pituusmerkinnästä, ..., itse koodisanasta ja loppumerkistä a_0.

Kasautuvan pituuden menetelmässä binäärimerkkijono, jonka pituus on n, voidaan muuttaa koodisanaksi, jonka pituus on

$$1+(n+1)+\lceil \log_k(n-1)\rceil+\lceil \log_k(\log_k(n-1)-2)\rceil+\cdots+1$$
$$=n+O(\log_k(n)).$$

Silloin kun aakkoston S koko on riittävän suuri, jäävät tämän menetelmän tarvitsemien bittien haittavaikutus vähäiseksi. Esimerkiksi 150 bitin merkkijono kuvautuu 164 bittiseksi eli se aiheuttaa n. 10% kasvun. Vastaavasti 2048 bittinen merkkijono kuvautuu 2068 bitiksi, joka on enää vain n. 1% kasvu. Pienillä merkkijonoilla tämä voi kuitenkin olla liikaa, jolloin on käytettävä jotain muita koodausmenetelmiä.

Toisenlainen tapa koodata on käyttää sellaisia koodeja, jotka eivät tarvitse erillistä merkkiä koodisanojen väliin. *Etuosa-koodi* määritellään siten, että koodi joukosta S joukolle Σ on etuosa-koodi, jos mikään koodisana ei ole toisen koodisanan etuosa [12]. Tällä tarkoitetaan sitä, että luettaessa koodia vasemmalta oikealle koodisana voidaan päätellä heti, kun se havaitaan. Tämän seikan vuoksi etuosa-koodi on myös yksiselitteisesti purettava koodi.

5. Perinteiset tiivistysmenetelmät

5.1 Shannon-Fano

Shannon ja Fano keksivät toisistaan tietämättä 1940-luvun loppupuolella tiivistysmenetelmän, joka perustuu lähetettävien viestien todennäköisyyksiin. Algoritmi on seuraavanlainen:

1. Järjestele kaikki mahdolliset viestit niiden esiintymistodennäköisyyksien mukaan laskevaan järjestykseen.
2. Jaa lista kahteen suurinpiirtein yhtäsuureen osaan.
3. Laita ensimmäisen ryhmän viestien koodisanan ensimmäiseksi merkiksi 1 ja toisen ryhmän 0.
4. Jatka rekursiivisesti kohdasta 1, kunnes jokaisessa ryhmässä on vain yksi viesti.

Esimerkiksi oletetaan viesteiksi a, b, c, d, e ja f, joiden vastaavat todennäköisyydet ovat 0.1, 0.3 0.1, 0.2, 0.1 ja 0.2. Lajiteltuna todennäköisyyksien mukaan saadaan viestit seuraavaan järjestykseen: b, d, f, a, c ja e. Ensimmäiseen ryhmään tulevat nyt viestit b ja d, joiden yhteenlaskettu todennäköisyys on 0.5. Toiseen ryhmään kuuluvat loput, joiden todennäköisyys on myöskin 0.5. Ensimmäinen ryhmän viestien koodisanat alkavat siis merkillä 0 ja toisen merkillä 1. Ensimmäinen ryhmä voidaan jakaa edelleen kahteen ryhmään, joista ensimmäinen on viesti b ja toinen on viesti d. Näitä ei voida jakaa enempää, joten viestin b Shannonin-Fanon koodisana on 00 ja viestin d koodisana vastaavasti 01. Viestit f, a, c, e voidaan taas jakaa kahteen ryhmään, joista ensimmäiseen tulevat viestit f ja a, ja toiseen ryhmään tulevat viestit c ja e. Nämä ryhmät voidaan jakaa vielä kertaalleen, jolloin lopuksi saadaan taulukon 5-1 koodisanat kaikille viesteille:

Viesti	Tn	Koodisana
a	0.1	101
b	0.3	00
c	0.1	110
d	0.2	01
e	0.1	111
f	0.2	100

Taulukko 5-1. Shannonin-Fanon koodisanat.

Kuten taulukosta 5-1 nähdään, saavat sellaiset viestit, joiden todennäköisyys on pieni, pidemmän koodisanan kuin ne viestit, joiden todennäköisyys on suuri. Esimerkiksi sanoma *abbfa* on Shannonin-Fanon koodeilla 1010000100101 eli 13 bittiä. Jos viesti esitettäisiin kiinteän pituisina koodisanoina, tarvittaisiin jokaiselle merkille kolme bittiä ja tällöin viestin pituus olisi 5 x 3 = 15 bittiä. Säästöä tulee siis kaksi bittiä, mutta esimerkissä merkki a esiintyy useammin kuin sille annettu todennäköisyys edellyttäisi. Jos koko sanoma koostuisi vain näistä harvinaisista merkeistä, ei säästöä syntyisi yhtään. Lisäksi kolmella bitillä voidaan esittää 7 eri merkkiä, jolloin tässä esimerkissä kiinteän mittaisessa koodisana esityksessä menee aina jonkin verran hukkaan.

Shannonin-Fanon koodit ovat etuosa-koodeja, mutta ne eivät ole optimaalisia entropian suhteen. Shannonin-Fanon koodien keskimääräinen pituus onkin puoliavoimella välillä [H, H+1), jossa H on lähdeviestin entropia [1].

5.2 Huffmanin-koodit

Shannonin artikkelin ilmestymisen jälkeen Huffman kehitti paremman algoritmin vuonna 1952. Huffmanin algoritmi toimii alhaalta ylöspäin, kun Shannonin-Fanon koodit puolestaan toimivat päinvastaiseen suuntaan. Hufmanin menetelmä voidaan esittää seuraavasti:

1. Listaa kaikki mahdolliset viestit todennäköisyyksineen.
2. Etsi kaksi viestiä, joilla on pienimmät todennäköisyydet.
3. Korvaa nämä kaksi viestiä yhdellä joukolla, johon kuuluvat molemmat viestit.
4. Toista kohtia 1-3, kunnes listassa on vain yksi viesti jäljellä.

Tästä muodostuu binaaripuu, jossa alkuperäiset viestit ovat lehdissä. Nyt voidaan muodostaa koodisana kullekin viestille seuraavasti:

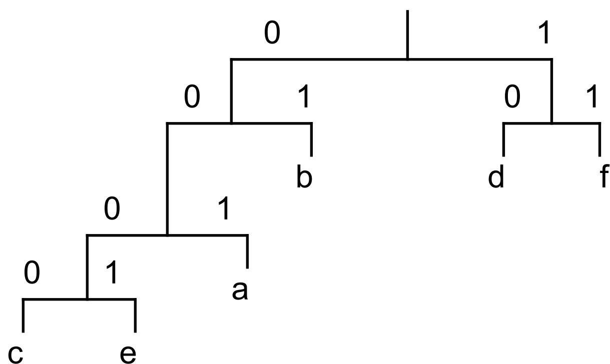
Kulje puu juuresta viestiin merkiten jokaista vasemman alipuun valintaa merkillä 0 ja jokaista oikean alipuun valintaa merkillä 1.

Taulukossa 5-2 on muodostettu Huffmanin koodit samoille viesteille kuin Shannonin-Fanon esimerkissäkin. Jokaisessa vaiheessa valitut viestit on merkitty varjostuksella.

1. kierros	**2. kierros**	**3. kierros**	**4. kierros**	**5. kierros**	**Loppu**
b 0,3	**b** 0,3	**b** 0,3	**{d, f}** 0,4	***{{{c, e}, a}, b}*** *0,6*	**{{{{c, e}, a}, b}, {d, f}}** 1,0
d 0,2	**d** 0,2	**{{c, e}, a}** 0,3	***b*** *0,3*	***{d, f}*** *0,4*	
f 0,2	**f** 0,2	***d*** *0,2*	***{{c, e}, a}*** *0,3*		
a 0,1	***{c, e}*** *0,2*	***f*** *0,2*			
c *0,1*	***a*** *0,1*				
e *0,1*					

Taulukko 5-2 Huffmanin koodit.

Viimeisen vaiheen tulosta {{{{c, e}, a}, b}, {d, f}} vastaa kuvan 5-1 binaaripuuta.



Kuva 5-1 Huffmannin koodipuu.

Puusta saadaan taulukon 5-3 koodisanat viesteille:

Viesti	Tn	Koodisana
a	0,1	001
b	0,3	01
c	0,1	0000
d	0,2	10
e	0,1	001
f	0,2	11

Taulukko 5-3 Huffmanin koodisanat.

Huffmanin koodeilla sanoma *abbfa* on 001010111001 eli 12 bittiä, joka on yksi bitti vähemmän kuin Shannonin-Fanon koodeilla.

Myöskin Huffmanin koodit ovat etuosa-koodeja, mutta toisin kuin Shannonin-Fanon koodit, Huffmanin koodit ovat optimaalisia siinä mielessä, että koodisanojen keskimääräinen pituus on pienin mahdollinen annetuilla viestien todennäköisyyksillä. Huffmanin koodit kuitenkin vaativat kaikkien mahdollisten viestien luetteloimista, mikä ei tietystikään ole käytännössä mahdollista. Sen sijaan Huffmanin koodit muodostetaan viestin yksittäisille merkeille tai tietyn pituisille osille. Tällöin saavutetaan pienin mahdollinen redundanssi näille osille, mutta ei koko viestille.

5.3 Aritmeettinen koodaus

Samoin kuin Huffmanin koodit, on aritmeettinen koodaus informaatioteoreettiselta kannalta optimaalinen. Aritmeettinen koodaus ei kuitenkaan hukkaa bittejä siinä, että koodisana täytyy koostua kokonaisista biteistä, kuten Huffmanin koodeissa.

Aritmeettisessa koodauksessa viesti esitetään puoliavoimella välillä [0, 1). Viestin kasvaessa tätä väliä pienennetään, jolloin sen esittämiseen tarvittavien bittien määrä kasvaa. Väliä pienennetään sitä enemmän, mitä epätodennäköisempi on lähetettävä symboli.

Ennen kuin yhtään symbolia on käsitelty, viestiä kuvaava väli on koko väli [0,1). Jokaisen viestin symbolin kohdalla tätä väliä pienennetään käsiteltävän symbolin todennäköisyyden verran. Oletetaan, että lähdeviestin aakkostona ovat edellisten

esimerkkien merkit a, b, c, d, e ja f samoilla todennäköisyyksillä: 0.1, 0.3 0.1, 0.2, 0.1 ja 0.2. Tällöin näille merkeille annetaan välit [0, 0.1), [0.1, 0.4), [0.4, 0.5), [0.5, 0.7), [0.7, 0.8) ja [0.8, 1). Oletetaan myös, että lähetettävänä viestinä on *abbfa*. Tällöin ensimmäinen merkki a pienentää välin [0, 1) väliksi [0, 0.1). Uuden välin alaraja saadaan laskettua kaavalla

$$L_{i+1} = L_i + (H_i – L_i)*LM_{i+1},$$

missä L_{i+1} uusi alaraja, L_i on vanha alaraja, H_i on vanha yläraja ja LM_{i+1} on uuden merkin alaraja. Vastaavasti yläraja lasketaan kaavalla

$$H_{i+1} = L_i + (H_i – L_i)*HM_{i+1},$$

missä HM_{i+1} on uuden merkin yläraja. Seuraava merkki b pienentää tätä uutta väliä väliksi [0.01, 0.04). Seuraavaksi merkki b tuottaa uuden välin [0.013, 0.022). Merkki f muuttaa tämän väliksi [0.0202, 0.022). Lopuksi merkki a tuottaa välin [0.0202, 0.02038). Pisteen kuvaamiseen tältä väliltä vaaditaan 11 bittiä eli yksi vähemmän kuin Huffmanin koodeilla. Yleisesti ottaen aritmeettinen koodaus ei kuitenkaan aina ole parempi kuin Huffmanin koodit, vaan niitä voidaan pitää yhtä hyvinä.

Koodin purkaja näkee heti välistä, että ensimmäinen merkki täytyy olla a, koska lähetetty väli on merkin a välin sisällä. Seuraavaksi purkaja tutkii, mitä koodaaja on tehnyt eli pienentää välin [0, 1) väliksi [0, 0.1). Nyt purkaja voi päätellä seuraavan merkin olevan b, sillä se on ainoa merkki, joka pienentää väliä siten, että uusi väli sisältää kokonaan viestin välin [0.0142, 0.01468). Näin jatkamalla pystyy purkaja muodostamaan alkuperäisen viestin.

Purkajan ei tarvitse välttämättä tietää kumpaakin välin päätepistettä, vaan mikä tahansa numero lopulliselta väliltä riittää. Tällöin purkajan täytyy tietää milloin viesti loppuu. Esimerkiksi yksi numero, kuten 0, voi merkitä mitä tahansa viesteistä: a, aa, aaa, aaaa..... Tällaisessa tapauksessa viestin loppuun on liitettävä jokin viestin loppumisesta kertova merkki, jolloin purkaja tietää lopettaa toimintansa. Tämän vuoksi aikaisemmin esitelty esimerkki tarvitsi siis lisää bittejä tämän loppumerkin esittämiseen.

