Sisällys

OPINTOJAKSON KUVAUS

Tavoitteet: Kurssin jälkeen opiskelija ymmärtää tietokoneen toimintaperiaatteen loogisten piirien tasolta lähtien. Opiskelija tietää yksityiskohtaisesti miten tietokone pystyy suorittamaan korkean tason ohjelmointikielen komentoja, mutta ymmärtää myös sen kykyjen rajat. Lisäksi opiskelija tuntee ns. tekoälyn piiriin kuuluvien hakuongelmien ja pelaamiseen liittyviä tärkeimpiä strategioita.

Sisältö: Opintojaksolla paneudutaan lyhyesti algoritmisen ongelmanratkaisun voimaan ja rajoituksiin sekä tutustutaan seikkaperäisesti tietokonejärjestelmän (laitteisto ja ohjelmisto) rakenteeseen ja toimintaan. Käsiteltäviä asioita ovat mm. tehtävien laskettavuus ja algoritmien kompleksisuus; tiedon esittäminen, lukujärjestelmät, loogiset piirit ja niiden avulla rakennetut tietokoneen komponentit; mikro-ohjelmointi ja konekieli; kieliopit, kääntäjät ja korkeantason kielenkääntäminen konekielelle. Opintojakson lopuksi käsitellään ns. älykkäissä järjestelmissä käytettyjä hakuongelmia sekä niiden ratkaisua käyttäen mm. syvyys- ja leveyshakua. Lisäksi käsitellään joitakin pelien pelaamiseen liittyviä strategioita (esim. minmax-tekniikka).

Esitiedot: Tietojenkäsittelytieteen perusteet I (TTP I) tai vastaavat tiedot.

Oppimateriaali: Kurssisivuilla olevan opintomonisteen Johdatus tietojenkäsittelytieteeseen s. 79 eteenpäin (monisteen alkuosa käsiteltiin jaksolla TTP I ja se löytyy jakson TTP I sivuilta), opiskeluopas (tämä moniste), verkkomateriaali.

Suoritustapa: Tentti.

Tenttipäivät: Ensimmäinen tentti on 19.10.2016. Katso muut tenttipäivät kurssisivuilta.

YHTEYDET MUIHIN OPINTOJAKSOIHIN

Opintojakso on jatkoa jaksolle Tietojenkäsittelytieteen perusteet I, jossa käytiin läpi opintomonisteen Johdatus tietojenkäsittelytieteeseen kaksi ensimmäistä lukua ja tällä kurssilla käydään läpi monisteen luvut 3-6 sivuuttaen kuitenkin useita opintomonisteen kohtia.

Ohjelmointikurssien jälkeen sinulla on valmiudet tutustua tarkemmin algoritmien ominaisuuksiin (opintomonisteen luvut 3 ja 6). Algoritmien lisäksi tällä jaksolla käsitellään erittäin seikkaperäisesti algoritmeja suorittavan koneen rakennetta ja toimintaa (luvut 4 ja 5), joita käsiteltiin pintapuolisesti ensimmäisen jakson Tietojenkäsittelytieteen perusteet I alussa.

OPPIMATERIAALIT JA KURSSIN SUORITUSTAPA

Opiskeluopas ja verkossa olevat lisäsivut

Tämä opiskeluopas ohjaa opintojakson opiskelua. Opiskeluoppaasta löydät jokaisen viikon oppimateriaalit ja harjoitustehtävät. Jokaisen viikon kohdalla kerrotaan myös viikon tärkeimmät asiat.

Kurssin kotisivuilla on ns. **lisäsivut**, jossa ovat mikro-ohjelmoitavan koneen rakenne ja yhteenveto koneen toiminnoista. Lisäksi siellä on opintomonisteen sivu 133, jossa on konettamme ohjaavien ohjausbittien toiminta. Nämä kaikki **jaetaan tentissä lainaksi**, joten tulosta ne kurssin kotisivuilta itsellesi ja tutustu niihin etukäteen mahdollisimman hyvin, jotta saat niistä tentissä 'kaiken irti'.

Viikon materiaali loppuu sivunumeron ilmaiseman sivun lopussa tai sen tienoilla, jos luku tai esimerkki on kesken. Viikoittainen materiaali ei aina sisällä kokonaista lukua eikä täten muodosta aina yhtenäistä kokonaisuutta. Tähän on kaksi syytä: 1) materiaalin pakkojakaminen luvuittain saa aikaan useimmiten sen, että viikkojen sisällöistä tulee erikokoiset, 2) vaikeampi ja laajempi kokonaisuus saadaan jaettua kahdelle viikolle, jolloin jälkimmäisen viikon asioita on helpompi ymmärtää, koska asiaa on jo käsitelty.

Usein osa viikolle tarkoitetuista harjoitustehtävistä koskee edellisen viikon asioita. Koska asiat ovat jo käsitelty, on kyseinen asia jo paremmin hallinnassa, jolloin opiskelijalla on paremmat edellytykset suoriutua hiukan vaativimmista tehtävistä.

Opintomoniste (ja kohdat joita ei vaadita tentissä)

Tietojenkäsittelytieteen perusteet, joka käsittää opintomonisteen Johdatus tietojenkäsittelytieteeseen sivut 79-206 siten, että seuraavia opintomonisteen kohtia ja asioita **ei käsitellä eikä vaadita tentissä**:

- Luku 3.1.3
- s. 83 – 84 syt-esimerkki
- s. 87 Määritelmä
- s. 90 (Kohdasta "Soveltamalla palautuskaavaa …") – s. 92 (3.2.3 saakka)
- s. 95 T(n):n määrääminen
- Lukuja 3.2.5 (s. 96-97) ja 3.3.2.1 (s. 100)-3.3.2.4 (s. 104) eli luvun 3 loppuun saakka
- s. 110 keskipaikkeilla olevaa matemaattista johtoa
- s. 112 Boolen algebran laskulakeja ei tarvitse osata ulkoa
- Luvun 4.2 (s. 113) alusta aina lukuun 4.2.3 saakka (s. 116). Luvun 4.2.3 alusta tulee ymmärtää käsitteet looginen piiri, portti ja veräjä (ja esim. s. 117 kuvaa ei siis tarvitse ymmärtää).
- s. 119 – 120 esimerkeistä ei vaadita sieventämistä

- s. 126 – 127 esimerkki
- s. 129 – 130 esimerkki
- s. 139 (Kohdasta ”Kertolaskun voi suorittaa nopeamminkin”) – 4.4.5 (s. 141) saakka
- JUMPSUB-käskyn toiminta s. 144 – 146 (mutta s. 145 esimerkki exp(n) vaaditaan)
- s. 151 kohdasta ”Välitöntä osoitusta tarvitaan …” luvun 4.6.3 (s. 153) alkuun saakka
- s. 164 Kappaleesta ”Osittava jäsentäminen” – s. 184
- s. 194 toisesta esimerkistä lähtien monisteen loppuun saakka

Merkitse nämä kohdat heti opintomonisteeseesi. Huomaa kuitenkin, että poisjätetyissä kohdissa on erittäin mielenkiintoisia asioita.

Oheislukemistoa

Opintomonisteen esipuheessa on mainittu kirjoja. Lisäksi jakson kotisivuilla on muutama hyvä linkki. *Internetistä* löytyy myös paljon tietojenkäsittelyn perusteita koskevaa tietoa ja nämä linkit ovat kurssisivuilla.

Opiskeluohje ja suoritustapa

Opiskelusi jakautuu viikoittain itseopiskeluun ja opintoryhmätyöskentelyyn. Lisäksi järjestetään yksi luentopäivä, jolloin opintojakson vastuuopettaja selvittää opintojakson tärkeimpiä ja vaikeimpia asioita.

Viikon teemaan liittyvä oppimateriaali kannattaa lukea ensin huolellisesti kokonaisuutena käyttäen apuna opiskeluoppaan selvennyksiä ja lisäselvityksiä, ja sen jälkeen uudestaan paloittain pohtien samalla osioon kuuluvia harjoitustehtäviä. Harjoitustehtävien tekeminen on hyvin tärkeää. Nimittäin ryhmäkokoontumisten aika ei todellakaan riitä kaikkien asioiden ‘alusta saakka’ käsittelyyn, joten harjoitustehtävien tekeminen etukäteen on välttämätöntä. Näin ollen itseopiskelu on erittäin tärkeätä.

Koska uusi oppimateriaali rakentuu usein jo aiemmin omaksutun oppiaineksen varaan, on syytä huolehtia jo alusta alkaen aikataulussa pysymisestä. Opintoryhmäkokoontumisiin kannattaakin valmistautua huolella etukäteen kartoittamalla opintomateriaalin epäselviksi jääneet kohdat ja miettimällä tarkasti ongelmakohtia koskevat kysymykset.

Opiskeluoppaassa esitetään kunkin viikon kohdalla kohta ‘Alustus’, jossa kerrotaan lyhyesti viikolla käsiteltävät asiat. Sen jälkeen seuraa osio ‘Help’, jossa selvennetään opintomonisteen tekstiä ja esimerkkejä. Nämä kohdat alkavat sivunumerolla, joka kertoo mihin kohtaa opintomonistetta ko. selvennys tai lisäesimerkki kuuluu. Esimerkkien ymmärtäminen on olennaista, koska niiden kautta tulee myös käsiteltävä asia ymmärrettyä. **Lue viikon materiaalista ensin opiskeluoppaan kohta ‘Alustus’ ja siirry sen jälkeen opintomonisteeseen. Opiskeluoppaan ‘Help’-osaa kannattaa lukea rinnan opintomonisteen kanssa. Jokaisen viikon yhteydessä luetellaan viikon tärkeimmät asiat, joten keskity erityisesti näiden asioiden opiskeluun.**

Opintoryhmässä paneudutaan itseopiskelun aikana opittuihin asioihin ja syvennytään erityisesti epäselviksi jääneisiin kohtiin. Ryhmäkokoontumisen tärkeä osa on käydä läpi viikolle tarkoitetut harjoitustehtävät ja opiskelijoiden niihin laatimia ratkaisuja. Jos opintoryhmä on jaettu alaryhmiin (joka on toivottavaa), voi kunkin alaryhmän jäsen esittää ryhmän laatiman ratkaisun. Näitä ratkaisuja pohditaan koko ryhmän kesken ja tärkeään asemaan nousee **palaute**, joka ratkaisusta annetaan. Huomatkaa myös se, että esitettyjen ratkaisujen ei tarvitse olla oikein, vaan tarkoituksena on korjata ratkaisu oikeaksi muiden ryhmäläisten ja tuutorin avustuksella. Harjoitustehtäviä tarkasteltaessa on syytä palata viikon materiaaliin, jotta ratkaisut tulee ymmärrettyä. Opiskelija saa aina myös tehtävien malliratkaisut. **Vaativammat tehtävät on varustettu merkinnällä *, joita ei vaadita tentissä ja ne voi jättää käsittelemättä, jos aika ei riitä.** Niistä annetaan kuitenkin malliratkaisut. Bonuksia laskettaessa näitä ei lasketa tehtävien kokonaismäärään, mutta voit kuitenkin kasvattaa niillä tekemiesi tehtävien yhteismäärää (ja näin ollen aktiivisuus voi olla yli 100% !!!)

Ryhmäkokoontumiset koostuvat aina seuraavista osista:

1. Varmistetaan, että kohdassa ‘viikon tärkeimmät asiat’ esitetyt asiat on ymmärretty.
2. Käydään lävitse viikon harjoitustehtävät.

Opintojakso suoritetaan hyväksytyllä tentillä. Tentissä vaaditaan kaikki jaksolla käsitellyt asiat mukaan lukien tämän opiskeluoppaan materiaali. Tentissä ei saa olla mukana mitään opintojakson materiaaleja, mutta tentissä jaetaan lainaksi mikro-ohjelmoitavaa konetta koskevat lisäsivut.

Läpipääsyyn vaaditaan noin puolet maksimipistemäärästä. Luvusta 4 tulee aina vähintään yksi kysymys ja tietenkin kohdissa ‘Viikon tärkeimmät asiat’ luetellut asiat kannattaa erityisesti lukea. Esseen ja ”algoritmin esittämisen” tms. tehtävän raja on tällä kurssilla epäselvä, koska kysymyksenä voi olla esim. jonkun algoritmin tai yleensä jonkin menetelmän tai laitteen toiminnan selittäminen.

VIIKKO 34

Sisältö: Tehtävän algoritminen ratkeavuus, tehtävän ja algoritmin kompleksisuus ja aikakompleksisuuden kertaluokat, algoritmin oikeellisuus, Hanoin tornit.

Oppimateriaali

Monisteen luku 3, josta ohitetaan edellä mainitut kohdat (näitä ei alla enää toisteta).

Itsenäisen työskentelyn avuksi

Alustus:

Tällä viikolla käsitellään algoritmeja ja niiden ominaisuuksia. Palauta siis mieleesi opintomonisteessa käytetty algoritmien syntaksi. Emme käytä Javan syntaksia, vaan jakson TTP I yksinkertaisempaa, mutta silti riittävän tarkkaa syntaksia. Luku 3 on (paikoitellen) teoreettinen ja käytetty todistustekniikka on useimmille uutta ja outoa. Huomaa kuitenkin, että luku sisältää useita tärkeitä ja mielenkiintoisia menetelmätieteisiin kuuluvia tuloksia. Näin ollen tartu materiaaliin ennakkoluulottomasti ja yritä saada siitä irti mahdollisimman paljon. Suurin osa luvun 3 asioista ohitetaan ja tärkeimmät asiat käyvät ilmi alla olevasta. *Tenttikysymykset luvusta 3 ovat yleisluontoisia ja kohtuullisia.*

Ohjelmoinnin kursseilla opittiin kirjoittamaan algoritmeja Java-kielellä. Tällä viikolla todetaan, että kaikki ohjelmointikielet ja yleiset algoritmien esitystavat ovat algoritmisesti yhtä voimakkaita, ekvivalentteja. Tietenkin tietyntyyppisten tehtävien ratkaiseminen saattaa olla helpompaa jollakin tietyllä kielellä. Lisäksi tällä viikolla todetaan, että on olemassa lukemattomia tehtäviä, joita ei voi ratkaista tietokoneella (joka toimii siis käyttäen nykyisen määritelmän mukaisia algoritmeja), jolloin voidaan siis todistaa, että ei ole ikinä mahdollista kirjoittaa algoritmia näiden tehtävien suorittamiseksi. Tällaisia tehtäviä kutsutaan ei-laskettavissa oleviksi tehtäviksi.

Tämän jälkeen tarkastellaan laskettavia tehtäviä eli sellaisia tehtäviä, joiden ratkaisemiseksi kyetään kirjoittamaan algoritmi. Annettu tehtävä voidaan useimmiten ratkaista usealla tavalla eli voidaan kirjoittaa useita algoritmeja saman tehtävän ratkaisemiseksi. Mikä näistä on paras? Tässä tarkastellaan paremmuuskriteerinä algoritmin suorituksen vaatimien askelten lukumäärää, algoritmin suorituksen vaatimaa aikaa eli aikakompleksisuutta. Arvioitaessa algoritmin aikakompleksisuutta, lasketaan jonkin keskeisen toiminnon suoritusten lukumäärä (esim. laskennallisissa tehtävissä kertolaskujen ja/tai yhteenlaskujen määrä, lajittelussa parittaisten vertailujen lukumäärä), joka antaa hyvän kuvan algoritmin suoritusnopeudesta verrattuna muihin saman tehtävän ratkaiseviin algoritmeihin. Algoritmin todellinen nopeus ajassa riippuu mm. tietokoneen prosessorista ja keskusmuistin koosta. Tietenkin on olemassa muitakin hyvyysmittoja kuten esimerkiksi käytetyn muistin määrä ja algoritmin yksinkertaisuus. Vaikka tehtävän ratkaisemiseen kyettäisiin kirjoittamaan algoritmi,

niin siitä huolimatta algoritmi saattaa olla kelvoton. Tällöin algoritmin suorittaminen kestää niin kauan, että sen suorittaminen ei ole käytännössä mahdollista.

Jos algoritmi ei sisällä rekursiota, voidaan kompleksisuus laskea suoraviivaisesti laskemalla yhteen toimintojen lukumäärä (vrt. opintomonisteen s. 88 vaihtolajittelun aikakompleksisuusanalyysi). Rekursiivisen moduulin[1] tapauksessa päädytään aina palautuskaavaan (esim. T(n)=T(n/2)+n), sillä sisältäähän rekursiivinen algoritmi saman tehtävän ratkaisemisen pienemmällä syötteen (parametrin) arvolla kun alkuperäinen tehtävä. Palautuskaavan ratkaiseminen on yleensä melko vaikeata, joten sitä ei vaadita tällä kurssilla. Tässä yhteydessä esitetään myös erittäin tärkeä ratkaisumetodi: hajota ja hallitse -periaate, jota voidaan soveltaa silloin, kun alkuperäinen tehtävä voidaan jakaa kahteen (noin) yhtä suureen osatehtävään. Tällaisista algoritmeista tarkastellaan esimerkkinä limityslajittelua.

Lopuksi käsitellään hyvin pintapuolisesti algoritmien oikeellisuutta ja algoritmien oikeaksi todistamista, joka on yleisesti ottaen vaikea ei-laskettavissa oleva tehtävä.