6. Staattiset ja mukautuvat mallit

Aikaisemmin esitetyt tiivistysmenetelmät vaativat toimiakseen viestien tai lähdeaakkoston todennäköisyydet etukäteen, eivätkä merkkien todennäköisyydet muutu kesken tiivistysprosessin. Siksi näitä tiivistysmenetelmiä kutsutaankin *staattisiksi*. Käytännössä tällainen vaatimus on mahdoton toteuttaa. Todennäköisyydet voivat lisäksi vaihdella eri viestien välillä, jolloin jollekin tietylle viestille todennäköisyydet tuottaisivat hyvän tiivistyksen, mutta jollekin toiselle huonon. Tämä voitaisiin korjata siten, että ennen viestin käsittelemistä koodaaja tutkisi viestin eri aakkosten todennäköisyydet ja vasta sen jälkeen alkaisi koodauksen. Tällaisen menetelmän heikkona puolena on se, että se ei ole enää on-line algoritmi, vaan sen olisi saatava selville koko viesti ennen kuin viesti voidaan käsitellä.

Mukautuva malli on sellainen, joka nimensä mukaisesti mukautuu viestin käsittelyn edetessä. Yksinkertaisimmillaan mukautuva menetelmä toimii siten, että koodaaja lähtee oletuksesta, että jokainen merkki on yhtä todennäköinen. Sitten se jokaisen merkin jälkeen kasvattaa juuri luetun merkin todennäköisyyttä. Vastaavasti purkaja pystyy muuttamaan jokaisen merkin käsittelyn jälkeen omia todennäköisyyksiään samalla lailla. Näin sekä koodaajalla että purkajalla on aina samat todennäköisyydet jokaiselle merkille.

Tällainen todennäköisyyksien muokkaaminen viestin käsittelyn kuluessa tuo esille seuraavan ongelman. Mitä tehdä merkeille joiden todennäköisyys on 0? Tällaiselle merkille pitäisi tulla koodisana, jonka pituus on -log 0, eli äärettömän monta bittiä. Mahdottomille tapauksille siis pitää antaa jonkin hyvin pieni todennäköisyys, mutta ei ole olemassa mitään yksikäsitteistä tapaa laskea, mikä se olisi. [1]

Mukautuva menetelmä voi olla joissakin tapauksissa hieman huonompi kuin staattinen, mutta staattisen menetelmän kannalta huonoissa tilanteissa se voi olla huomattavasti parempi. Jos staattisen menetelmän malli on täysin väärä käsiteltävälle viestille, voi tuloksena olla hyvinkin paljon alkuperäistä viestiä suurempi tulos [1].

6.1 Mallin merkitys tiedon tiivistämisessä

Tiedon tiivistys voidaan jakaa kahteen toiminnalliseen osaan: *malliin* ja *koodausyksikköön*. Näistä malli on tiivistyksen kannalta oleellisempi ja siten mielenkiintoisempi. Malli kuvaa tiedon rakennetta eli sitä, miten yksittäiset merkit eli tiivistettävät elementit liittyvät toisiinsa ja miten tieto on jäsennelty. Perinteisessä tekstin koodauksessa mallit perustuvat yksittäisten merkkien tutkimiseen. Tiivistettävä data kuvataan esimerkiksi yksittäisten kirjainten tai tavujen todennäköisyyksillä. Kyseessä on siis Markovin ketju, jossa yksittäisten merkkien todennäköisyydet riippuvat ainoastaan edellisten merkkien esiintymisestä syötteessä ja ovat muutoin toisistaan riippumattomia. Myös muunlaisia malleja voidaan käyttää. Etenkin kuvien tiivistyksessä yleisesti käytetty häviöllinen *JPEG* (Joint Photographic Experiment Group) -tiivistys käyttää mallinaan kaksiulotteista Fourier-muunnosta. Tässä menetelmässä malli jättää myös pois sellaista informaatiota, jota ei pidetä kokonaisuuden kannalta tarpeellisena saaden näinollen tiivistystä vieläkin tehokkaammaksi kuvan laadun kustannuksella.

Mallin tehtävänä on antaa varsinaiselle koodausyksikölle todennäköisyydet, joilla merkki koodataan. Näin ollen tehokkaan tiivistämisen edellytys on tiivistettävää tietoa mahdollisimman hyvin kuvaava malli. Jos malli pystyy tuottamaan jokaisen merkin absoluuttisen todennäköisyyden eli sen todennäköisyyden, jolla koodattava merkki esiintyy koko tiivistettävässä dokumentissa, päästään tiivistyksessä hyvin lähelle entropiaa. Vastaavasti huono malli tuottaa vääriä todennäköisyyksiä, jolloin koodausyksikkö antaa näille merkeille liian pitkiä koodisanoja ja tiivistys heikkenee. Staattisissa koodausmenetelmissä mallin osuus on hyvin vähäinen, oletetaanhan niissä, että merkkien todennäköisyydet tunnetaan jo edeltäkäsin. Mukautuvissa menetelmissä juuri malli kertoo sen, kuinka yksittäisten merkkien todennäköisyys muuttuu.

Tiivistysohjelmassa malli sisältyy siihen algoritmiin, jolla tiivistys toteutetaan. Mitä enemmän malli itsessään sisältää informaatiota tiivistettävästä datasta, sitä vähemmän täytyy lähteen kertoa tietoa vastaanottajalle. Onhan jo mallin kertomisella välitetty jo ennen tiivistetyn tiedon lähettämistä jotain myöhemmin tulevasta tiedosta.

Näin ollen sellaiset mallit, jotka sisältävät paljon tietoa tiedosta eli jonkinlaista metatietoa, pitäisi toimia tehokkaammin kuin sellaiset mallit, jotka eivät sisällä mitään oletusta tiivistettävästä tiedosta. Huonona puolena on tietysti se, että malli rajoittaa tiivistysalgoritmin käyttöä vain sellaiselle datalle, josta sillä on tietoa.

Koodausyksikön tehtävänä on tuottaa merkeille mahdollisimman lyhyet koodisanat annetuilla todennäköisyyksillä. Riippuen mallin ja koodausyksikön välisestä työnjaosta voi koodausyksikön tehtäväksi jäädä esimerkiksi vain koodisanojen esittäminen mahdollisimman tehokkaasti. Tällöin malli antaa todennäköisyyksien lisäksi myös koodisanat.

6.2 Mukautuva Huffmanin koodaus

Huffmanin koodaukselle kehitettiin 1978 mukautuva algoritmi. Siinä koodipuun jokaisessa solmussa, myöskin sisäsolmuissa, on tallennettuna merkkien esiintymismäärät. Esimerkiksi kuvan 5-1 puun solmuissa c ja e on näiden merkkien esiintymismäärät tähän mennessä. Näiden solmujen vanhemmassa on tallennettuna niiden summa. Seuraavan kerran kun c luetaan, täytyy c solmun lukumäärää kasvattaa yhdellä. Tämä seurauksena voidaan joutua muuttamaan puun rakennetta siten, että se vastaa uutta tilannetta. Jos merkki c esiintyy nyt tiheämmin kuin merkki e, täytyy näiden solmujen paikkaa vaihtaa. Sitten täytyy tietysti päivittää myös solmujen vanhemmat ja isovanhemmat aina juureen saakka. Yleisesti koko puun muoto voi muuttua prosessin edetessä.

Kasvatettaessa jokaisessa solmussa esiintymismääriä tulee ottaa huomioon myös ylivuototilanteet eli tilanteet, joissa esiintymismääriä ei pystytä enää pitämään yllä käytettävän kokonaislukujen esitystarkkuuden rajoissa. Yleisesti tällainen tilanne voidaan ratkaista skaalaamalla esiintymistiheydet jollain sopivalla luvulla. Tällöin puun rakenne ei muutu, koska solmujen suhteet eivät muutu. Tässäkin tapauksessa täytyy tosin ottaa huomioon mahdolliset osamäärät ja pyöristykset, jolloin voidaan joutua muuttamaan myös puun rakennetta, mutta todennäköisesti ei kovin paljoa. [1]

6.3 Lempelin - Zivin koodaus

Lempelin - Zivin koodaus on yksi kaikkein eniten käytetyimmistä koodausmenetelmistä. Sitä käytetään muun muassa suosituissa *zip*-ohjelmistoissa. Lempelin - Zivin -koodaus eroaa aiemmin esitetyistä menetelmistä siten, että siinä ei koodata yksittäisiä merkkejä. Lempelin - Zivin -koodaus muodostaa lähteen jäsentelyn aikana lähdeviestien joukon, joille määritellään koodisanat.

Algoritmi etsii lähteestä sanoja, osamerkkijonoja, jotka koostuvat äärellisestä lähdeaakkostosta. Nämä sanat eivät saa pituudeltaan ylittää etukäteen määrättyä vakiota L_1. Koodausmenetelmä koodaa nämä sanat yksikäsitteisesti purettaviksi koodisanoiksi. Osamerkkijonot määritellään siten, että niillä kaikilla on lähes yhtäsuuri todennäköisyys esiintyä lähteessä. Lopputuloksena on, että usein esiintyvät symbolit ovat pidemmissä osamerkkijonoissa kuin harvemmin esiintyvät lähdeaakkoston merkit. [6]

Lempelin - Zivin -algorimissa on oppiva järjestelmä, joka mukautuu lähdekoodin ominaisuuksiin. Mukautuminen perustuu taulukkoon, jossa on jo löydettyjä osamerkkijonoja. Taulukkoon lisätään uusia sanoja siten, että kun uusi sana tulee lähteeltä, laitetaan se taulukkoon muodossa (*i, c*), missä *i* on jo taulukossa oleva sana ja *c* on uusi merkki.

Esimerkiksi katsotaan, miten muillakin algoritmeilla tiivistetty merkkijono *abbfa*, tiivistyy Lempelin - Zivin -koodauksella. Aluksi taulu on tässä esimerkissä tyhjä, vaikka se voidaan tietysti jo etukäteen täyttää lähdeaakkoston merkeillä eli yksimerkkisillä sanoilla.

Seuraavaksi luemme merkin *a* ja lisäämme sen taulukkoon. Koodisanaksi se saa (0,a) Samoin lisäämme merkin *b*, koska aikaisemmin ei ole ollut tällä merkillä alkavaa sanaa. Tämän osamerkkijonon koodisanaksi tulee (0, *b*).

Seuraavaksi voimme käyttää taulukossa jo olevaa sanaa b ja lisätä siihen merkin f. Uusi lisättävä sana on siis *bf*, ja se saa koodiksi (2, f). Lopuksi merkki a on jo taulukossa, joten mitään ei tarvitse lisätä. Taulukko 6-1 kuvaa lopputilannetta.

Sana	Koodi	Koodi bitteinä
a	(0, a)	0000
b	(0, b)	0001
bf	(2, f)	1111

Taulukko 6-1 Lempelin - Zivin koodit.

Lempelin - Zivin algoritmissa koodisanat muodostetaan yksinkertaisesti ahneella menetelmällä etsimällä lähteestä pisin merkkijono, joka on jo olemassa taulukossa ja lisäämällä siihen yksi merkki. Tällainen menetelmä on hyvin tehoton pienillä esimerkeillä kuten yllä. Esimerkiksi jos oletamme, että lähdeaakkosto koostuu enintään neljästä merkistä, jolloin voimme koodata nämä merkit 2 bitillä. Lisäksi voimme tässä tapauksessa rajoittaa taulukon maksimikoon vaikka sopivasti kolmeen sanaan, joiden indeksit voidaan myöskin kertoa 2 bitillä. Indeksi 0 merkitsee, että merkkiä ei ole aikaisemmin ollut taulukossa. Näin ollen merkkijonon abbfa tiivistetyksi pituudeksi tulee 4x4 bittiä eli yhteensä 16 bittiä. Siis selvästi heikoin tulos verrattuna aikaisempiin algoritmeihin. Lempelin – Zivin menetelmä on kuitenkin paljon käytetty ja hyväksi havaittu tiivistysmenetelmä, joka onkin asymptoottisesti optimaalinen. Se kuitenkin tarvitsee melko pitkiä lähteitä ennen kuin tiivistys on tehokasta. Lisäksi Lempelin - Zivin-koodaus voi menettää mukautuvat ominaisuutensa, jos lähteen ominaisuudet vaihtelevat aikaa myöten. Tällöin se voi jäädä kiinni vanhoin ominaisuuksiin eikä enää mukaudu muutoksiin lähteessä. [6]

Koska algoritmi koodaa jokaisen merkin lisäksi myös indeksin aikaisempaan merkkijonoon, kasvaa koodi aluksi huomattavasti, mahdollisesti jopa 50%. Tätä voidaan lieventää esimerkiksi tallentamalla jokainen lähdeaakkoston merkki valmiiksi taulukkoon.

6.4 Mukautuva aritmeettinen koodaus

Aritmeettisessa koodauksessa täytyy pitää yllä jokaisen merkin esiintymismääriä sekä kokonaismäärää, jonka avulla voidaan laskea merkkien todennäköisyydet. Algoritmi käyttää myös kumulatiivisia todennäköisyyksiä, joiden laskeminen saattaa olla työlästä, jolloin ne kannattaa pitää tallessa erillisessä taulukossa. Joka kerta kun mallia muutetaan, lajitellaan merkit uudestaan oikeaan järjestykseen. Tämä voidaan

tehdä tehokkaasti siten, että pidetään yllä indeksiä, jolla tiedetään missä kunkin merkin lukumäärä on. Kun merkin lukumäärää kasvatetaan, vaihdetaan sen indeksiä sen merkin kanssa, joka on sillä paikalla, johon merkki kuuluu. Myös kaikki merkin yläpuolella olevien merkkien lukumääriä täytyy kasvattaa, jotta kumulatiivinen summa säilyisi. Silloin kun merkkejä on paljon, kuten binaaridatassa, joka koostuu 256 merkistä, hidastuu tämä algoritmi huomattavasti. Siihen onkin kehitetty parannettuja menetelmiä.[1]

7. Kielioppiin perustuva ohjelmatiedoston tiivistys

Tietoa saadaan tiivistettyä sitä tehokkaammin, mitä korkeamman tason malleja käytetään tiivistyksessä. Tällöin arvaus siitä, mikä seuraava tiivistettävän lähteen merkki tulee olemaan, on yleensä parempi. Perinteiset tiivistysmenetelmät perustuvat siihen, että koodattavat merkit ovat joko kokonaan toisistaan riippumattomia tai riippuvat edellisistä merkeistä jokin kiinteän etäisyyden verran. Tällaiset menetelmät eivät ota siis huomioon lähteessä mahdollisesti olevia sisäisiä rakenteita kuten kielioppia. Tavallisessa puhutussa kielessä kielioppisäännöt ovatkin liian monimutkaisia ja usein jopa ristiriitaisia, jotta niitä voitaisiin käyttää hyväksi tiivistyksessä. Kieliopillisesti oikeissa ohjelmakoodeissa kielioppi on kuitenkin yksikäsitteisesti määritelty ja säännöt helposti käytettävissä. [9]

Kielioppiin perustuvissa tiivistysmenetelmissä lähdekoodi esitetään jäsennyspuuna ja lähes kaikissa ohjelmakoodeissa on, tai ainakin pitäisi olla, kommenteilla ja muotoi-lulla selkeytetty ohjelman kulkua ja kerrottu lukijalle, miten ohjelma toimii. Siksi pelkkään kielioppiin perustuva tiivistys olisi hukkaava menetelmä ja tiivistyksen purun tuottama toimiva koodi olisi käytännössä hyödytöntä. Muotoilu voidaan kyllä palauttaa ns. pretty printer -ohjelmilla, jolloin muotoilusäännöt ovat tavallaan tiivistettynä ko. ohjelman sääntöihin. Tulos vastaa kuitenkin hyvin harvoin täysin alkuperäistä koodia.

7.1 Contlan mentelmä

Contla [3] esittää artikkelissaan perusalgoritmin sille, miten kieliopillisesti oikea lähde koodi voidaan tiivistää kielioppiin perustuen. Tiivistys tapahtuu tallentamalla tehokkaasti jäsentelijän polku eli vain ne kieliopin säännöt, jotka jäsentelijä joutuu käymään läpi.

Seuraavaksi käydään läpi Contlan esittelemä algoritmi lähdekoodin tiivistämiseksi. Tarkastellaan aluksi Pascal-ohjelman bisect alkua

```
PROGRAM bisect(input, output);
CONST eps = 1e-14;
VAR x,y: REAL;.
```

Tämä koodinpätkä voitaisiin koodata koodisanoilla: 00001, 1, 00010, 1, 00011, 0, 1, 0, 1, 1, 00100, 1, 011, 1, 00101, 0, 1, 1, 10, 00110, 0, 0, 1. Tässä yhteydessä koodisanojen merkitys olisi taulukon 7-1 mukainen.

Koodisana	Merkitys
00001	tunnus bisect.
1	apumerkki <IDENTIFIER>
00010	tunnus input
1	apumerkki {, tunnus}
00011	tunnus output
0	pilkuilla erotetun jonon loppu, jota
1	apumerkki <BLOCK>
0	apumerkki <LABEL DECN PT>
1	apumerkki <CONST DFN PT>
1	apumerkki <CONST
00100	tunnus eps
1	apumerkki <CONSTANT>
011	kolmas vaihtoehto
1	apumerkki <UNSIGN NUMP>
00101	tunnus 1
0	apumerkki <FRACT PT>
1	apumerkki <EXPONENT>
1	apumerkki <SIGN>
10	toinen vaihtoehto
001100	tunnus 14
0	apumerkki <SEQ CONST DEFN>
0	apumerkki <TYPE DEFN PT>
1	apumerkki <VARIABLE DFN PT>

Taulukko 7-1 Koodisanat bisect-ohjelmassa.

Taulukon 7-1 koodisanat on muodostettu tallentamalla jäsentelijän kulkua. Pascal-kielen kieliopissa voisi ensimmäinen BNF–sääntö olla

<PROGRAM> ::= <PROGRAM> tunnus (<IDENTIFIERS>); <BLOCK>.

Nyt jäsentelijän kulusta on tallennettu aluksi loppumerkin PROGRAM hyväksyvä sääntö, sitten käyttäjän terminaalin tunnus (bisect) hyväksyvä sääntö jne. Kokonaisuudessaan listasta tulisi seuraavanlainen:

1. 'PROGRAM'
2. tunnus
3. avaava sulku
4. <IDENTIFIERS>

5. sulkeva sulku
6. puolipiste
7. <BLOCK>.

Rivit 4 ja 7 kuvaavat jäsentelijän polkua silloin kun se kohtaa apumerkin. Ensimmäinen apumerkki rivillä neljä on määritelty seuraavasti:

<IDENTIFIERS> ::= tunnus {, tunnus}.

Ohjelmassa bisect on kaksi tunnusta input ja output, joten neljännestä rivistä tulee laajennettuna

4. tunnus

4.1 {, tunnus}.

Rivi 4 käsittelee tapauksen, jossa on vain yksi tunnus ja rivi 4.1 ne tapaukset, joissa on tunnuksia useampia pilkuilla eroteltuna. Edelleen rivi 4.1 jakaantuu kahteen tapaukseen eli siihen että selaaja löytää pilkun (,) tai tunnuksen, siis

4.1 pilkku

4.2 tunnus.

Toistaiseksi kielioppi vaatii, että vähintään yksi apumerkki <IDENTIFIER> on olemassa lähdekoodissa. Näin ei kuitenkaan aina ole, joten sääntö numero neljä täytyy muokata vielä lisää. Koko sääntö on nyt:

4. Apumerkki, joka muodostuu

4.1 tunnuksesta,

4.2 apumerkistä, joka muodostuu jonosta

4.3 pilkkuja

4.4 tunnuksia ja

4.5 apumerkki loppuu.

Contlan algoritmi muodostaa listan kaikista niistä kieliopin merkeistä, jotka jäsentelijä löytää jäsennellessään lähdekoodia. Esimerkiksi jos aikaisemmin esitetyn bisect-ohjelman ensimmäinen rivi olisikin muotoa PROGRAM bisect(input);, niin tuloksena olisi lista

1. PROGRAM,
2. tunnus (bisect),
3. avaava sulku,
4. tunnus (input),
5. ohita pilkuilla erotettujen tunnusten lista (säännöt 4.2-4.4),
6. sulkeva sulku ja
7. puolipiste.

Lista kuljetuista säännöistä muodostetaan siten, että jos sääntö on valittu, merkitään sitä numerolla 1, muuten numerolla 0. Listaa voidaan vielä yksinkertaistaa seuraavilla säännöillä:

1) Lähdekoodina on ainoastaan kieliopillisesti oikeita ohjelmia, joten terminaalimerkkejä ei ole mukana listassa. Esimerkiksi jokainen Pascal-ohjelma alkaa varatulla sanalla PROGRAM. Siksi esimerkiksi bisect-ohjelman kohdalla voidaan ohittaa säännöt 1, 2, 3 ja 5.

2) Kieliopilliset tilanteet sarja, vaihtoehto, jono tai valinnainen merkitään numerolla 1, jos jäsentelijä on käynyt sen läpi, muuten numerolla 0.

   a) Sarja on useamman säännön liitos, siten jos se on merkitty numerolla 1, kaikki siihen kuuluvat säännöt on käyty lävitse.

   b) Vaihtoehtotilanteissa numero 1 merkitsee sitä, että jokin vaihtoehto on valittu. Silloin täytyy myös kertoa, mikä vaihtoehto tuli valituksi ja kuinka monta erilaista vaihtoehtoa on olemassa. Esimerkiksi jos sääntö on

      <RELATIONAL OPERATORS> ::= <>|=|<=|>=|<|>|IN,

   niin se koodataan

      1 sääntö käytiin lävitse
      7 7 eri vaihtoehtoa
      2 vaihtoehto 2 (=) valittiin.

   c) Jonossa merkitään jonon loppu numerolla 0. Esimerkiksi sääntö

      <IDENTIFIERS> ::= {, tunnus}

koodataan

1	jono on olemassa eli se käydään läpi
1	pilkku ja tunnus löytyi
0	kun jono loppuu.

d) Vaihtoehtoiset tilanteet koodataan, kuten sarjat.

Jäsentelijän käydessä lähdekoodia läpi ja tallentaessa säännöt, jotka se käy läpi, katoaa kaikki käyttäjän tunnukset, kuten muuttujien ja alirutiinien nimet. Ne täytyykin tallentaa jo selausvaiheessa symbolitauluun. Symbolitaulu sisältää käyttäjien tunnuksista myös tietoa sen tyypistä, sisennyksistä ja ennen kaikkea yksilöllisen kokonaisluvun, *ID-luku*, jolla ko. tunnus erotetaan muista tunnuksista. Nämä yksilölliset luvut tulevat mukaan jäsentelijän tuottamaan listaa ja toimivat siellä osoittimina symbolitauluun.

Selauksen aikana kannattaa ottaa selville, kuinka monta symbolia symbolitaulusta löytyy, sillä silloin tunnusten ID-luku voidaan koodata mahdollisimman vähällä bittimäärällä. Esimerkiksi bisect-ohjelmassa on 22 tunnusta, joten niiden koodaamiseen tarvitaan 5 bittiä.

Lisää tehokkuutta saadaan vaihtoehtojen koodaamiseen, sillä vaihtoehtojen määrä kutakin vaihtoehtoista sääntö kohden on tiedossa etukäteen. Siksi ei tarvitse erikseen koodata vaihtoehtojen määrää, vaan voidaan antaa suoraan tieto siitä, monesko vaihtoehto valittiin riittävällä määrällä bittejä. Esimerkiksi aikaisemmin esitetty relaatio-operaattorin valinta voidaan koodata seuraavasti:

1	<RELATIONAL OPERATORS>
010	toinen vaihtoehto seitsemästä.

Bisect-ohjelman koodaus muodostuu siis taulukon 7-2 mukaisesti.

Koodisana	Sääntö	Symboli
	varattu sana PROGRAM	PROGRAM
00001	tunnus = 1	bisect
	avaava sulku	(
1	<IDENTIFIERS>	
00010	tunnus = 2	input
1	{, tunnus}	
	pilkku	,
0011	tunnus = 3	output
0	jonon loppu	
	sulkeva sulku	)
	puolipiste	;
1	<BLOCK>	
0	<LABEL DCN PT>	
1	<CONST DFN PT>	
	varattu sana 'CONST'	'CONST'
1	<CONSTANT	
00100	tunnus = 4	Eps
	yhtäsuuruusmerkki	=
1	<CONSTANT>	
011	kolmas vaihtoehto	
1	<UNSIGNED NUMBER>	
00101	etumerkitön	1
0	ei <FRACTION PT>	
1	<EXPONENT>	
	eksponentti	E
1	<SIGN>	
10	negatiivi	-
00110	etumerkitön	14
	puolipiste	;
0	ei <SEQ CONST DFN>	
0	ei <TYPE DFN PT>	
1	<VARIABLE DCN PT>	

Taulukko 7-2 Bisect-ohjelman koodisanojen muodostuminen.

Tiivistetyn ohjelman purkaminen on helpompaa kuin tiivistäminen sillä purettaessa täytyy ainoastaan seurata, mitä sääntöjä on koodattu ja tarvittaessa hakea symbolitaulusta käyttäjän terminaalit oikeille paikoilleen. Tietysti symbolitaulukin kannattaa tiivistää, mutta siihen voidaan käyttää jotain perinteistä menetelmää.

Contlan koodausmenetelmän tuloksia on taulukossa 7-3. Ohjelma yksi on bisect-ohjelma, joka on hyvin pieni laskenta-ohjelma, jossa on hyvin vähän syöttö- ja

tulostustapahtumia. Ohjelma kaksi on keskikokoinen ohjelma, jossa on vähän laskentaa, mutta paljon syöttö- ja tulostustoimintoja. Kolmas ohjelma on myöskin keskikokoinen, mutta sisältää paljon laskentaa ja hyvin vähän syöttö- ja tulostustoimintoja.

Ohjelma	Koko (bittejä)	Tunnuksia	Polun koko (bittejä)	Tiivistetty koko (bittejä)	tiivistys-suhde
1	2088	22	668	1058	66%
2	16176	52	3631	5336	44%
3	13576	68	4310	6524	64%

Taulukko 7-3 Contlan tuloksia.

7.2 Kieliopin ja jäsentelijän valinta Stonen mukaan

Tässä kappaleessa käydään läpi Stonen artikkeli [11], jossa tutkitaan, millaisia vaikutuksia on kieliopin ja jäsentelijän valinnalla tiivistyksen tehokkuuteen.