Help:

Opintomoniste s. 80. Esitettyjä algoritmien määrittelytapoja ei tietenkään ole tarkoitus ymmärtää eikä niitä vaadita, vaan tarkoituksena on tuoda ilmi, että algoritmi voidaan määritellä usealla - ulkoapäin hyvin erilaiselta näyttävältä - eksaktilla (täsmällisellä) tavalla. Kaikki määritelmät ovat ekvivalentteja eli samanarvoisia siinä mielessä, että kaikilla esitetyillä algoritminen määrittelytavoilla voidaan kirjoittaa algoritmi tarkalleen samojen tehtävien ratkaisemiseksi.

Huomaa, että luvussa 3.1.4 on koottu laskettavuutta koskevat asiat lyhyesti ja ytimekkäästi.

Kompleksisuus. Algoritmin kompleksisuutta määrättäessä tarkastellaan kaikkien tarvittavien resurssien määrästä. Huomaa erityisesti, että *kompleksisuudella ei tarkoiteta algoritmin monimutkaisuutta ja ymmärrettävyyttä* (toisin kuin arkikielessä). Me rajoitamme tarkastelun pelkästään aikaan, jolloin puhutaan aikakompleksisuudesta. Usein tietty algoritmi suorittaa eri määrän askelia eri syötteillä. Esimerkiksi listan suurimman alkion määräämiseen tarvittava vertailujen määrä on suoraan verrannollinen listan alkioiden lukumäärään (tarvittavien vertailujen määrä = listan alkioiden lukumäärä). Toisaalta jos verrataan samaan tehtävään laadittuja eri algoritmeja samalla syötteellä, niin algoritmit saattavat toimia hyvinkin eri tavalla. Tarkastellaan esimerkiksi n-komponenttisen listan lajittelua. Jos lista on jo järjestyksessä, niin useat lajittelumenetelmät huomaavat sen jo ensimmäisellä listan läpikäynnillä, kun taas toiset algoritmit tekevät vertailuja aina saman määrän syötteen ’hyvyydestä’ riippumatta. Täten syöte saattaa vaikuttaa algoritmin suoritusnopeuteen. Sen vuoksi tarkasteltaessa *algoritmin aikakompleksisuutta*, tarkastellaan huonointa tapausta eli tapausta, jossa syöte on mahdollisimman huono algoritmin kannalta. Tällöin voidaan sanoa, että algoritmin suoritusaika on enintään aika, joka kuluu

[1] Tällä jaksolla puhutaan moduuleista eikä metodeista kuten Javan yhteydessä.

huonoimmassa tapauksessa, jota sanotaan algoritmin aikakompleksisuudeksi. Näin ollen em. lajittelumenetelmien aikakompleksisuus on sama (harjoitustehtävänä), vaikka ensin mainittu toimii käytännössä nopeammin, kun lista on 'lähes järjestyksessä'. Annettuun tehtävään (esim. lajitteluun) voidaan kehittää useita algoritmeja ja näistä parhaan algoritmin kompleksisuutta sanotaan ko. tehtävän kompleksisuudeksi. Erota siis käsitteet **tehtävän kompleksisuus** ja yksittäisen **algoritmin kompleksisuus**.

s. 85. Esimerkiksi $q(x)=5x^4+x^2+6$ on neljännen asteen polynomi (n=4, a_4=5, a_3=0, a_2=1, a_1=0, a_0=6), ja esimerkiksi $q(2)=5x^4+x^2+6=80+4+6=90$. Esimerkiksi q(2):n laskemiseksi tulee moduulia P kutsua seuraavasti: P(6,0,1,0,5,2). Tässä ei kuitenkaan tarvitse ymmärtää polynomin määritelmää, vaan tarkastella vain s. 85-86 moduuleja ja niissä olevia laskutoimituksia: + ja *.

s. 86-87. Kun määrätään algoritmin aikakompleksisuus T(n), on saatu lauseke yleensä monimutkainen ja saattaa sisältää esimerkiksi summalausekkeen, jossa on useita termejä. Tällöin se pyritään esittämään yksinkertaisemmassa muodossa eli yksinkertaisemman funktion f(n) avulla, mutta kuitenkin niin, että yksinkertaisempi muoto f(n) on likimain T(n):n suuruinen, kun n on suuri. Tällöin sanotaan, että ne ovat asymptoottisesti samaa **suuruusluokkaa** ja tällöin merkitään T(n)~f(n). Tällöin siis T(n) ja f(n) ovat samaa suuruusluokkaa, kun n on suuri. Esimerkiksi jos tarkastellaan opintomonisteen s. 86 taulukkoa, niin siitä nähdään, että esim. n^2 ja $n^2+5n+3n$ ovat samaa suuruusluokkaa, kun n on esim. 1 miljoona. Käsite 'samaa suuruusluokkaa' määritellään tarkasti (jota siis ei tarvitse osata) opintomonisteen s. 87: T(n) ja f(n):n suhde vaihtelee vain kiinteän n:stä riippumattoman vakion verran olipa n kuinka suuri tahansa.

Miksi olemme kiinnostuneet vain suurista syötteistä? Kun syöte on pieni, niin silloin mikä tahansa algoritmi toimii nopeasti. Sen sijaan kun syöte on suuri, niin silloin algoritmien tehokkuudella alkaa olla jo eroa. Sen vuoksi meille riittää tieto siitä, mikä on aikakompleksisuuden T(n) asymptoottinen suuruusluokka.

Alla oleva tulos voidaan todistaa perustuen s. 87 määritelmään, mutta me otamme sen vain kätevänä faktana.

Yleisesti voidaan osoittaa, että jos $T(n)=a_kn^k+a_{k-1}n^{k-1}+...+a_1n^1+a_0$, missä kaikki kertoimet a_i ovat reaalilukuja ja ensimmäinen kerroin $a_k>0$, niin silloin $T(n) \sim n^k$

Näin ollen polynomisen lausekkeen suurusluokan määrää sen korkein potenssi ilman termin kerrointa. Esimerkiksi $100n^2+50n \sim n^2$ ja $45n^2+70n^2 \sim n^2$, vaikka pienillä n:n arvoilla ko. lausekkeet ovat suuruudeltaan hyvin erilaisia. Asymptoottinen yhtäsuuruus tarkoittaakin sitä, että kun n on hyvin suuri, niin lausekkeet ovat vain n:stä riippumaton vakioarvon päässä toisistaan eli samaa suuruusluokkaa.

Sivun 86 ylälaidan esimerkeissä esitettiin algoritmi ja laskettiin algoritmissa suoritettujen kerto- ja yhteenlaskujen lukumäärä. Jos tarkastellaan algoritmin

asymptoottista aikakompleksisuutta, niin silloin tulee ensin miettiä, mikä on algoritmissa suoritettava keskeinen operaatio. Kertolaskuja suoritetaan enemmän kuin yhteenlaskuja, joten on luonnollista tarkastella vain niiden asymptoottista käyttäytymistä (lisäksi myöhemmin todetaan, että kertolaskun suorittaminen on raskaampi toimenpide kuin yhteenlasku). Ensimmäisessä algoritmissa suoritetaan $T_1(n)=2+3+...+(n+1)$ kertolaskua. Tämä on aritmeettinen sarja (kahden peräkkäisen termin erotus on aina vakio), jolle on olemassa yleinen summakaava. Ns. induktioperiaatteella voidaan osoittaa, että

$$1+2+3+...+n = n(n+1)/2 = \frac{n^2+n}{2},$$

joka on ~ n^2.

Induktioperiaate on erittäin tärkeä todistustekniikka, jota käytetään matemaattisten kaavojen sekä tietynlaisten algoritmien oikeellisuuden ja kompleksisuuden todistamisessa. Induktioperiaatetta ei vaadita tällä kurssilla, mutta se on kuvattu seikkaperäisesti luvussa 3.3.2.1. Huomaa, että asymptoottista arvoa määrättäessä ei tarkka arvo n(n+1)/2 ole tärkeä, vaan se että ko. lauseke on ~ n^2. Edellä laatikossa olevan kaavan nojalla $T_1(n)\sim n^2$.

s. 88. Tarkastellaan seuraavaksi logaritmeja (*ei vaadita* tenteissä), jotka tulevat usein esille algoritmien kompleksisuusanalyyseissä. Yleensä jos tehtävän koko puolittuu jossain algoritmin vaiheessa (kuten s. 90 limityslajittelussa), niin se saa aikaan logaritmilausekkeen kompleksisuuden lausekkeeseen. Tietojenkäsittelytieteessä tarkastellaan yleensä 2-kantaista logaritmia, kun taas koulussa käsitellään yleistä k-kantaista logaritmia ja erityisesti tapauksia k=10 tai k=e=2.7182…= Neperin luku (tällöin puhutaan luonnollisesta logaritmista, jota merkitään usein: ln). Tämä johtuu siitä, että 'sattuneista syistä' tietojenkäsittelytieteessä kakkosen potenssit ovat hyvin tärkeitä. Yleinen k-kantainen logaritmi määritellään seuraavasti:

$\log_k(n)=m$ silloin ja vain silloin kun $n=k^m$ (k potenssiin m).

Yleensä logaritmin kantaluku k kirjoitetaan sanan log eteen yläindeksiksi tai perään alaindeksiksi (kuten yllä). Jos käsitellään samankantaisia logaritmeja, voidaan kantaluku k jättää kirjoittamatta, jos se on tiedossa. Näin tehdään oppimateriaaleissamme (k=2).

1) Näin ollen 2-kantaiselle logaritmille pätee seuraava:

$\log_2(n)=m$ silloin ja vain silloin kun $n=2^m$.

Siis esimerkiksi $\log_2(1)=0$, $\log_2(2)=1$, $\log_2(4)=2$, $\log_2(8)=3$, $\log_2(16)=4$, $\log_2(32)=5$ jne. Yleisesti: $\log_2(2^x)=x$. Esimerkiksi $\log_2(32)=5$, koska $2^5=32$.

2) 10-kantaiselle logaritmilla laskettaessa $\log_{10}(1)=0$, $\log_{10}(10)=1$, $\log_{10}(100)=2$, $\log_{10}(1000)=3$ jne.

Huom. Kun seuraavassa (ja myös opintomonisteessa) kirjoitetaan log, tarkoittaa se aina 2-kantaista logaritmia $\log_2$. Lisäksi merkinnästä jätetään usein sulkeet pois ja kirjoitetaan lausekkeen log(n) sijasta log n tai logn.

(*** logaritmien käsittelyn loppu ***)

s. 88. ylälaidan esimerkki **vaihtolajittelun** aikakompleksisuus. Algoritmi (TTP I-moniste s. 61) koostuu kahdesta sisäkkäisestä silmukasta. Uloin silmukka käy arvot i=1, ... , n−1.

Kun i=1, suoritetaan sisempi silmukka n−1 kertaa.
Kun i=2, suoritetaan sisempi silmukka n−2 kertaa.
...
Kun i=n−2, suoritetaan sisempi silmukka 2 kertaa.
Kun i=n−1, suoritetaan sisempi silmukka 1 kertaa.

Kaiken kaikkiaan suoritetaan siis T(n)=1+2+3+...+(n−1) vertailua. Näin ollen

$T(n) = 1+2+3+...+(n-1) + n - n$

$= n(n+1)/2 - n$

$= n(n-1)/2$

$= \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$, joten

$T(n) \sim n^2$.

s. 88. taulukko: a=vuosi, d=päivä, h=tunti, s=sekunti, ms=millisekuntti=10^{-3}s, μs=mikrosekuntti=10^{-6}s. Taulukon lukujen tarkoitus on havainnollistaa eri kompleksisuusluokkien eroja. On tietenkin selvää, että algoritmin vaatiman todellisen ajan määrään vaikuttaa kellotaajuuden lisäksi monet muut asiat; esim. välimuistin (cache) määrä, keskusmuistin koko, muistin käsittelyn nopeus (väylän leveys),….

s. 90 puoliväli - s. 91 ei tarvitse lukea. Tässä osoitetaan, että limityslajittelun aikakompleksisuus on n*logn, joka on siis parempi kuin esim. kuplalajittelun ja vaihtolajittelun aikakompleksisuus, joka on n*n.

Opintomonisteen s. 92 esitetään klassinen *Hanoin tornien* ongelma, joka esitetään useimmissa alan oppikirjoissa. Ongelman ratkaiseva algoritmi toimii loistavana esimerkkinä modulaarisuuden, rekursion ja parametrien voimasta vaikean tehtävän ratkaisemisessa. Itse moduulista tulee hyvin lyhyt ja se todella toimii. Lisäksi tämä on ainoa ratkaisumenetelmä, jos ei sallita turhia siirtoja! Tehtävä ratkaistaan käyttämällä rekursiota, mutta tietenkin tästä voidaan kirjoittaa iteratiivinen ratkaisu, joka matkii rekursiivista ratkaisua. Itse asiassa jokaisesta rekursiivisesta ratkaisusta voidaan tehdä vastaava iteratiivinen ratkaisu – tietokonehan ei ole rekursiivinen!

S. 94 esitetyissä algoritmeissa *Hanoi* on ehkä harhaanjohtavaa käyttää tolppien niminä Lähtö, Maali ja Apu, koska näiden nimien looginen merkitys (siis se mikä on lähtö-, maali- ja aputolppa) pätee kyllä moduulin otsikossa, mutta ei enää moduulin sisällä olevissa moduulin Hanoi kutsuissa. Sen vuoksi olisi ehkä selvempää identifioida tolpat vain nimillä X, Y ja Z. Tällöin algoritmi saa muodon:

```
MODULE Hanoi(n,X,Y,Z)
(* Toiminta: siirtää n kiekkoa X:stä Y:hyn käyttämällä apuna tolppaa Z *)
   IF n>0 THEN
      Hanoi(n-1,X,Z,Y)
      Siirrä X:stä Y:hyn kiekko numero n
      (* eli siirrä X:n päällimmäinen kiekko Y:hyn)
      Hanoi(n-1,Z,Y,X)
   ENDIF
ENDMODULE
```

Saamme siis samoja osatehtäviä, mutta **tolppien roolit (lähtö, apu, maali) ovat erilaiset** eri osatehtävissä ja lisäksi **kiekkojen määrä pienenee** osatehtävissä. Roolien muuttaminen ja kiekkojen vähentäminen ratkeaa näiden asioiden **parametrisoinnilla.** Parametrina olevien tolppien (2., 3. ja 4. parametri) **looginen merkitys** (eli rooli: lähtö-, maali- tai aputolppa) tehtävää suoritettaessa **ilmaistaan** niiden **järjestyksellä** parametrilistassa: siirretään tolpasta (2. parametri) tolppaan (3. parametri) käyttäen apuna tolppaa (4. parametri). Tällöin esim. kutsu
Hanoi(n-1,Z,X,Y) tarkoittaa, että siirrä n-1 kiekkoa tolpasta Z tolppaan X käyttäen apuna tolppaa Y. Vaikka yleensä muuttujat pyritään nimeämään niin, että nimi kuvaa muuttujan sisältöä, niin tässä se ei johda hyvään tulokseen. Tämä johtuu siitä, että muuttujat (tolpat) ovat eri rooleissa algoritmin eri suoritusvaiheissa. Algoritmin toimintaa on havainnollistettu myös Villessä, mutta se ei juuri auta, ellei moduulin idea ole auennut.

Sivulla 95 lasketaan moduulin aikakompleksisuus, jota ei siis vaadita tentissä. Siinä tarvittava matematiikka on lukiotasoa ja se kannattaa lukea läpi.

Viikon tärkeimmät asiat

Tehtävän laskettavuus, tehtävän ja algoritmin aikakompleksisuus, asymptoottinen aikakompleksisuus, aikakompleksisuuden kertaluokat, vaihtolajittelun aikakompleksisuus, hajota ja hallitse –periaate, Hanoin tornit.

Harjoitustehtävät

1. Mikä on seuraavien algoritmien aikakompleksisuus T(n) ja mikä on tarkasteltava alkeisoperaatio. Anna myös aikakompleksisuuksien asymptoottinen suuruusluokka.

   a) n-kertoman palauttava moduuli (TTP I-opintomoniste s. 45).
   b) n-kertoman palauttava rekursiivisen moduulin (TTP I-opintomoniste s. 53).
   c) n-alkioisen lukujonon maksimin määrääminen (TTP I-opintomoniste s. 33).
   Ohje: a) ja b): kerto- ja yhteenlaskujen määrä. c): Kompleksisuutta määrättäessä tarkastellaan algoritmin kannalta huonointa syötettä (järjestystä).

2. Edellä tarkasteltiin vaihtolajittelun (TTP I-moniste s. 61) aikakompleksisuutta. Mikä on TTP I-opintomonisteen s. 35 lajittelualgoritmin (algoritmi, jossa on muuttuja *vaihtoja*) aikakompleksisuus ja sen suuruusluokka.
   Ohje: Tässä tulee siis tarkastella huonointa syötettä.
   Mitä tämä kertoo näiden kahden lajittelualgoritmin hyvyydestä käytännössä? Miten syöte (eli missä järjestyksessä alkiot ovat) vaikuttaa algoritmien tehokkuuteen.