Ohjelma voidaan esittää kieliopin sääntöjen avulla lähtemällä alkusymbolista. Jos aina valitaan vasemmanpuoleisin sääntö, on kyseessä vasemmanpuoleisin johto ja vastaavasti voidaan tehdä oikeimmanpuoleisimman säännön perusteella. Stone kutsuu tällaista koodausta *johtokoodaukseksi*. Toinen mahdollisuus koodata ohjelmakoodi on tallentaa jäsentelijän tekemät päätökset, kun se käy koodia läpi. Tällaista koodausta Stone kutsuu *jäsennyskoodauksesi*. Hän nimittää molempia yhdessä analyyttiseksi koodaukseksi.

Jäsennyskoodaus ja johtokoodaus tuottavat hieman erilaisen tuloksen, vaikka ovatkin samantyyppisiä menetelmiä. Esimerkiksi jollakin kieliopilla syntyy vain yksi johtokoodaus, mutta kieliopilla voi olla useita erilaisia jäsentelijöitä, jotka kaikki tuottavat erilaiset jäsennyskoodaukset. Ohjelmointikielillä voi tietysti olla useita

erilaisia kielioppeja, jotka tuottavat saman kielen. Näistä kaikista tulisi silloin erilaiset johto- ja jäsennyskoodaukset.

Silloin kun johto- tai jäsennyskoodauksessa tiedetään jokaisen valinnan suhteelliset todennäköisyydet, on melko helppo muodostaa näillä menetelmillä ohjelmakoodin tiivistysmenetelmä. Kuitenkin todennäköisyyksien selville saaminen edellyttäisi melko suurta tilastollista analyysia eikä niitä siten ole mielekästä kerätä kokeilemalla erilaisilla kieliopeilla ja jäsentelijöillä. Ongelmana onkin tietää etukäteen, mikä kielioppi ja mikä koodaus tuottaa parhaan tuloksen.

7.2.1 Johtokoodauksen valinta

Otetaan esimerkiksi kielioppi G_1 joka on esitetty taulukossa 7-4 ja ohjelma a,a,b.

Sääntö	Koodi
L::=E	L0
L::=E, L	L1
E::=a	E0
E::=b	E1

Taulukko 7-4 Kielioppi G_1.

Vasemmanpuoleisin koodaus tuottaa nyt koodin: L1, E0, L1, E0, L0, E1. Koska jokaisella säännöllä on vain kaksi eri vaihtoehtoa, voidaan tässä tapauksessa koodata seuraavanlaisella binaarijonolla: 101001. Siinä merkitään 0-bitillä ensimmäistä sääntöä ja 1-bitillä jälkimmäistä. Vastaavasti oikeanpuoleisin koodaus tuottaa koodin: L1, L1, L0, E1, E0, E0. Tämä ei ole vasemmanpuoleisin koodi käännettynä, vaan sen permutaatio. Binaarijonon pituus on siis edelleen samankokoinen. Siitä seuraa, että koodauksen kannalta voidaan valita vapaasti parhaiten kuhunkin tilanteeseen sopivin sääntöjen soveltamisjärjestys.

7.2.2 Kieliopin valinta

Otetaan lähempään tarkasteluun seuraavat neljä tyyppiä sekä niiden merkitykset. Tässä, kuten muuallakin, merkki λ tarkoittaa tyhjää merkkijonoa.

1. jono: P ::= A B C ...

2. valinta: P ::= A
P ::= B
...

3. toisto: P ::= λ
P ::= A P

4. listat: P ::= A
P ::= A, P.

Jono liittyy tilanteeseen, jossa kieliopissa on vain yksi P–sääntö eli sääntö, jonka vasemmalla puolella on apumerkki P. Kun kieliopin sääntö P ::= A B C on valittu sovellettavaksi vasemmanpuoleisimmassa koodauksessa, apumerkki A valitaan todennäköisyydellä 1.0. Myöhemmin myös apumerkkeihin B ja C sovelletaan jotain sääntöä. Tässä säännössä ei siis ole mahdollisuuksia valintaan eikä näin ollen tarvitse myöskään tarvitse tiivistettäessä tuottaa koodia. Jonot eivät siis tuota koodia tiivistettäessä.

Tarkastellaan seuraavaksi tapausta, jossa apumerkillä on kaksi tai useampia sellaisia sääntöjä, joiden vasemmalla puolella kyseinen apumerkki on. Esimerkiksi Pascal-ohjelmointikielessä on seuraavanlaisia sääntöjä:

Stmt ::= Structured-Stmt

Stmt ::= Simple-Stmt

Structured-Stmt ::= begin ...

Structured-Stmt ::= while ...

Simple-Stmt ::= Variable Expression

Simple-Stmt ::= Proc-Ident Actual-Param-List-Option.

Näistä voidaan yleistää kielioppi G_2:
S ::= C
S ::= U
C ::= b
C ::= w
U ::= v
U ::= p.

Toinen mahdollisuus olisi kielioppi G_3, jossa säännöt
S ::= b
S ::= w

S ::= v
S ::= p.

G_3 tuottaa saman kielen kuin kielioppi G_2. Sanotaan, että kielioppi G_3 on saatu kieliopista G_2 kerrostuttamalla. Jotta kielioppeja G_2 ja G_3 voitaisiin verrata toisiinsa, täytyy lauseille b, w, v ja p antaa todennäköisyydet p_b, p_w, p_v ja p_p. Nyt voimme antaa säännöille todennäköisyydet taulukkojen 7-5 ja 7-6 mukaisesti.

Sääntö	Todennäköisyys	Koodisana
S ::= C	p_b+p_w	S0
S ::= U	p_v+p_p	S1
C ::= b	$p_b/(p_b+p_w)$	C0
C ::= w	$p_w/(p_b+p_w)$	C1
U ::= v	$p_v/(p_v+p_p)$	U0
U ::= p	$p_p/(p_v+p_p)$	U1

Taulukko 7-5 Todennäköisyydet kieliopin säännöille G_2.

Sääntö	Todennäköisyys	Koodisana
S ::= b	p_b	Sb
S ::= w	p_w	Sw
S ::= v	p_v	Sv
S ::= p	p_p	Sp

Taulukko 7-6 Todennäköisyydet kieliopin G_3 säännöille.

Ohjelmilla on nyt taulukossa 7-7 esitetyt vasemmanpuoleisimmat koodaukset.

Ohjelma	G_2 koodaus	G_3 koodaus
B	S0C0	Sb
W	S0C1	Sw
V	S1U0	Sv
P	S1U1	Sp

Taulukko 7-7 Esimerkkiohjelman vasemmanpuoleiset koodit.

Jos koodisanat ovat Huffmannin koodeja ja siten optimaalisia, keskimääräinen ohjelmien koodattu pituus kieliopilla G_2 on

$$(p_b+p_w)*L(S0) + (p_v+p_p)*L(S1) + (p_b+p_w)*(p_b/(p_b+p_w))*L(C0) +$$
$$(p_b+p_w)*(p_w/(p_b+p_w))*L(C1) + (p_v+p_p)*(p_v/(p_v+p_p))*L(U0) +$$
$$(p_v+p_p)*(p_p/(p_v+p_p))*L(U1),$$

missä L(s) on merkkijonon s pituus. Samoin kieliopilla G_3 keskimääräinen pituus on

$$p_b*L(Sb) + p_w*L(Sw) + p_v*L(Sv) + p_p*L(Sp).$$

Voidaan osoittaa, että keskimääräinen pituus kieliopilla G_2 ei ole koskaan pienempi kuin kieliopilla G_3. Merkitään S0C0 kun liitetään koodit S0 ja C0 yhteen ja samalla lailla muille koodisanoille. Nyt S0C0, S0C1, S1U0 ja S1U1 ovat koodisanoja, joilla voidaan koodata ohjelmat kieliopilla G_3. Koska L(S0C0) = L(S0) + L(C0), joka on sama kuin keskimääräinen pituus kieliopilla G_2. Jos näin ei olisi, niin silloin kieliopilla G_3 ei olisi optimaalisia koodisanoja, jolloin syntyisi ristiriita.

Yleistyksenä voidaan todeta, että mikään kieliopin kerrostuttaminen ei voi lisätä koodauksen tehokkuutta, vaan yleensä vain heikentää sitä. Siis kannattaa pitää vaihtoehtojen määrä mahdollisimman suurena. Parasta tiivistystä haluttaessa kannattaa kieliopit muodostaa siten, että oikea puoli säännöstä alkaa aina terminaalilla tai on tyhjä. Greibachin normaalimuodossa oleva kielioppi on juuri tällaisessa muodossa.

Tutkitaan seuraavaksi toistorakennetta tarkastelemalla seuraavanlaista kielioppia G_4:

P ::= λ
P ::= A P
A ::= a
A ::= b.

Tällä kieliopilla johtokoodaus koodaisi jokaisen merkin a kahdella bitillä, kerran säännölle P ::= A P ja kerran säännölle A ::= a. Myöskin jokainen merkki b koodattaisiin kahdella bitillä ja lopuksi P ::= λ tuottaisi yhden bitin. Ohjelman pituus voidaan laskea kaavalla $L = 2*N_a + 2*N_b + 1$, missä N_a on merkkien a lukumäärä ja N_b merkkien b lukumäärä. Kielioppi olisikin parempi kirjoittaa seuraavanlaiseen muotoon.

P ::= λ
P ::= a P
P ::= b P.

Nyt koodin pituus voidaan laskea kaavalla $L = L_a*N_a + L_b*N_b + L_t$, missä L_a, L_b ja L_t ovat merkkien a-, b- ja λ-valintojen koodien pituudet. Jos koodisanat ovat Huffmanin koodeja, niiden pituudet ovat pakostakin 2, 2 ja 1 siten, että kaikkein todennäköisin

sääntö on yhden bitin pituinen. Jos merkki a tai merkki b on todennäköisin, tulee tällä kieliopilla lyhyempi koodaus kuin alkuperäisellä kieliopilla G_4. Siinäkin tapauksessa, että λ on yleisin merkki, niin koodaus on samanpituinen kuin alkuperäisellä kieliopilla. Tietysti kielioppia voidaan edelleen tehostaa muodostamalla seuraavanlaisia sääntöjä: P ::= a, P ::= b, P ::= a a P, P ::= a b P, jne. Tällaiselle muokkaamiselle tulee tietysti käytännössä jossain vaiheessa raja vastaan, mutta teoreettisesti sitä voidaan jatkaa kuinka pitkään tahansa.

Tutkitaan lopuksi listoja. Tarkastellaan uudestaan kielioppia G_1. Tämä kielioppi voidaan parantaa muotoon G_7, joka on

L ::= a

L ::= b

L ::= a, L

L ::= b, L.

Myös tätä kielioppia G_7 voitaisiin parantaa samalla lailla kuin aikaisemmin kielioppia G_3.

7.2.3 Jäsennyskoodaus vai johtokoodaus

Seuraavaksi tutkitaan, saadaanko jäsennyskoodauksella parempia tuloksia samanlaisille osille (jono, valinta, toisto ja lista). Kuten johtokoodauksessa todettiin ei jäsennyskoodauksessa koodata jonoja ollenkaan. Tässä ja myöhemminkin oletetaan tavallisten LR- ja LL-jäsentelijöiden rakenne ja toiminta tunnetuiksi.

Jäsennyskoodaus osaa tehdä valintoja suoraan kielopin G_3 tapaan vaikka kielioppina olisikin G_2. LR-jäsentelijän tilat kieliopille G_2 voisivat olla taulukon 7-8 mukaiset.

Tila		Jäsennysvaihtoehdot
0	[A:_S][s:_C][S:_U][C:_b] [C:_w][U:_v][U:_p]	case b: shift 4 case w: shift 5 case v: shift 6 case p: shift 7
1	[A:S_]	oletus: hyväksy
2	[S:C_]	oletus: pelkistä käyttäen sääntöä S ::= C
3	[S:U_]	oletus: pelkistä käyttäen sääntöä S ::= U
4	[C:b_]	oletus: pelkistä käyttäen sääntöä C ::= b
5	[C:w_]	oletus: pelkistä käyttäen sääntöä C ::= w
6	[U:v_]	oletus: pelkistä käyttäen sääntöä U ::= v
7	[U:p_]	oletus: pelkistä käyttäen sääntöä U ::= p

Taulukko 7-8 LR jäsentelijän tila kieliopille G_2.

Koska koodia ei tarvitse tuottaa tiloille 1-7, on kieliopin G_2 koodaus yhtä hyvä kuin kieliopin G_3 koodaus.

Samalla lailla käy toistojen listojen kanssa. Jäsentelijä kieliopille G_4 tuottaa koodia lähestulkoon samalla lailla kuin kielioppi G_5 tuottaisi jäsennyskoodauksella. Listakieliopilla G_1 jäsentelijä tuottaa yhtäpitkän koodin kuin vasemmanpuoleisin johtokoodaus. Kuitenkin jos kieliopissa olisi kolme tai enemmän lisävalintoja kuten: E ::= a, E ::= b ja E ::= c, saattaisi LR-jäsentelijä saada lisää etua. Etenkin jos listan alkamisen todennäköisyys merkillä c olisi suuri, mutta myöhemmin merkin c esiintymisen todennäköisyys olisikin pieni.

7.2.4 Jäsentelijän valinta

Tutkitaan taas kielioppia G_4. LL-jäsentelijä voidaan kuvata taulukolla, missä rivit indeksoidaan apumerkeillä ja sarakkeet indeksoidaan lähdekoodin merkeillä. Kieliopin G_4 tapauksessa saadaan taulukko 7-9

	A	b	$end
P A	P ::= A A ::= a	P ::= A A ::= b	P ::= λ

Taulukko 7-9 LL-jäsentelijä.

LL-jäsentelijä käyttää pinoa, jossa on kieliopin symbolit, ja tekee päätöksen jäsennyksen etenemisestä aina kun pinon päällimmäisenä on apumerkki. Apumerkki ja seuraava syötemerkki yhdessä toimivat koordinaatteina tauluun, joka kertoo, mitä sääntöä seuraavaksi sovelletaan.

LL-jäsentelijä ei koskaan laajenna saatavilla olevia vaihtoehtoja kuten LR-jäsentelijä. LL-jäsentelijä käsittelee kaikki merkit a ja b samalla lailla eikä pidä ensimmäisiä merkkejä listassa mitenkään erikoisina toisin kuin LR-jäsentelijä. LR-jäsentelijä pystyy siis valitsemaan tilanteesta riippuen paremmin, minkä säännön se seuraavaksi valitsee. Siitä pitäisi olla käytännössä hyötyä.

7.2.5 Kielen entropia

Perinteinen entropian kaava $-\Sigma p_i * \log(p_i)$ ei ole kovinkaan käytännöllinen tapa laskea ohjelmakoodin entropiaa. Kuitenkin jos kielelle on olemassa kielioppi todennäköisyyksineen, voidaan entropia päätellä. Huomattakoon, että lähdekoodin jakaminen ei muuta sen entropiaa eli kaikki todennäköisyyskieliopit, jotka kuvaavat saman kielen tuottavat saman entropian. (Todennäköisyyskieliopilla tarkoitetaan tässä kielioppia, jonka A-säännöillä on kiinteät soveltamistodennäköisyydet, joiden summa on 1.) Kuitenkin kielioppien G_2 ja G_3 yhteydessä huomattiin, että lähdekoodin jakaminen voi vaikuttaa keskimääräiseen koodin pituuteen.

Oletetaan, että lähteestä tulee viestejä todennäköisyyksillä p_0, p_1, ..., p_n. Viestit voidaan jakaa kahteen ryhmään, jotka sisältävät viestit 0 - k ja k+1 - n. Merkitään

$PL = p_0 + p_1 + ... + p_k$ ja

$PH = p_{k+1} + P_{k+2} + ... + p_n.$

Alemman ryhmän, PL, entropia on

$$HL = \sum_{i=0}^{k} \left(\frac{p_i}{PL} \log \left(\frac{PL}{p_i} \right) \right)$$

ja vastaavalla tavalla lasketaan ylemmän ryhmän entropia HH. Näin ollen koko lähteen entropia on

= -PL*log(1/PL) + PL*HL - PH*log(1/PH) + PH*HH

= -PL*log(1/PL)- $\sum_{i=0}^{k} p_i$ *log(PL/p_i) - PH*log(1/PH)- $\sum_{i=I+1}^{n} p_i$ *log(PH/p_i)

$$= -\sum_{i=0}^{n} p_i \log \left(\frac{1}{p_i} \right).$$

Siis jaetun lähteen entropia on sama kuin alkuperäisen lähteen.

Jatketaan nyt entropian määrittelemistä. Määritellään ensiksi matriisi Q, jossa on sellaiset alkiot q_{ij}, että jos apumerkki i esiintyy vasemmalla puolella sääntöä j, niin q_{ij} on säännön j soveltamistodennäköisyys, muuten q_{ij} = 0. Kieliopille G_4 todennäköisyyksillä P0, P1, A0 ja A1 on

$$Q = \begin{vmatrix} P0 & P1 & 0 & 0 \\ 0 & 0 & A0 & A1 \end{vmatrix}.$$

Määritellään sellainen matriisi C alkioilla c_{ij}, että c_{ij} on apumerkin j esiintymiskertojen määrä säännössä i oikealla puolella. Kieliopilla G4 saadaan matriisi

$$C = \begin{vmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{vmatrix}.$$

Määritellään matriisi A matriisituloksi A = Q*C eli

$$A = \begin{vmatrix} p1 & p1 \\ 0 & 0 \end{vmatrix}.$$

Lopuksi määritellään matriisi A’ = 1/(1-A)

$$A' = \begin{vmatrix} \frac{1}{p_0} & \frac{p_1}{p_0} \\ 0 & 1 \end{vmatrix}.$$

Matriisin A' ensimmäinen rivi, joka vastaa aloitussymbolia, antaa odotusarvon sille, kuinka monta kertaa kukin apumerkki esiintyy sattumanvaraisessa johdossa. Käsittelemällä apumerkkejä erillisinä lähteinä voidaan niiden entropiat laskea. Kieliopin tuottaman kielen entropia on siis

$$H = \sum_{j=1}^{N} A_{ij} * H_j ,$$

missä H_j on apumerkin j entropia ja N kaikkien apumerkkien lukumäärä.

7.2.6 Stonen ehdotus koodausmenetelmäksi

Seuraavaksi yritetään parantaa aikaisemmin esitettyjen menetelmien heikkoja kohtia toistojen ja listojen käsittelyssä. Tarkoituksena on parantaa koodauksen informaatioherkkyyttä. Esimerkiksi taulukon 7-10 kielioppi määrittelee saman kielen kuin kielioppi G_1.

Sääntö	Koodisana
L ::= E	L0
L ::= pari	L1
L ::= E, pari	L2
pari ::= E, E	P0
pari ::= E, E,	P1
E ::= a	E0
E ::= b	E1

Taulukko 7-10 Kielioppi G_{1b}.

Vasemmanpuoleisin johto alkaa kolmivalinnoilla ja pidemmillä ohjelmilla jatkuu parien etsimisillä. LR-jäsentelijällä tälle kieliopille on neljä tilaa, jotka valitsevat merkkien a ja b väliltä, ja sillä on kolme tilaa, joilla valitaan mahdollinen jatko tai ei. Näin ollen LR-jäsennyskoodaus on samanlainen kuin johtokoodaus. Taulukossa 7-11 on verrattu molempia menetelmiä eripituisilla listoilla. LR-jäsentelijän sarakkeessa merkki B tarkoittaa valintaa kahdesta (binaarinen valinta).

Pituus	vasemmanpuoleisin	LR
1	L0	B
2	L1P0	BB
3	L2P0	BBB
4	L1P1P0	BBBB
5	L2P1P0	BBBBB
6	L1P1P1P0	BBBBBB
7	L2P1P1P0	BBBBBBB
8	L1P1P1P1P0	BBBBBBBB

Taulukko 7-11 Koodausmenetelmien vertailu.

Koodisanojen L0, L1 ja L2 pituudet ovat 1, 2 ja 2 bittiä jossakin permutaatiossa sekä koodisanat P0 ja P1 ovat yhden bitin pituisia kumpikin. Näin ollen listoille, joiden pituus on 5 tai enemmän, vasemmanpuoleisin koodaus on parempi. Ainoastaan listoille, joiden pituus on 1 tai 2 on LR-jäsentelijä parempi. Jos tätä pidempien listojen todennäköisyys on riittävän suuri, kannataa siis käyttää vasemmanpuoleista jäsennyskoodausta. Vasemmanpuoleisin jäsennyskoodaus täytyy kuitenkin tuottaa peruuttavalla jäsentelijällä. Jos käytössä on rekursiivinen jäsentelijä, täytyy kielioppi muuntaa takaisin kieliopiksi G_1.

Suurempien osalistojen ottamista omiksi säännöiksi voidaan jatkaa vielä pidemmälle, vaikkapa seuraavanlaiseksi kieliopiksi G_{1c}:

L ::= E

L ::= E, E

L ::= E, E, E

L ::= neliköt

L ::= E, neliköt

L ::= E, E, neliköt

L ::= E, E, E, neliköt

neliköt ::= E, E, E, E

neliköt ::= E, E, E, E, neliköt

E ::= a

E ::= b.

Kielioppi näyttää aika omituiselta, joten kannattaa ottaa käyttöön uusi merkintä, tähti, joka tarkoittaa ”niin monta kertaa kuin on tarpeen”. Nyt kielioppi voidaan esittää muodossa

L ::= E (, E)*
E ::= a
E ::= b.

Vasemmanpuoleisin johtokoodaus olisi nyt (E0 | E1)(n)(E0 | E1) ... (E0 | E1). Sen koodaamiseen tarvitaan sellainen etumerkkikoodi luonnollisille luvuille, joka on optimaalinen todennäköisille listan pituuksille. Stone ehdottaa seuraavanlaista menetelmää. Etsitään äärellisestä määrästä ohjelmia pisin jonon pituus N. Oletetaan, että jonon pituudet 0 - N esiintyvät todennäköisyyksillä p_i, i = 0(1)N. Luodaan Huffmannin koodit N+2 viestille. Viestien 0 - N todennäköisyydet ovat P_i sekä erikoisviesti äärettömän pienellä todennäköisyydellä. Jos käytössä tulee vastaan ohjelma, jossa on jono, jonka pituus on suurempi kuin N, lähetetään erikoisviesti sekä loppupituus unaarikoodeilla, jossa 0 toimii erottimena.

7.3 Muotoilutietojen säilyttäminen tiivistyksessä

Davies ja Witten [4] ovat kehittäneet menetelmän, jolla myös koodin muotoilu saadaan säilymään. He koodaavat tiivistykseen mukaan myös suhteelliset siirtymäkoordinaatit, jotka kertovat, mistä seuraava merkkijono alkaa suhteessa edelliseen merkkijonoon. Esimerkiksi koordinaatit (0, 0) tarkoittavat, että seuraava merkkijono on samalla rivillä edellisen kanssa eikä niiden välillä ole yhtään välilyöntiä. Vastaavasti koordinaatit (1, 4) tarkoittavat sitä, että seuraava symboli sisältää rivinvaihdon ja on siten seuraavalla rivillä (**1**, 4) ja sisennetty neljä (1, **4**) välilyöntiä oikealle. Paluu alkuperäiseen sarakkeeseen voidaan antaa negatiivisella luvulla. Koordinaatit voidaan tallentaa yhtenä yksikkönä ja jos koodataan jo esiintynyttä koordinaattia, tallennetaan vain viittaus aikaisempaan koordinaattiin. Jos jompikumpi tai molemmat koordinaatin arvoista ovat uusia, tallennetaan ne erikseen. Koska tavallisessa lähdekoodissa käytetään yleisesti vain muutamia siirtymiä, niin koodaus saadaan tehokkaaksi. Joka tapauksessa tiivistetyn koodin

koko kasvaa jonkin verran, mutta vastaavasti purkuvaiheessa voidaan alkuperäinen koodi palauttaa täsmällisesti.

Ohjelmakoodin symbolitaulu, joka sisältää koodin tunnukset (*identifiers*) eli muuttujien nimet sekä vakiot, täytyy kuitenkin koodata jollain perinteisellä menetelmällä. Mielellään myös kommentit koodataan mukaan, vaikka ne eivät olekaan perinteisesti symbolitaulussa. Tiivistystä varten ne voidaan kuitenkin katsoa kuuluviksi *käyttäjän terminaaleihin* (muuttujien nimet, vakiot, jne.) ja siten varastoida symbolitauluun. Tällainen toimenpide vaatii muutoksen myös kielioppiin, joten kommentit voidaan koodata myös erikseen, jos kielioppiin ei haluta tehdä kovin suuria muutoksia. Myöhemmin esitetään kuitenkin menetelmiä, joiden avulla tiivistystä voidaan tehostaa ja tällöin joudutaan yleensä muokkaamaan myös kielioppia.

8. Kielioppiin pohjautuva informaatiomalli

Kielioppiin perustuvassa informaatiomallissa todennäköisyyksiä ei tutkita yksittäisten merkkien kannalta, vaan siitä millaisia sääntöjä noudatetaan viestiä luotaessa. Näitä sääntöjä kutsutaan *kieliopiksi*, joka määrittelee *kielen*. Kieli on kaikkien niiden merkkijonojen joukko, jotka voidaan tuottaa kyseisen kielen määrittävällä kieliopilla.

Kontekstittomat kieliopit eli sellaiset kieliopit, joiden säännöissä on vasemmalla puolella vain yksi symboli, voidaan ilmaista Backus-Naur normaalimuodossa *BNF*. BNF-kielioppi on neilikkö $\Gamma = (\Sigma_N, \Sigma_T, \Phi, \delta_0)$, jossa Σ_N on *apumerkkien* joukko, Σ_T on *terminaalien* joukko, joka on erillinen Σ_N-joukon kanssa. Φ on *sääntöjen* joukko ja δ_0, *aloitussymboli*, on joukon Σ_N alkio. Aloitusmerkki δ_0 kuvaa kielen suurinta kieliopillista osaa, joka on ohjelmakoodeissa, koodi itse. Terminaalit, Σ_T, ovat ne merkit, joita kielessä voi esiintyä ja joukon Φ alkiot ovat sääntöjä, jotka määrittelevät, miten kieliopillinen yksikkö muodostuu. Jokaista apumerkkiä kohden täytyy joukossa Σ_N olla jokin sääntö, jonka vasemmalla puolella kyseinen apumerkki on. [2]

Säännöt ovat muotoa $\delta ::== \alpha_1 | \alpha_2 | \ldots | \alpha_n$, missä δ on jokin joukon Σ_N apumerkki ja kukin α_i on jokin vaihtoehtoinen tapa muodostaa δ. Vaihtoehdot koostuvat terminaaleista ja apumerkeistä, jotka korvaavat apumerkin δ. Tällaista korvaamista kutsutaan *säännön soveltamiseksi*. Jos aloitussymbolista voidaan muodostaa merkkijono, joka koostuu terminaaleista, niin tämä merkkijono kuuluu ko. kieleen. Samalla voidaan muodostaa kuvio, joka kertoo mitä sääntöjä missäkin vaiheessa ollaan käytetty. Tällainen kuvio muodostaa puun, jota kutsutaan *jäsennyspuuksi*. Vastaavasti voimme tutkia, kuuluuko jokin merkkijono kieleen vai ei, yrittämällä muodostaa jäsennyspuu, jonka lehdissä on merkkijonon merkit ja juuressa aloitussymboli. Tällaista toimintaa kutsutaan *jäsentelyksi*.