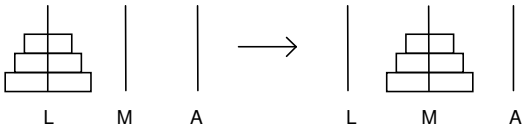
3. Määrittele seuraavalle moduulille alku- ja loppuehdot (lopputila) sekä lisää jokaisen IF-lauseen perään kommentti, joka ilmaisee, mitä ko. lauseen suorituksen jälkeen tiedetään x:n, y:n ja z:n keskinäisestä järjestyksestä. Moduulin vaihda[2] kutsu vaihda(a,b) vaihtaa muuttujien a ja b sisällöt keskenään.

```
MODULE järjestä(x, y, z)
        IF y < x THEN vaihda(x, y) ENDIF
        IF z < y THEN vaihda(y, z) ENDIF
        IF y < x THEN vaihda(x, y) ENDIF
ENDMODULE
```

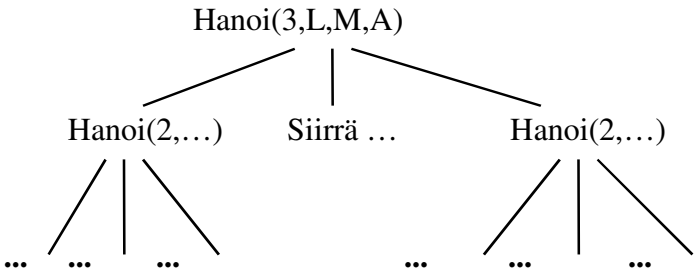
[2] Muistanet OOP-jaksolta, että Javalla ei tällaista metodia vaihda voi kirjoittaa, elleivät a ja b ole mutatoituvaa oliotyyppiä. Vaihto voidaan kuitenkin aina hoitaa kyseisessä kohtaa kolmella asetuslauseella.

4. -5. Esitetään Hanoin tornien ongelma graafisesti seuraavaan tapaan (kolmen kiekon tehtävä):

Hanoi(3,L,M,A)



Tässä L, M ja A edustavat todellisia parametreja kutsussa Hanoi(3,L,M,A). Havainnollista kolmen kiekon tehtävän ratkaisua kuvaamalla yo. tavalla kaikki alkuperäisen kutsun Hanoi(3,L,M,A) generoimat osatehtävät (ja niiden generoimat osatehtävät jne.) moduulin Hanoi mukaisesti.
Ohje: Pura ensin kutsun Hanoi(3,L,M,A) toiminnot moduulin mukaisesti rekursiivisesti. Tämä saa aikaan kaksi kahden kiekon tehtävää (eli kaksi moduulin Hanoi kutsua, jossa 1. parametrin arvo on 2) ja yhden kiekon siirron. Tätä havainnollistaa alla olevan kuvan toinen rivi. Kumpikin kahden kiekon tehtävä saa taas aikaan kaksi yhden kiekon tehtävää (eli kaksi moduulin Hanoi kutsua, jossa 1. parametrin arvo on 1) ja yhden kiekon siirron, jne. Piirrä näistä hierarkkinen rakenne[3], jossa ylimpänä on kutsu Hanoi(3,L,M,A), josta lähtee alaspäin kolme viivaa (kaarta), joiden ulommaiset osat sisältävät moduulin Hanoi kutsut ja keskellä on yhden kiekon siirto. Jokaisesta kutsusta lähtee taas kolme viivaa alaspäin jne. Kun olet saanut kutsut purettua, kirjaa siirrot oikeassa järjestyksessä (esim. alla suoritetaan vasemman puoleinen Hanoin kutsu loppuun, sen jälkeen Siirrä … ja lopuksi oikeanpuoleinen Hanoin kutsu loppuun).



[3] Tästä tulee itse asiassa (3-haarainen) puu (ks. TTP I-opintomoniste)

VIIKKO 35

Sisältö: Tietokoneen matemaattiset ja fysikaaliset perusteet, loogiset portit.

Oppimateriaali

Opintomoniste s. 105-117.

Itsenäisen työskentelyn avuksi

Alustus:

Tällä viikolla tarkastellaan tietokoneen matemaattisia ja fysikaalisia perusteita. Opintojen alussa esitettiin tietokoneen rakenne ja toimintaperiaate sekä tietokoneessa käytetty tiedon esitysmuoto pääpiirteissään. Nyt käymme nämä asiat läpi perusteellisesti piiritasolla.

Vaikka tietokoneen toiminta perustuu tiedon binääriesitykseen (2-järjestelmään), niin tässä käsitellään ensin yleisesti lukujärjestelmiä. Nimittäin on tärkeätä ymmärtää eri lukujärjestelmien väliset yhteydet. Sen jälkeen tarkastellaan kokonaislukujen ja reaalilukujen esitystä tietokoneessa. Kokonaisluvut esitetään käyttäen ns. *kahden komplementtiesitystä* ja reaaliluvut käyttäen apuna ns. mantissaa ja eksponenttia.

Ennen kuin siirrytään fysikaalisiin ominaisuuksiin, esitetään matemaattinen struktuuri Boolen algebra, koska kaikki *tietokoneen piirit voidaan ymmärtää Boolen algebran lausekkeina*. Näin ollen ’piirejä voidaan manipuloida matematiikan avulla’. Fysikaalisten perusteiden esittely aloitetaan opintomonisteessa melkoisen alhaalta tasolta: atomeista ja elektroneista. Esitettyjä (s. 113-116) fysikaalisia perusteita ei voi ymmärtää ellei omaa vähintään lukion tasoista fysiikan osaamista. Asioita ei vaadita tentissä, mutta ne kannattaa lukaista läpi, jotta saisit jonkinlaisen käsityksen tietokoneen komponenttien rakennusosista ja niiden fysikaalisista perusteista.

Help:

s. 106. k-kantaisen lukujärjestelmän luvun tulkinta on samanlainen kuin tuttu 10-järjestelmän luvun tulkinta. Esimerkiksi desimaaliluku 123.45_{10} tarkoittaa lukua $1*10^2$ (sadat) + $2*10^1$ (kymmenet) + $3*10^0$ (ykköset) + $4*10^{-1}$ (kymmenesosat) + $5*10^{-2}$ (sadasosat).

s. 107. Huomaa, että **muunnoskaava** on kirjoitettu ko. sivun esimerkissä näkyviin, jotta kyseiset muunnokset voitaisiin suorittaa myös oikealta vasemmalle. Tarkastellaan sivun 107 esimerkin toista esimerkkiä:

$20120_3 = 2\cdot3^4 + 0\cdot3^3 + 1\cdot3^2 + 2\cdot3 + 0 = (((\underline{2}\cdot3 + \underline{0})\cdot3 + \underline{1})\cdot3 + \underline{2})\cdot3 + \underline{0} = 177_{10}$

Tässä ensimmäinen yhtälö $20120_3 = 2\cdot3^4 + 0\cdot3^3 + 1\cdot3^2 + 2\cdot3 + 0$ saadaan suoraan s. 106 määritelmän perusteella, josta tulos 177_{10} voidaan jo suoraan laskea. Toista yhtäsuuruutta $2\cdot3^4 + 0\cdot3^3 + 1\cdot3^2 + 2\cdot3 + 0 = (((\underline{2}\cdot3 + \underline{0})\cdot3 + \underline{1})\cdot3 + \underline{2})\cdot3 + \underline{0}$ on jo vaikeampi keksiä suoraan itse, mutta se saadaan suoraan s. 107 muunnoskaavasta. Tämä antaa algoritmin (s. 107 alaosa) sille, miten 177 muunnetaan 3-järjestelmään: jaetaan lukua toistuvasti kolmella, jolloin jakojäännökset antavat tuloksen (ks. 108 ensimmäinen esimerkki). Näin ollen tässä (eikä muissakaan esimerkeissä) ei ole tarkoitus se, että osaa suoraan kirjoittaa yhtälön $(((\underline{2}\cdot3 + \underline{0})\cdot3 + \underline{1})\cdot3 + \underline{2})\cdot3 + \underline{0} = 177_{10}$, vaan tällöin muunnos tehdään peräkkäisillä jakolaskuilla.

Palataan hetkeksi viime viikon asioihin. Hanoin tornien yhteydessä s. 95 osoitettiin (tätä ei vaadita) että siirtojen määrä on $T(n)=2^{n-1}+2^{n-2}+\ldots+2^1+2^0$. Tällöin todettiin näppärästi että $T(n)=2^n+1$, joka perusteltiin 2-järjestelmän lukujen tulkinnan perusteella (ks. s. 95 alaviite).

Miten muunnokset eri lukujärjestelmien välillä kannattaa suorittaa?