Olkoon $P_S(\delta{:}i)$ todennäköisyys, jolla apumerkki δ korvataan vaihtoehdolla i. Tällöin annetun viestin todennäköisyys voidaan laskea niiden sääntöjen tulona, joka tarvitaan viestin jäsentelyyn [2].

Olkoon $I_S(\delta{:}i)$ yhden sellaisen säännön johtamisen informaatiosisältö bitteinä, jossa apumerkki δ korvataan vaihtoehdolla *i*. Tällöin $I_S(\delta{:}i) = -\log_2 P_S(\delta{:}i)$. Koko viestin informaatiosisältö on yksittäisten johtojen informaatiosisältöjen summa.

Kieliopin entropia voidaan laskea seuraavasti. Olkoon $H_S(\delta)$ kaikkien niiden merkkijonojen keskimääräinen informaatiosisältö, jotka δ muodostaa. Tällöin $H_S(\delta)$ = 0, jos δ on terminaalimerkki, koska mitään johtoja ei voida enää soveltaa. Jokaiselle johdolle $\delta ::== \alpha_1 | \alpha_2 | \ldots | \alpha_n$ olkoon $n_\Sigma(\alpha_i)$ terminaalien ja ei-terminaalien lukumäärä vaihtoehdossa *i*. Merkitään α_{ij} sitä symbolia, joka on vaihtoehdossa *i* paikalla *j*. Nyt jokaiselle apumerkille δ joukossa Σ_N on voimassa yhtälö

$$H_S(\delta) = \sum P_S(\delta : i) \times \left[-\log P_S(\delta : i) + \sum_{j=1}^{n_\Sigma(a_i)} H_S(\alpha_{ij})\right] \quad .$$

Löytämällä ratkaisu kaikkiin näihin yhtälöihin saadaan kieliopin informaatiosisältö, joka ilmaistaan bitteinä viestiä kohden. On myös mahdollista, ettei ratkaisua yhtälöryhmään löydy. Tällöin todennäköisyydet eri säännöille ovat mahdottomia. Jos todennäköisyydet annetaan todellisten merkkijonojen pohjalta, tällaisia tilanteita ei voi syntyä. [2]

9. Oman algoritmin kuvaus

Tässä luvussa kuvataan, kuinka tekemäni kielioppiin perustuva tiivistysohjelma toimii teorian tasolla.

Lähdekoodin tiivistys voidaan jakaa seuraaviin osatehtäviin. (Prosessia on yritetty selventää kuvassa 9-1.)

Jaetaan lähdekoodi *merkkeihin* (tokens), jotka luokitellaan *kieliopillisiin terminaaleihin* (avainsanat, operaattorit, välimerkit, jne.) ja käyttäjän terminaaleihin. Käyttäjän terminaalit tallennetaan symbolitauluun. Tämä vaihe voidaan toteuttaa selausohjelmalla, joka voidaan tuottaa etukäteen esimerkiksi *LEX*-kääntäjätyökalulla. Omassa ohjelmassani olen käyttänyt LEX–ohjelman kanssa yhteensopivaa, mutta hieman tehokkaampaa Flex–ohjelmaa, joka on saatavilla GNU lisenssin alla lähes kaikille tietokoneille.

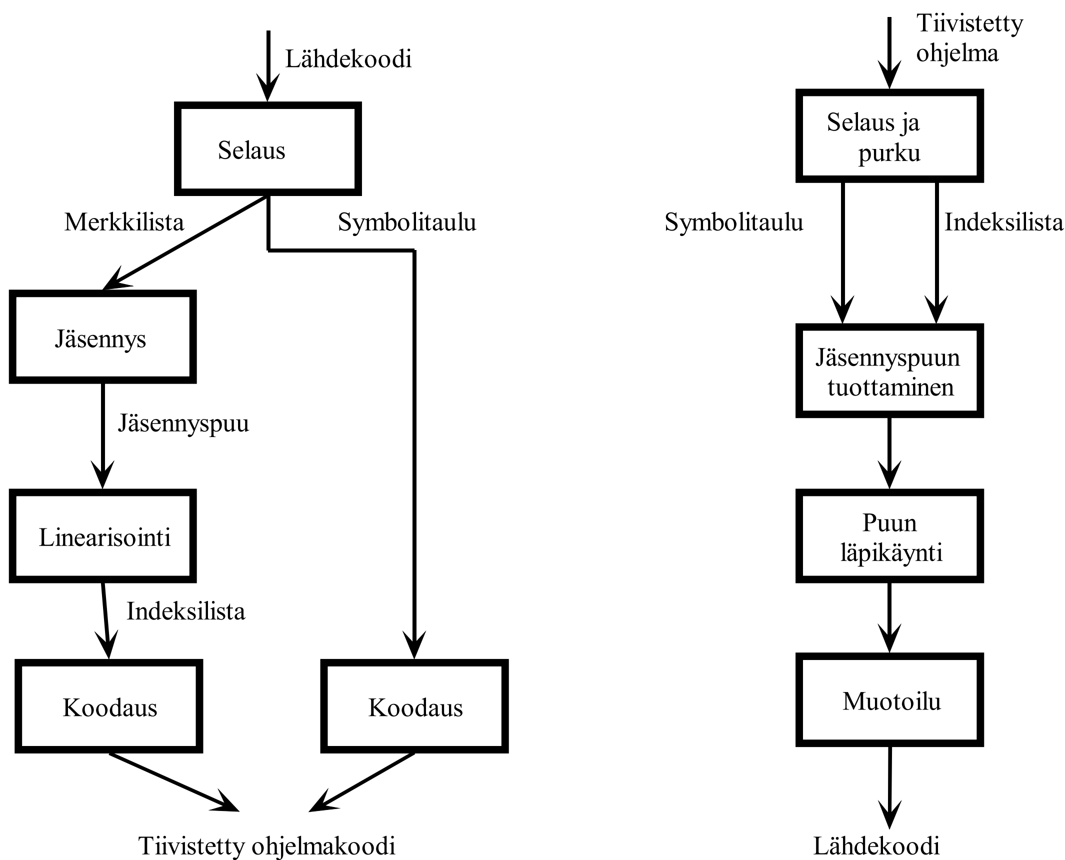
Seuraavaksi tuotetaan jäsentelijällä jäsennyspuu, jonka solmuissa on kahdentyyppisiä indeksejä: kieliopin mukaisia johtoja sekä osoittimia symbolitauluun. Puuta voidaan yksinkertaistaa, koska kieliopillisia terminaaleja ei tarvita myöskään sellaisia sisäsolmuja, joista voidaan tuottaa vain yksi vaihtoehto, ei tarvitse tallentaa.

Kolmannessa vaiheessa edellä tuotettu jäsennyspuu linearisoidaan eli solmut käydään läpi esimerkiksi esijärjestyksessä. Tuloksena oleva kokonaislukulista esitetään mahdollisimman tehokkaalla tavalla käyttäen esimerkiksi jotain perinteistä tiivistysmenetelmää kuten aritmeettista koodausta tai Lempelin – Zivin –koodausta, jota olen käyttänyt omassa ohjelmassani. Toinen mahdollisuus olisi esittää syntyneet kokonaisluvut esimerkiksi Elias –koodeilla.

Viimeisessä vaiheessa symbolitaulu tiivistetään vielä jollain perinteisellä menetelmällä.

Edellä kuvattu tiivistysmenetelmä siis hukkaa kaikki muotoilusäännöt. Ne voidaan koodata kuitenkin erikseen, jos niiden säilyttäminen on oleellista. Tällöin tiivistyksen tehokkuus luonnollisesti kärsii hieman.[4] Vastaavasti tiivistyksen purku tapahtuu päinvastaisessa järjestyksessä.

Jäsentelijä voidaan tuottaa automaattisesti esim. *YACC*-ohjelmalla tai muulla vastaavalla työkalulla. Se edellyttää, että kielioppi on kontekstion (context free), jollaisia ovat kaikki kieliopit, joissa säännön vasemmalla puolella on vain yksi symboli. Ne voidaan siis esittää BNF-notaatiolla. Lähes kaikki ohjelmointikielet ovat kontekstittomia. Tällainen ohjelma kuuluu UNIX-käyttöjärjestelmän perustyökaluhin. Omassa ohjelmassani olen käyttänyt Bison nimistä vastaavaa ohjelmaa, joka Flex–ohjelman tavoin on saatavissa lähes kaikkiin tietokonemalleihin.



Kuva 9-1 Vasemmalla puolella on kuvattu tiivistyksen työvaiheet ja oikealla vastaavasti purkuun liittyvät toimet [7].

Esimerkkinä käydään läpi seuraavanlainen Pascalia muistuttava ohjelmakoodin pätkä.

```
BEGIN
      a := 10;
      b := 20;
      tulos := a*b;
END
```

Hieman supistettuna Pascalin kielioppi voidaan kuvata seuraavasti:

```
p1:program :==  'BEGIN' statements 'END'
     p2:statements :== statement ';' statements
     p3:statements :== statement (§
     p4:statement :== IDENTIFIER ':=' expression
     p5:statement :== ''
     p6:expression :== IDENTIFIER (§
     p7:expression :== IDENTIFIER '*' IDENTIFIER
     p8:expression :== INTEGER (§
```

Pykälällä merkityt kieliopin säännöt voidaan ohittaa, koska niiden oikealla puolella on vain yksi apumerkki tai terminaali.

Lisäksi tarvitsemme seuraavanlaisia leksikaalisia termejä:

`'BEGIN', 'END', ';', ':=', '*'`

Käyttäjän terminaalit esimerkissämme ovat

`IDENTIFIER`, `INTEGER`, jotka muodostuvat seuraavasti:

`IDENTIFIER ::= LETTER{LETTER|DIGIT}*`

eli jokainen tunnus alkaa kirjaimella, jonka jälkeen voi tulla nolla tai useampia kirjaimia tai numeroita. Vastaavasti INTEGER voidaan esittää muodossa

`INTEGER ::= DIGIT{DIGIT}*.`

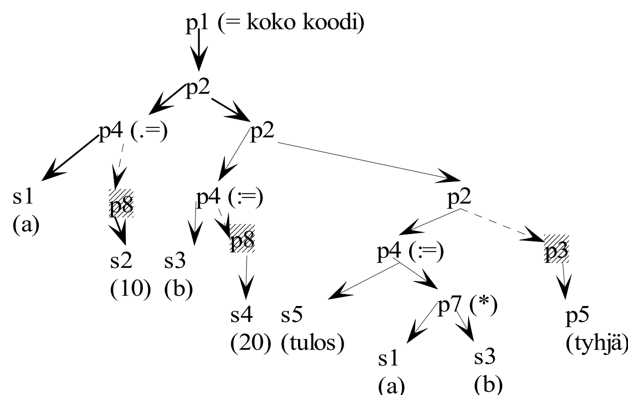
```
LETTER ja DIGIT ovat muotoa

LETTER ::= 'A',...'Z', '_'
```

eli kirjaimet koostuvat englanninkielisistä aakkosista ja alaviivasta ja

```
DIGIT ::= '0', ..., '9'.
```

Kielioppiin perustuvassa tiivistyksessä ei tarvitse välittää leksikaalisten merkkien semanttisesta sisällöstä eli siitä, mitä ne merkitsevät siinä ympäristössä, missä ne esiintyvät. Oikeat kääntäjät joutuvat tietysti huomioimaan myös tämän seikan. Sen vuoksi monet jäsentelijöitä tuottavat ohjelmat vaativat, että leksikaalinen analysaattori tuottaa myös merkkien merkityksen. Jäsentelijän ei kuitenkaan tarvitse siitä välittää.

Ohjelmakoodin selaus siis tuottaa nyt vain leksikaaliset merkit ja symbolitaulun. Symbolitaulu sisältää kaikki muuttujat eli esimerkissämme muuttujat *a*, *b*, ja *tulos* sekä vakiot *10* ja *20*. Yhteensä siis viisi alkioita. Symbolitaulu tiivistetään jollain perinteisellä menetelmällä kuten esimerkiksi aritmeettisella koodauksella, joten sitä ei käsitellä tässä sen enempää. Leksikaaliset merkit, *BEGIN*, *a*, *:=, 10, ;, b, :=, 20, ;, tulos, :=, a, *, b, ;, END*, jäsennellään puuksi, joka on esitetty kuvassa 9-2. Koska jäsennyspuussa viitataan symbolitaulun alkioihin, niin merkitään niitä vielä seuraavasti: *s1 = a, s2 = 10, s3 = b, s4 = 20* ja *s5 = tulos* .



Kuva 9-2 Esimerkkiin liittyvä jäsennyspuu.

Kun jäsennyspuu linearisoidaan esimerkiksi käymällä se läpi esijärjestyksessä eli ensin mainitaan juuri, sitten vasen alipuu ja lopuksi vielä oikea alipuu, saadaan esimerkkitapauksessa seuraavanlainen lista: p1p2p4s1s2p2p4s3s4p2p4s5p7s1s3p5. Tämä lista pitää vielä esittää mahdollisimman tehokkaassa muodossa. Yksinkertaisimmillaan, joskaan ei kovin tehokkaasti, se voidaan esittää 4-bittisinä koodeina seuraavasti: px ->x ja sx ->x+8, jolloin 3 alinta bittiä käytetään itse koodiin ja ylin bitti kertoo, onko kyseessä viittaus symbolitauluun vai kieliopin sääntöihin. Tällöin koko puun esittämiseen tarvittaisiin 16*4 = 64 bittiä. Tähän tulee kuitenkin lisätä vielä symbolitaulun koodaaminen. Jäsennyspuun sääntöjen määrä on selvillä jo etukäteen eli ne voitaisiin esittää hyvinkin kiinteänpituisilla koodeilla. Symbolitaulun kokoa ei tietystikään voida ennakoida, vaan se vaihtelee aina koodattavan materiaalin mukaan. Tehokkaampi koodaus saadaankin käyttämällä esimerkiksi aritmeettista koodausta lukujen esittämiseen. Tuloksena olevaa lukujonoa voidaan tietysti ajatella ihan tavallisena tekstinä ja tiivistää se millä tahansa tavanomaisella koodauksella välittämättä sisällön merkityksestä.

Tiivistystehokkuutta voidaan hieman lisätä, jos jätetään tyhjät johdot pois, koska ne tulevat näkyviin jäsennyspuussa eksplisiittisesti. Ne kuitenkin joissain tilanteissa nopeuttavat itse jäsennystä. [8]

10. Parannuksia perusalgoritmiin

10.1 Globaalit ja paikalliset johtonumerot

Edellä esitetyssä perusalgoritmissä esitetään kieliopin säännöt globaalina listana, jossa jokaisella säännöllä on oma yksilöllinen koodinsa. Se ei ole kuitenkaan välttämätöntä, sillä säännöt voidaan esittää yksikäsitteisesti myös siten, että kerrotaan mikä vaihtoehtoisista säännöistä valitaan. Peltola ja Tarhio [9] kutsuvat tällaista jonoa paikallisiksi johtonumeroiksi (local production numbers l. LPN) erotukseksi globaalisista johtonumeroista (global production numbers l. GPN). LPN esitysmuodolla on myös se etu, että sellaiset apumerkit, joilla on vain yksi johto, voidaan jättää kokonaan pois listasta. Myös listan numerot ovat pienempiä kuin GPN-jonossa, jolloin niiden esittämiseen voidaan käyttää vähemmän bittejä. Esimerkiksi aikaisemmin esillä olleessa Pascal-tyyppisessä kieliopissa tarvitsisimme vain lukuja 0, 1 tai 2, sillä ainoastaan expression-säännöillä on kolme erilaista johtoa ja muilla vähemmän. Näin ollen jäsennyspuu voitaisiin koodata käyttäen ainoastaan kolmen bitin koodisanoja esimerkiksi seuraavasti: px -> x ja sx -> x+3 ja aloittaen x:n nollasta. Samoin ensimmäinen sääntö p1 voidaan jättää pois, koska sillä ei ole vaihtoehtoja. Tarvitaan siis 15*3 bittiä = 45 bittiä puun esittämiseen. LPN-jonon etu GPN-esitykseen tulisi vielä voimakkaammin esille, jos sääntöjä olisi enemmän. Tällöin saman koodipätkän esittämiseen tarvittaisiin LPN-esityksessä edelleen sama määrä bittejä, kun GPN-esityksessä lukujen koko kasvaisi ja siten niiden esittämiseen myös tarvittaisiin enemmän bittejä. Olen omassa ohjelmassani ottanut myös käyttöön LPN-jonot.

10.2 Moniselitteiset kieliopit

Katajainen ja muut [8] jättivät omassa versiossaan kieliopista pois operaattoreiden assosioituvuudet ja laskujärjestyksen. Peltola ja Tarhio [9] esittävät lisäksi, että moniselitteisillä kieliopeilla voidaan edelleen tehostaa tiivistystä. Tämä johtuu siitä, että moniselitteisten kielioppien avulla voidaan vähentää jäsennyspuun sisäsolmujen määrää, koska käytännössä apumerkkien johtovaihtoehtojen jakauma on hyvin vino.

Huonona puolena on se, että erilaisten johtojen määrä kasvaa, jolloin myös LPN-luvut voivat kasvaa. Samoin moniselitteisen kieliopin jäsentäminen on hyvin vaikeata ja hidasta, vaikka jonkinlaiset moniselitteisyydet voidaankin käsitellä LR-jäsentelijällä. Koodauksen purku on kuitenkin helppoa myös moniselitteisillä kieliopeilla.

10.3 Mukautuvat kieliopit

Moniselitteisten kielioppien edut ilman suurempia haittoja saadaan kuitenkin esille, kun jäsennyksen aikana lisätään uusia sääntöjä vain silloin kun niistä on etua. Tarkastellaan esimerkiksi seuraavaa kielioppia:

L ::= L; S | S

S ::= i := E

E ::= E +T | E - T | T

T ::= T * F | T / F | F

F ::= (E) | i | n.

Nyt jos koodattavana on seuraavanlainen lauseke: I_1 = i := i +1 * (i - i / n), siitä muodostuvaan puuhun tulee 25 sisäsolmua, jotka kaikki pitää koodata. Jos kielioppiin lisätään säännöt T ::= n, E ::= n ja S ::= I := n, tarvitaan vain 20 sisäsolmua, ja vastaavasti niiden koodaukseen tarvitaan vähemmän bittejä. Peltola ja Tarhio kutsuvat tällaista mallia mukautuvaksi kieliopiksi [9]. Seuraavissa säännöissä on oletettu, että kaikki johdot ovat järjestetty.

Olkoon f(p) laskuri säännölle p, aluksi f(p) = 0. Joka kerta, kun sääntöä p käytetään, kasvaa f(p) yhdellä. Olkoon apumerkillä X on johtovaihtoehdot p_i: X ::= a_i, i = 1, ..., n. Olkoon $F(p_j) = f(p_j)/(\Sigma_i f(p_i)+1)$ ja olkoon vakio c välillä $0 < c < 1$. Jos $F(p_j)$ on suurempi kuin c ja X on säännön oikealla puolella enintään kerran joka johdolla, niin uusi johto q muodostetaan säännöstä p_j. Riippuen oikeasta puolesta a_j, on olemassa kaksi vaihtoehtoa:

1. X ei esiinny a_j:ssä. Olkoot $q_1, ..., q_m$ johdot, joilla on X oikealla puolella ja olkoon $k = \min\{i \mid f(q_i) = r\}$, missä $r = \max\{f(q_1), ..., f(q_m)\}$. Uusi sääntö q muodostetaan soveltamalla sääntöä p_j sääntöön q_k siten, että kaikkein vasemmanpuoleisin X:n esiintymä säännön q_k oikealla puolella korvataan oikealla puolella a_j. Laskurit päivitetään siten, että $f(q_k) := f(q_k) - b$, $f(_q) := b$ ja $f(p_j) := f(p_j) - b$, missä $b = \min\{f(p_j), f(q_k)\}$. Jos p_j tai q_k on jo aikaisemmin lisätty sääntö, niin se poistetaan kieliopista.

2. X esiintyy a_j:ssä. Uusi sääntö q muodostetaan säännöstä p_j korvaamalla kaikkein vasemmanpuoleisin X:n esiintymä a_j:ssä a_j:lla. Esimerkiksi, jos p_j on L ::= L;S eli X on L ja a_j on L;S, niin uusi sääntö on L ::= L;S;S. Laskurit päivitetään siten, että $f(q) := \lfloor f(p_j)/3 \rfloor$ ja $f(p_j) := 0$. Jos p_j on jo aikaisemmin lisätty kielioppiin, se poistetaan sieltä.

Sääntöjä siis myös poistetaan kieliopista, jos niitä ei käytetä riittävän usein. Olkoon säännöllä X vaihtoehdot p_i: X ::= a_i, $i = 1,..., n$, missä p_j on lisätty sääntö. Jos $F'(p_j) = (f(p_j) +1)/(\Sigma_i f(p_i) + 1) < d$, missä vakio d on välillä $0 < d < c/3$, niin p_j poistetaan kieliopista. Muiden sääntöjen laskurit jätetään ennalleen, jottei samaa sääntöä muodostettaisi enää uudestaan. Arvot F(p) ja F'(p) voidaan laskea joko joka koodausaskeleella tai kiinteiden askelmäärien välillä.

Peltola ja Tarhion kokeessa aikaisemmin esitetty I_1 vaatii alkuperäisellä kieliopilla 28 bittiä, kun LPN-sarja koodataan aritmeettisella koodauksella. Mukautuvalla kieliopilla sama lauseke voidaan koodata käyttäen 26 bittiä. (c = 0,6 ja d = 0,1)

10.4 Symbolitaulun organisoimisesta

Ohjelmakoodin jäsentelyn aikana kerätään käyttäjän terminaalit, kuten tunnisteet, vakiot ja muuttujien nimet symbolitauluun. Jäsennyspuun lehdissä sitten viitataan symbolitaulun arvoihin. Todellinen kääntäjä joutuu tallentamaan symbolitauluun paljon sellaistakin tietoa, jota tiivistyksessä ei tarvita. Alkeellisin hukkaavan menetelmän tiivistäjä ei tarvitsisi tallentaa mitään muuta kuin käyttäjän terminaalin tyypin eli sen, onko terminaali esimerkiksi muuttuja, vakio tai jotain muuta.

Käytännössä symbolitauluun täytyy tallentaa myös muuttujien nimet ja kommentit, jotta puretusta ohjelmasta saataisiin tulokseksi mitään järkevää.

Katajainen ja Mäkinen esittävät [7], että symbolitaulua ylläpidettäisiin usealla move to front -heuristiikalla toimivalla listalla. Jokaiselle tyypille olisi oma listansa. Tämä on selvä poikkeus siihen, että kielioppiin perustuvassa tiivistyksessä ei käytetä hyväksi semanttista informaatiota.

Symbolitauluviittaus jäsennyspuussa määrittelee ensin, mitä listaa käsitellään ja sitten sijainnin ko. listalla. Useat listat lisäävät bittien määrä koodisanassa, joka ilmaisee mitä listaa käytetään, mutta Katajainen ja Mäkinen sanovat, että neljä bittiä eli 16 eri listaa pitäisi olla riittävästi. Etuna saadaan se, että koodisana, joka ilmaisee sijaintia listalla lyhenee ja move to front -heuristiikka toimii tehokkaammin, kun käyttäjän terminaalit ovat omissa listoissaan. Yleensä ohjelmointikielissä muuttujien nimet esiintyvät ryhmissä, jolloin move to front -heuristiikalla saadaan hyödynnettyä tätä ominaisuutta. Listojen määrä riippuu käytettävästä kieliopista, joten mukautuvissa kieliopeissa voidaan joutua käyttämään erilaisia tyyppilistoja.

11. Kokeellisia tuloksia

Katajainen ja muut [8] kokeilivat perusalgoritmiä tiivistämällä Pascal-ohjelmia, joskin Pascalin kielioppia oli hieman muutettu mahdollisimman suuren tehokkuuden saamiseksi. Alkuperäisessä versiossa lausekkeet, joissa oli laskutoimituksia, toimivat huonosti, koska kääntäjät tarvitsevat lisätietoa laskutoimitusten laskujärjestyksestä sekä operattoreiden assosioituvuudesta. Tiivistysohjelman ei kuitenkaan tarvitse tietää niistä mitään, vaan riittää, että alkuperäinen ohjelmakoodi voidaan palauttaa purkuvaiheessa. Tiivistysohjelma voi olettaa, että kaikki operaattorit assosioituvat oikealle ja ovat kaikki samanarvoisia. Valitettavasti he eivät kerro minkälaisia Pascal-ohjelmia he tiivistivät.

Alkuperäinen koko	Symbolitaulu	Indeksilista	Tiivistetty koko	Teho (%)
388	100	95	195	50,1
790	152	159	311	39,4
1797	288	451	739	41,1
3435	397	1012	1409	41,0
4254	438	1091	1529	36,0
6265	1360	1351	2711	43,3

Taulukko 11-1 Katajaisen ja muiden tuloksia perusalogoritmillä.