1. Muunnettaessa k-järjestelmän lukua 10-järjestelmään, on se helpointa tehdä käyttäen suoraan s. 106 määritelmää.
2. Muunnettaessa 10-järjestelmän lukua k-järjestelmään, kannattaa muunnos tehdä peräkkäisillä jakolaskuilla s. 106-107 taitteen mukaisesti.
3. Muunnos k-järjestelmän ja k^j-järjestelmän välillä kannattaa tehdä j numeron lohkoissa s. 108 keskellä olevan esimerkin mukaisesti.
4. Muunnettaessa k-järjestelmän lukua s-järjestelmään ($k\neq10$, $s\neq10$ ja ei ole kyseessä kohdan 3 tapaus) kannattaa muunnos suorittaa 10-järjestelmän kautta.

s. 109 tarkastellaan esimerkinomaisesti kolmea erilaista **kokonaislukujen binäärisiä esitystapaa** lähtien liikkeelle puhtaasta binääriesityksestä, ottamalla sen jälkeen huomioon etumerkki (etumerkki+itseisarvo esitys) päätyen lopulta kahden komplementtiesitykseen, joka on näistä ainoa tapa, jolla voidaan esittää sekä positiiviset että negatiiviset luvut niin, että yhteen- ja vähennyslasku toimivat oikein. Näin ollen kahden komplementtiesitys on kelvollinen kokonaislukujen esitystapa tietokoneessa ja niinpä sitä useimmiten käytetäänkin. **Tästä eteenpäin oletamme, että kokonaisluvut esitetään käyttäen kahden komplementtiesitystä[4], ellei erikseen mainita, että tarkastelemme puhdasta binääriesitystä.** Huomaa, että kahden komplementin binäärilukuja (ja myös puhtaita binäärilukuja) voidaan laskea yhteen, vähentää ja kertoa allekkain samalla (koulusta tutulla) tavalla kuin 10-järjestelmän lukuja, mutta tällöin 0+0=0, 1+0=0+1=1, 1+1=0 ja muistiin 1, 0–0=0, 0–1=1 ja lainataan 1, 1–0=1, 1–1=0, 0*0=0*1=1*0=0, 1*1=1, 1+1+1=1 ja muistiin 1.

[4] ja tällöin esim. 1111_2 ei ole 15_{10} vaan -1_{10}

s. 110 määritelmä. Huomaa, että positiivisen 10-järjestelmän kokonaisluvun m kahden komplementtiesitys on sama kuin luvun puhdas binääriesitys, mutta **negatiivisen** luvun −m esitys määritellään samaksi kuin positiivisen kokonaisluvun $-m+2^n$ puhdas binääriesitys. Siis myös negatiivisten lukujen kahden komplementtiesitys on helppo määrätä suoraan käyttäen puhdasta binääriesitystä. Käytännössä negatiivisen luvun −m kahden komplementtiesitys muodostetaan vastaavan positiivisen luvun m_{10} puhtaasta binääriesityksestä käyttäen opintomonisteessa mainittua menetelmää: vaihdetaan m:n esityksessä nollat ykkösiksi ja päinvastoin ja lisätään tähän binärilukuun 1. Huomaa kuitenkin, että tämä on ns. peukalosääntö, joka on todistettu oikeaksi käyttäen s. 110 määritelmää. Yleisesti voidaan todeta, että jos 10-järjestelmän luvun m kahden komplementtiesitys on muotoa $(m_{n-1}\ m_{n-2}\ ...\ m_0)_2$, niin silloin $m_{n-1}=0$, jos $m\geq 0$ ja $m_{n-1}=1$, jos $m<0$.

Opintomonisteen s. 110 todistuksessa (ei vaadita) käytetään aiemmin esiintynyttä kaavaa: $\sum_{i=0}^{n-1} 2^i = 2^n - 1$. Todistuksesta nähdään myös, että sääntöä voidaan käyttää myös kääntäen: negatiivisen luvun kahden komplementtiesityksestä saadaan vastaavan positiivisen kokonaisluvun kahden komplementtiesitys samalla säännöllä.

Tehtävä (käsitellään opintoryhmässä vain jos jää aikaa): Tarkista seuraava edellistä yksinkertaisempi peukalosääntö 4-bittisille (pätee myös yleisesti) kahden komplementin kokonaisluvuille käyttäen s. 109 taulukkoa. Olkoon x positiivinen kokonaisluku väliltä -8…+7, jolloin x:n puhdas binääriesitys ja kahden komplementin esitys ovat samoja. Silloin negatiivisen luvun −x kahden komplementtiesitys saadaan x:n puhtaasta binääriesityksestä seuraavasti: Kopioidaan x:n bitit oikealta käsin ensimmäiseen ykköseen asti se mukaan lukien sellaisenaan, ja siitä vasemmalle olevat bitit komplementoidaan. Huomaa, että tämäkin sääntö pätee myös kääntäen: negatiivisen luvun kahden komplementin esityksestä saadaan vastaava positiivinen luku samalla säännöllä.

s. 110. Kun kaksi lukua, jotka esitetään käyttäen kahden komplementtiesitystä, lasketaan yhteen, niin yhteenlaskun tulos ei ole mitä tahansa, vaikka tulos ylittäisikin lukualueen eli tapahtuu ns. *ylivuoto*. Nimittäin **tulos on oikein modulo 2^n** (so. saatu tulos eroaa oikeasta tuloksesta 2^n:n moninkerran verran) eli tulokset ovat kongruentteja modulo 2^n. Kongruenssin käsite on hyvin tärkeä matematiikassa, mutta meillä riittää ymmärtää, että kaksi lukua x ja y ovat kongruentteja modulo m (merkitään: $x\cong y \pmod m$), jos niiden erotus on m:n moninkerta. Esimerkiksi $9\cong 1 \pmod 4$ ja myös $\pmod 2$, $32\cong 16\cong 0 \pmod{16}$. Seuraavaksi havainnollistetaan esimerkillä sitä, että ylivuodon sattuessa yhteenlaskun tulos on oikein modulo 2^n. Olkoon n=4 ja lasketaan (−5)+(−6) käyttäen kahden komplementtilukuja. Oikea tulos on −11, jota ei voida esittää nelibittisillä luvuilla, joten tapahtuu ylivuoto:

−5 = 1011
−6 = 1010
$0101_2 = 5_{10}$ (viimeisen yhteenlaskun $1+1=10_2$ muistinumero 1 vuotaa yli).

Nyt $-11 \cong 5 \pmod{16}$ eli saatu tulos on oikein mod 16. *Ylivuoto* tarkoittaa siis sitä, että tulos ei ole arvoalueella ja tulos on siis väärin. Huomaa, että jos ylin (eli vasemman

puoleisin) bitti vuotaa yli, niin tulos saattaa silti olla oikein (ks. tämän viikon tehtävän 4 lasku 6+(-3)). Ylivuoto ei siis tarkoita sitä, että ylin bitti vuotaa yli laskettaessa laskua esim. allekkain tai käyttäen myöhemmin rakennettavaa yhteenlaskulaitetta.

s. 110 loppu: excess-koodaus. Jos esim. n=4, jolloin lukualue on –8…7, niin 10-järjestelmän luvun excess-8 koodaus tarkoittaa sitä, että siihen lisätään ensin 8, ja sen jälkeen siitä muodostetaan tavallinen binääriluku. Esim. –3 excess-8 koodattuna on luvun +5 binääriesitys 0101. Jos taas n=3, niin lukualue on –4…3 ja tällöin puhutaan excess-4 koodauksesta. Ks. monisteen s. 109 taulukko.

s. 112. **Boolen algebra.** Huomaa s. 113 alussa oleva teksti, joka kertoo yhteyden mm. ohjelmointiin.

s. 113. **Lisäesimerkki**. Boolen kaava voidaan todistaa oikeaksi 1) **totuustaulukolla** tai 2) soveltamalla Boolen algebran laskulakeja. Osoitetaan, että kaava x+xy=x pätee kaikilla x:n ja y:n arvoilla.

1) *Todistus käyttäen totuustaulukkoa*: Muodostetaan totuustaulukko, jossa sarakkeina ovat x ja y sekä todistettavassa kaavassa esiintyvät lausekkeet.

x	y	xy	x+xy
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Tässä x:lle ja y:lle annetaan arvoiksi kaikki mahdolliset kombinaatiot ja lasketaan lausekkeen x+xy arvo (taulukossa on apusarake xy:tä varten). Vertaamalla nyt x:n ja x+xy:n sarakkeita, huomataan, että ne ovat identtiset. Siis x=x+xy.

2) *Todistus käyttäen Boolen algebran laskulakeja*: Periaatteessa kysymyksessä on ‘koulusta tuttu’ lausekkeiden sievennys, mutta nyt käytettävät laskulait ovat hieman poikkeavat. Tarkoituksena on siis sieventää lauseke (x+xy) yksinkertaisempaan muotoon. Todistettava kaava on hyvin yksinkertainen ja itse asiassa se on yksi Boolen algebran laskulaeista (absorptiolaki). Näytämme kuitenkin miten se voidaan todeta oikeaksi käyttäen sen yläpuolella (s. 112) olevia kaavoja. Kirjoitetaan selvyyden vuoksi × näkyviin, koska lausekkeissamme esiintyy arvo 1. Lisäksi kirjoitamme näkyviin kaikki mahdolliset välivaiheet.

$x + x \times y = x \times 1 + x \times y$ (ykkösalkio)

$= x \times (1 + y)$ (distributiivisuus eli voidaan ottaa 'tekijä')
$= x \times 1$ (koska $1 + y = 1$)
$= x$

Monimutkaisempiin sievennyksiin palataan ensi viikolla.

(*** lisäesimerkin loppu ***)

s. 113-116. Vaikka tätä osiota ei tentissä vaaditakaan, niin siitä on hyvä ymmärtää ainakin seuraavat asiat:

1. **Diodi** on komponentti, joka johtaa sähköä toiseen suuntaan, mutta eristää toiseen suuntaan.
2. **Transistori** on elektroninen kytkin, jolla voidaan estää tai sallia virran kulku.
3. Diodien ja transistorien avulla voidaan rakentaa **loogisia portteja (veräjiä)**, jotka toteuttavat tietyn totuusarvoisen funktion. Ns. perusportit ovat: NOT, AND, OR, NAND, NOR, XOR ja EQV (jotka esitellään opintomonisteen sivulla s. 117)
4. Perusporttien avulla voidaan rakentaa hyvinkin monimutkaisia **loogisia piirejä**, jotka toimivat haluamallamme tavalla.

s. 118. Huomaa, että loogisille porteille pätee lisäksi seuraavat **kaavat** (ei tarvitse osata ulkoa):

$$x \text{ NAND } y = \overline{xy} = \bar{x} + \bar{y}$$
$$x \text{ NOR } y = \overline{x + y} = \bar{x}\,\bar{y}$$
$$x \text{ XOR } y = x \oplus y = \bar{x}\,y + x\,\bar{y}$$
$$x \text{ EQV } y = \overline{x \oplus y} = \bar{x}\,\bar{y} + x\,y$$

Viikon tärkeimmät asiat

Lukujärjestelmät, kokonaislukujen (kahden komplementti) ja liukulukujen esitys tietokoneessa, Boolen algebran laskulakien merkitys, loogiset portit ja niiden symbolit.

Harjoitustehtävät

1. a) Muuta 2-järjestelmän luku 1011.01 10-järjestelmän luvuksi.

   b) Muuta 10-järjestelmän luku 34.4 2-järjestelmän luvuksi (huom. ei pääty).

2. Suorita (allekkain) seuraavat laskutoimitukset puhtaalla binääriaritmetiikalla:

   a) 10101010 − 10001111 (ei saa palauttaa yhteenlaskuun vaan pitää vähentää)

   b) 10101×11011

3. Oletetaan, että liukuluvut tallennetaan 8 bitillä:

   1. bitti: Etumerkkibitti (0 = ei-negatiivinen luku, 1 = negatiivinen luku),

   Bitit 2-4: 3-bittinen eksponentti excess-koodattuna (lisäys +4),

   Bitit 5-8: 4-bittinen mantissa (itseisarvo).

   a) Miten koodataan 10-järjestelmän luku 1.25 ?

   b) Mitä kymmenjärjestelmän lukua edustaa bittijono 10101010 ?

4. -5. Tarkastellaan yhteenlaskuja 6 + 3, (−6) +3, 6 + (−3) ja (−6) + (−3), kun luvuille käytetään kahden komplementin 4-bittistä esitystä. Suorita kyseiset yhteenlaskut allekkain ja pohdi sitä milloin tulos on oikein ja minkälaisen säännön voimme muodostaa tuloksen oikeellisuuden testaamiseksi.
   Ohje: Vaikka tuloksen ylin bitti vuotaa vasemmalta yli, niin lasku voi silti olla oikein! Tuloksen oikeellisuus nähdään summattavien ja tuloksien ylimmistä biteistä (jotka siis kertovat luvun etumerkin).

6. Osoita De Morganin laki $\overline{x+y} = \bar{x}\,\bar{y}$ (s. 112) oikeaksi totuustaulukon avulla.
   Ohje: Ks. edellä ollut kaavan x + xy = x todistus.

VIIKKO 36

Sisältö: Tietokoneen komponentteja: loogiset piirit, yhteenlaskulaite, vähennyslaskulaite, yhden bitin muistava laite eli kiikku, rekisterit, väylät, kello.

Oppimateriaali

Opintomoniste s. 118-129.

Itsenäisen työskentelyn avuksi

Alustus:

Aluksi käsitellään loogisia piirejä, joiden looginen toiminta tulee ymmärtää, vaikkakin piirien toimintaa fyysisellä tasolla ei tarvitse osata. Loogisia perusportteja yhdistelemällä voidaan rakentaa loogisia piirejä moniin eri tarkoituksiin. Kuten aiemmin jo todettiin, loogiset piirit voidaan esittää Boolen lausekkeina, joita voidaan sieventää käyttämällä Boolen algebran laskulakeja. Viime viikolla käsiteltiin yksi hyvin yksinkertainen sieventämistehtävä ja tällä viikolla esitetään (mutta ei vaadita) muutama mutkikkaampi esimerkki lausekkeiden sieventämisestä. Sieventämisen tarkoitus on ilmeinen: matematiikkaa käyttäen voidaan tarvittavan 'raudan' määrää vähentää. Sen jälkeen käsitellään tietokoneen peruskomponentit, jotka voidaan rakentaa loogisilla piireillä. Näitä ovat aritmetiikasta huolehtiva laite, muisti, väylät (tieto-, ohjaus- ja osoiteväylä), loogisen vertailun suorittava laite ja kello, joka tahdistaa tietokoneen eri komponenttien toiminnan. Näistä esitetään piiritasolla: yhteenlaskulaite, vähennyslaskulaite, kiikku eli yhden bitin muistava laite, kiikuista muodostettu rekisteri ja väylät ja niiden liitäntä tietokoneen muihin komponentteihin.

Help:

Boolen algebran lauseke vastaa siis loogista piiriä. **Piirin sieventämisellä** tarkoitetaan lausekkeen esittämisestä yksinkertaisemmassa muodossa siten, että siinä on mahdollisimman vähän ns. perusportteja, joita ovat: NOT, AND, OR, NAND, NOR, XOR ja EQV. Voidaan osoittaa, että kaikki nämä perusportit voidaan esittää käyttämällä vain NAND-portteja (tai vastaavasti NOR-portteja), joka tarkoittaa sitä, että jokainen tietokoneen piiri voidaan rakentaa käyttäen vain NAND-portteja (NOR). Näin kone on halvempi rakentaa, koska tarvitaan vain yhdenlaisia 'rakennuspalikoita'. Kuitenkin ensin tarkasteltava lauseke kannattaa sieventää.

s. 119 esimerkki: **f:n lausekkeen muodostaminen totuustaulukon nojalla**. *Miksi riittää tarkastella vain totuustaulukon niitä rivejä, joissa f=1*, kun muodostetaan f:n lauseketta f(x,y,z)? Ensimmäinen rivi, jolla f saa arvon 1, saadaan (x,y,z)-kombinaatiolla (0,1,1) eli kun x=0, y=z=1. Mutta mikä on sellainen lauseke, joka saa arvon 1 tällä (x,y,z)-kombinaatiolla ja arvon 0 kaikilla muilla (x,y,z)-kombinaatiolla. Selvästi tällainen lauseke on $\bar{x}yz$. Vastaavasti seuraavia taulukon 1-rivejä vastaavat lausekkeet ovat $x\bar{y}z$ ja $xy\bar{z}$. Kun nämä lausekkeet yhdistetään OR-operaatiolla (eli +) saadaan lauseke, joka saa arvon 1 ainoastaan (x,y,z)-kombinaatiolla (0,1,1), (1,0,1) ja (1,1,0) ja arvon 0 kaikilla muilla (x,y,z)-kombinaatiolla. Saatu lauseke on etsimämme ja se voidaan piirtää opintomonisteessa esitetyllä tavalla. Piiri on kuitenkin melko monimutkainen, joten sitä voidaan yksinkertaistaa (eli vähentää loogisten veräjien määrää) sieventämällä saatua Boolen lauseketta käyttämällä Boolen algebran laskulakeja.

Huom. Boolen lausekkeiden sieventämistä ei vaadita tentissä. Lisäksi Boolen algebran laskulakeja ei tarvitse osata ulkoa.

s. 120 esimerkki. Huomaa, että johdettu sievennetty tulos voidaan tarkistaa helposti totuustaulukon avulla.