Alkuperäisessä lähtessä [8] menetelmän teho oli laskettu kaavalla $T = \frac{alkuperäinenkoko - tiivistettykoko}{alkuperäinenkoko}$. Taulukossa 11-1 teho on tiivstetyn koon ja alkuperäisen koon suhde, kuten kaikissa muissakin taulukoissa tässä esityksessä. Huomattavaa Katajaisen ja muiden tuloksissa on se, että tiivistyksestä saatava hyöty ei kasvanut kovinkaan suureksi, parhaimmillaankin vain 36,0%. Verratessa tätä perinteisillä menetelmillä saatavaan tiivistykseen, joka on oman kokemukseni mukaan ohjelmakoodeilla noin 30% luokkaa, ei näissä tapauksissa ole saatu minkäänlaista etua kieliopin hyväksikäytöstä. Syynä voi tietysti olla tiivistettävän materiaalin pieni koko, vaikka heidän tuloksissa ei ole nähtävissä tiivistyksen paranemista lähdetiedostojen koon kasvaessa. He myöskin käyttivät kiinteän pituisia koodisanoja, joiden koko oli $\lceil \log(\text{solmujen määrä} + \text{symbolien määrä}) \rceil$. Heidän mukaansa Huffmannin koodauksella ei olisi saatu huomattavia parannuksia koodin tehokkuuteen. Tiivistys oli myös hyvin hidasta. Esimerkiksi 6265 tavun kokoisen

lähdekoodin tiivistykseen kului heillä aikaa 146 sekuntia DEC-2060-tietokoneella. Vertailun vuoksi mainittakoon, että 486 tietokoneeni tiivistää noin 9000 tavun lähdekoodin alle sekunnissa käytettäessä suosittua pkzip-tiivistäjää. Osaltaan syynä on tietysti se, että Katajainen ja muut käyttivät Prolog-kieltä, joka tulkattavana on hieman hitaahkoa.

Peltola ja Tarhio [9] saivat omalla parannetulla tiivistysversiollaan huomattavasti parempia tuloksia, jotka on esitetty taulukossa 11-2. Molemmissa versioissa on tiivistettävänä materiaalina ollut Pascal lähdekoodia, tosin Peltola ja Tarhio ovat käyttäneet oikeita sovelluksia kuten Knuthin TeX-syteemiä sekä omaa tiivistysohjelmaansa. Siksi heidän kokeensa antaa ehkä oikeampia tuloksia. Myös tiivistetyt lähteet ovat huomattavasti suurempia.

Alkuperäinen koodi (bitteinä)	Jäsennyspuu	Tiivistetty tulos (bitteinä)	Teho (%)
2101897	212925	586164	27,9
669865	53452	189101	28,2
262290	18253	73379	28,0
112182	11827	38478	34,3
9408	321	1194	12,7

Taulukko 11-2 Peltolan ja Tarhion tuloksia käyttäen LPN-listaa.

Taulukosta 12-1 näkee hyvin, kuinka voimakkaasti tiivistys paranee käytettäessä LPN-listaa koodauksessa. Peltola ja Tarhio eivät kuitenkaan kerro mitään tiivistykseen käytettävästä ajasta, joten heidän tiivistysohjelmansa todellinen hyöty jää hieman avoimeksi, vaikka tulokset näyttävät paremmilta kuin mitä olisi saatu aikaiseksi perinteisillä tiivistysmenetelmillä.

Näiden tulosten perusteella on kuitenkin mahdotonta sanoa, onko kielioppiin perustuvasta tiivistyksestä todellista hyötyä. Tiivistysteho saattaa olla hieman parempi kuin merkkeihin perustuvilla tiivistysmenetelmillä, mutta suurena haittana on tiivistyksen rajoittuminen vain syntaktisesti oikeisiin lähdekoodeihin. Lisäksi näyttää siltä, että itse koodaus on hyvin hidasta vaikka purku olisikin helppoa ja nopeata.

Viimeaikoina ovat yleistyneet verkon yli ajettavat tulkattavat ohjelmakoodit kuten Java. Tällaisissa tapauksissa näistä tiivistysmenetelmistä voidaan saadaankin kohtuullisen hyviä tuloksia aikaiseksi. Etenkin kun verkon yli haettavassa lähdekoodissa ei tarvita kommentteja eikä merkityksellisiä muuttujien nimiä, joten

tiivistystä voidaan vieläkin kasvattaa. Onko niistä todellista kilpailijaa muille tiivistyskeinoille, jää nähtäväksi.

12. Omia tuloksia ja johtopäätelmiä

Olen kokeillut kielioppiin perustuvaa tiivistystä omalla ohjelmallani, joka tällä hetkellä toteuttaa perusalgoritmin seuraavasti. Ensin tiivistettävä Java-koodi selataan läpi leksikaalisella skannerilla, joka on tuotettu Vern Paxtonin GNU lisenssin alla julkaistulla flex-nimisellä selauskoodin tuottaja ohjelmalla. Flex toimii samalla lailla kuten perus-UNIX työkalu lex, mutta tuottaa hieman nopeampaa C-koodia. Tässä vaiheessa ei kuitenkaan ole vielä käytetty kaikkia optimointimahdollisuuksia, joten ohjelma ei toimi vielä täydellä vauhdilla.

Selausvaiheessa muodostetaan symbolitaulu, jonne tallennetaan kaikki käyttäjän symbolit sekä myös merkintä siitä, onko symboli käyttäjän antama tunnus vai merkkijono literaali. Toistaiseksi tätä tietoa ei kuitenkaan käytetä hyväksi. Kommentit myöskin hukataan kokonaan.

Seuraavassa vaiheessa muodostetaan jäsennyspuu jäsentelijän avulla. Jäsentelijä on tuotettu Bison-nimisellä jäsennyskoodin tuottajalla eli ns. kääntäjä kääntäjällä. Bison tuottaa LALR-jäsentelijän kontekstittomalle kieliopille C-koodina. Bison vastaa UNIX-työkalua YACC, mutta myös se toimii hieman alkuperäistä tehokkaammin. Myös Bison on julkaistu GNU lisenssin alla.

Jäsennysvaiheessa tuotetaan LPN-jono, joka lopuksi yhdessä symbolitaulun kanssa tiivistetään GNU Zip -ohjelmalla, joka käyttää Lempelin - Zivin algoritmia koodauksessaan. Alkuperäinen versio käytti tähän aritmeettista koodausta, mutta muutaman kokeilun jälkeen kävi ilmi, että GNU Zip tuottaa jonkin verran tehokkaampaa koodausta.

Taulukossa 12-1 on tiivistetty muutamia pieniä Java-ohjelmia. Taulukossa alkup. tarkoittaa koodin tiivistämätöntä kokoa; puu jäsennyspuun tiedoston kokoa; taulu symbolitaulun kokoa; lopullinen koko tiivistyksen tulosta sekä ZIP Zip-ohjelmalla tuotettua tiedoston kokoa. Kaikki koot ovat tavuina, joka on tässä järjestelmässä 8 bittiä.

N	Alkup.	Puu	Taulu	Lopullinen	ZIP
1	123	336	42	317	212
2	819	5216	207	675	495
3	5 583	29 500	1 136	2 114	1 1948
4	32 469	197 052	2104	8 437	7 371

Taulukko 12-1 Koetuloksia Java-koodilla.

Taulukosta huomataan, että tämä versio ohjelmasta tuottaa hyvin suuria puita ja siten koko tiivistys tuottaa huonoja tuloksia verrattuna siihen, että koko tiedosto tiivistettäisiin Zip-ohjelmalla. Jatkossa onkin tarkoitus muuttaa kielioppia siten, että kaikki säännöt, joissa oikealla puolella ei ole kuin yksi merkki jätetään pois. Lisäksi symbolitaulun organisoinnilla voidaan ehkä hieman parantaa tulosta, mutta suurin syy ohjelman heikkouteen on juuri puun muodostuksessa.

Ohjelmakoodien tiivistäminen kielioppiin perustuvalla menetelmällä voidaankin siis kuvata kolmella sanalla: rajoittunutta, hidasta ja tehotonta. Rajoittuneisuus verrattuna perinteisiin merkkipohjaisiin tiivistysmenetelmiin tulee esiin hyvin siinä, että ohjelmakoodien täytyy olla kieliopiltaan oikeita ja juuri tiivistäjän ymmärtämää versioita. Lähes kaikki ohjelmointikielten, jopa standardoitujen, kuten C++, toteutukset ovat käytännössä erilaisia, joten tiivistäjä ei todennäköisesti osaa kuin jonkin tietyn valmistajan versiota tiivistettävästä kielestä. Tietysti LEX- ja YACC–työkalujen avulla on melko helppo lisätä uusia kieliä ja murteita tiivistäjän tietämykseen. Silloin tosin tiivistäjän ohjelmakoodin koko kasvaisi huomattavasti ja samoin myös sen tehokkuus heikkenisi, kun kielioppia eikä jäsentäjää voida muokata mahdollisimman tehokkaiksi tiivistyksen kannalta.

Tiivistyksen hitaus ei tietysti ole aina ongelma etenkin, jos tiivistystä käytetään esimerkiksi arkistointiin tai varmuuskopioiden ottamiseen. Kielioppiin perustuva tiivistys ei sellaisenaan kelpaisikaan on-line tiivistysmenetelmäksi esimerkiksi tietoverkkoihin, mutta miksi tyytyä näin hitaaseen menetelmään, kun nopeampia ja tehokkaampia on olemassa.

Kielioppiin perustuva tiivistyksen pitäisi teoreettisesti olla tehokkaampaa kuin pelkkiin merkkeihin perustuva tiivistys, sillä siinä tiivistäjällä on enemmän

informaatiota lähteen rakenteesta. Näin ei kuitenkaan ole näiden kokeiden perusteella. Syy siihen saattaa olla se, että perinteiset menetelmät ovat aikojen saatossa saatu optimoitua erittäin tehokkaiksi ja ne siten pääsevät hyvin lähelle lähteen entropiaa. Kielioppiin perustuvat menetelmät ovat kuitenkin vielä sen verran uusia, että niistä ei saada kaikkea tehoa ulos. Lisäksi on hyvä huomata, että kielioppiin perustuvissa menetelmissä lopullinen tiivistys tapahtuu käyttämällä näitä perinteisiä tiivistysmenetelmiä. Lisäksi kaikki lähdekoodissa esiintyvät käyttäjän terminaalit joudutaan tiivistämään samalla lailla kuin ennenkin. Nopeasti katsoen muutamia Borlandin C++ kääntäjän mukana tulevia esimerkkiohjelmia, näyttäisi käyttäjän terminaalit poislukien kommentit vievän noin 25% lähdekoodin koosta. Kommentit puolestaan voivat olla jopa kaksinkertaistaa koodin koon. Näin ollen kielioppiin perustuva tiivistys voisi saada tehokkuushyötyä vain lopusta 75% koodia.

Kielioppiin perustuva tiivistys ei ole kuitenkaan täysin merkityksetön. Franz [5] on kehittänyt käsitteen *slim-binary* eli kevyt koodi. Nimitys on vastakohta ohjelmakoodille, joka sisältää binaarit monelle eri alustalle ja on siten kooltaan monenkertainen verrattuna tavalliseen yhdelle alustalle tuotettuun koodin. Kevyt koodi toimii kuten Javan p-koodi eli se on osittain käännettyä koodia, joka on tiivistetty käyttäen kielioppiin perustuvaa menetelmää. Koodin koko on alle puolet vastaavasta Javan p-koodista ja itse asiassa tiiviimpää kuin perinteisellä Lempelin - Zivin koodauksella.

Verkottuneessa ja hyvin heterogeenisessa laiteympäristössä, tuleekin kokoajan tärkeämmäksi saada ohjelmointikustannukset alhaisiksi. Yksi tapa on tuottaa sellaista koodia, joka sellaisenaan käy kaikille laitteistoille ja on helppo jakaa verkossa. Tällaisen koodin levitykseen kielioppiin perustuva tiivistys kelpaa hyvin.

Lähdeluettelo

[1] Bell, T. C., Cleary J. G., Witten I. H., *Text Compression,* Prentice Hall, Engelwood Cliffs, 1990.

[2] Cameron, R., Source encoding using syntactic information source model. *IEEE Transactions on Information Theory*, 34, 4, July 1988

[3] Contla, J. F., Compact coding of syntactically correct source programs, Software-Practice and Experience, 15(7), 625-636, July 1985

[4] Davies R. M., Witten I. H., Compressing computer programs, Manuscript.

[5] Franz, M., Adaptic compression of syntax trees and iteratic dynamic code optimization: Two basic technologies for mobile-object systems., *Lecture Notes in Computer Science*, 1222, 263 - 278, Springer Verlag, 1997.

[6] Hirschberg, L. D. S., Data compression, *ACM Computing Surveys*, **19**, 3, September 1987, 261-296.

[7] Katajainen, J. Mäkinen, E., On using type information in syntactical data compression. In *Proceedings of the Third Symposium on Programming Languages and Software Tools*. University of Tartu, Dept. of Computer Science, August 1993, 59-65.

[8] Katajainen, J., Penttonen M., Teuhola, P., Syntax-directed compression of program Files. *Software-Practice and Experience*, **16**, 3 (1986), 269-276.

[9] Peltola, H. Tarhio, J., On syntactical data compression. In: *Proceedings of the Second Symposium on Programming Languages and Software Tools*. University of Tampere, Dept. of Computer Science, Report A-1991-5, August 1991, 205-214.

[10] Sippu, S. Soisalon-Soininen, E. *Parsing Theory*, Springer-Verlag, 1988.

[11] Stone, R. G., On the choice of grammar and parser for the compact analytical encoding of programs*, The Computer Journal*, 29, 4, 1986, 307 - 314.

[12] Storer, J. A., *Data Compression: Methods and Theory*, Computer Science Press, Rockville, 1988