Boolen lausekkeiden sieventämisessä käytetään yleensä seuraavia strategioita (ei vaadita)**:**

1. Otetaan jokin lauseke (yleensä muuttuja) tekijäksi siten, että toinen tekijä olisi helposti sievennettävissä (esim. yllä toinen tekijä on $z+\bar{z}$, jonka arvo on 1).
2. Käytetään distributiivisuuslakeja.
3. Käytetään De Morganien lakeja.

s. 121 **Puolisummain**. Tarkastellaan sivun alalaidan taulukkoa, joka kuvaa kahden bitin yhteenlaskua. Piirin syötteinä ovat x ja y sekä tulosteina s ja m. Muodostettaessa m:n lauseketta, voidaan taulukosta valita se ainoa rivi, jolla m saa arvon 1. Tällöin x=y=1, eli m=xy. s saa arvon 1 kahdella (x,y)-kombinaatiolla eli $s = x\bar{y} + \bar{x}y = x \oplus y$ (ks. yllä oleva teksti miksi riittää tarkastella vain niitä rivejä, joissa ulostulon arvo on 1).

s. 122-123. Tässä rakennetaan **kokosummain** eli yhteenlaskulaite n-bittisille luvuille. Huomaa, että piiri matkii koulusta tuttua allekkain laskua. Luvut esitetään käyttäen kahden komplementtiesitystä, jolloin n:llä bitillä voidaan esittää luvut ovat väliltä -2^{n-1} ... $2^{n-1}-1$. Esimerkiksi 4 bitillä voidaan esittää luvut −8 … +7. Ylivuodon sattuessa tulos on oikein modulo 2^n (katso myös s. 109).

s. 123. Koska a−b=a+(−b), voidaan **vähennyslasku** toteuttaa käyttäen yhteenlaskulaitetta ja vastalukuoperaatiota. Huomaa, että tässä voi a ja b olla myös negatiivisia. Vastalukuoperaatio voidaan taas toteuttaa muuttamalla tarkasteltavan

luvun bitit (0↔1) ja lisäämällä saatuun bittijonoon 1 tai käyttämällä viime viikon tekstin (lisä)tehtävässä annettua sääntöä. Nytkin ylivuodon sattuessa tulos on oikein modulo 2^n riippumatta siitä kumpaa (vastaluvun avulla tai kokovähentimellä) menetelmää vähennyslaskun suorittamiseen käytetään.

Huomaa, että s. 124 ylhäällä havainnollistetaan vain binäärilukujen allekkain suoritettua vähennyslaskua. Kun luvut tulkitaan kahden komplementin binääriluvuiksi, niin tulos ei ole oikein (1101=-3, 0111=7 ja 0110=6) ja se ei voikaan olla oikein, koska oikeata tulosta -10 ei voida esittää neljällä bitillä. Tulos on nytkin oikein mod 16. Vähennyslaskussakin voidaan kehittää viime viikon tehtävän 4-5 tapaan säännöt, joilla voidaan helposti todeta, milloin tulos on oikein. Näissäkin oikeellisuuden ratkaisee lukujen ylimmät bitit, jotka kertovat luvun merkin. Kokeileppa!

s. 124 **Kiikku**. Muistiominaisuus saadaan aikaan NOR-porttien ristiinkytkennällä. **Piirissä johdot x ja y ovat aina tietyssä tilassa riippuen syötteistä D ja C. Sen sijaan M:n tila ei ole välttämättä stabiili eli muuttumaton**, koska se voi jäädä 'heilahteleen' tilojen 0 ja 1 välille johtuen ristiinkytkennästä. Ohjausosa on kuitenkin rakennettu niin, että näin ei pääse tapahtumaan.

Kiikun toiminta on seuraava: Jos C=0, niin kiikku muistaa bitin M (siis M:n arvo ei muutu olipa D mikä tahansa). Jos C=1, niin muistiin tallentuu databitin D arvo riippumatta siitä, mitä muistissa oli ennen (siis olipa M:n vanha arvo mikä tahansa).

Kontrollibitti C kertoo sen onko kyseessä bitin M suojaaminen (C=0) vaiko databitin D tallennus (C=1) muistiin M.

Tarkastellaan seuraavaksi, että piirimme toimii juuri edellä kuvatulla tavalla eli s. 125 olevaa totuustaulukkoa. Alla oleva päättely on mekaaninen ja tätä tapaa voi soveltaa kaikkien piirien yhteydessä. Tietenkin tervettä järkeä kannattaa käyttää ja näin tulee tehdä myös tämän kiikun yhteydessä. Nimittäin kiikun toiminta voidaan selvittää myös suoraan, kuten monisteen s. 126 ylhäällä todetaan (lue tarkkaan!).

1. **Muista bitti M.** *Tällöin tulee asettaa C=0.* Nyt x=y=0 olipa D 0 tai 1. Tällöin M:n tila ei muutu, joka nähdään katsomalla kahta peräkkäistä M:n tilaa. Jos

   - M:n vanha arvo M_t=0, niin silloin ylemmän NOR-portin ulostulo on 1, joka menee sisään alempaan NOR-porttiin, josta tulee ulos 0=M_{t+1}.
   - M_t=1, niin vastaavasti kuin edellä päätellään, että M_{t+1}=1.

   Näin ollen kummassakin tapauksessa piiri on stabiili eli M:n arvo ei muutu. Tällöin piiri siis **muistaa** bitin M.

2. **Tallenna M=0**. *Tällöin tulee asettaa C=1 ja D=0.* Silloin x=0, y=1. Jos

- M:n vanha arvo M_t on 0, niin M_{t+1}=0 eli M:n tila ei muutu. Piiri on siis heti stabiili.
- M:n vanha arvo M_t on 1, niin M muuttuu tilaan 0 eli M_{t+1}=0. Nyt M:n arvo muuttui, joten meidän tulee varmistua, että piiri on stabiili eli että M pysyy tilassa 0 (muutenhan se jää heilahtelee tilojen 0 ja 1 välille). Näin ollen meidän tulee katsoa vielä M_{t+2}. Edellisen nojalla kuitenkin M:n arvo 0 säilyy, koska M_{t+2}=0. Piiri on siis stabiili.

Näin ollen M menee tilaan 0 riippumatta sen vanhasta tilasta. Tällöin voidaan tallentaa databitti 0 muistiin M.

3. **Tallenna M=1.** *Tällöin tulee asettaa C=1 ja D=1.* Silloin x=1, y=0. Tällöin M menee tilaan 1 riippumatta sen vanhasta tilasta. Tässä todetaan ensin, että jos M_t=1, niin M_{t+1}=1 eli piiri on heti stabiili. Sen sijaan tila M_t=0 johtaa ensin tilaan M_{t+1}=1 ja tämä tila säilyy, koska M_{t+2}=1. Tällöin voidaan siis tallentaa databitti 1 muistiin M.

s. 128-129. **Dekooderi.** CPU:ssa oleva data tarkoittaa siis CPU:n jonkin rekisterin sisältöä ja R ja W tulkitaan kyseisen rekisterin R ja W kontrolleiksi.

s. 129 kuvan esimerkissä muistipaikan M13 edessä olevan dekooderin tulos saa olla 1 vain silloin, kun CPU:n osoitebitit (’a niin kuin address’) ovat: a3=1, a2=1, a1=0, a0=1, ja dekooderin tulos pitää olla 0 kaikilla muilla bittikombinaatioilla. Kaikkien muiden muistipaikkojen dekooderit rakennetaan vastaavasti, jolloin kaikkien muiden muistipaikkojen dekooderien tulos on 0 kombinaatiolla a3=1, a2=1, a1=0, a0=1. Jotta saisimme aikaan em. toiminnon, tulee muistipaikan M13 dekooderi rakentaa seuraavasti: osoiteväylän syöte viedään AND-porttiin niin, että bittiä 0 vastaavat syötteet komplementoidaan ennen AND-porttiin viemistä. Näin ollen syötettä a1 (ainoa nolla) vastaavan johdon sisältö komplementoidaan ennen dekooderiin syöttöä ja muut viedään sellaisenaan dekooderiin.

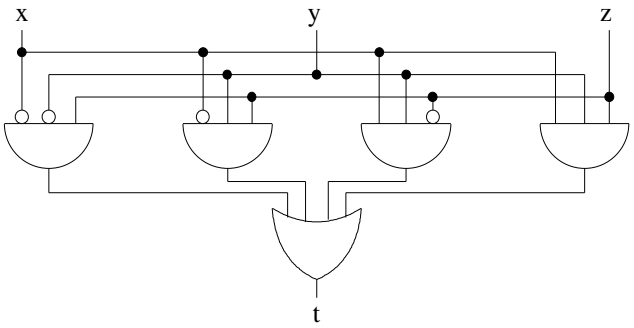
Viikon tärkeimmät asiat

Loogisten piirien konstruoiminen, yhteenlaskulaite, vähennyslaskulaite, kiikku eli yhden bitin muistava laite, kiikuista muodostettu rekisteri ja väylät ja niiden liitäntä tietokoneen muihin komponentteihin (siis lähes kaikki viikon asiat).

Harjoitustehtävät

1. Havainnollista 4-bittisen yhteenlaskulaitteen toimintaa laskulla −6 + 3 kirjoittamalla 'johtojen arvot' yhteenlaskulaitteeseen kuten edellä tehtiin. Huomaa, että luvut esitetään käyttäen kahden komplementtiesitystä.

2. *[5]Seuraavassa on annettu eräs looginen piiri. Tehtävänä on suunnitella yksinkertaisempi piiri, joka toteuttaa saman loogisen funktion (toiminnan). Piirrä myös sievennetty piiri.
   Ohje: Muodosta ensin piiriä vastaava Boolen lauseke ja sievennä se käyttäen s. 104 kaavoja.



3. - 4. Konstruoi nelibittinen kokovähennyslaskulaite ja esitä, miten se laskee vähennyslaskun 4 − 7 kirjoittamalla bitit laitteen johtoihin.
   Ohje: Mieti miten lainaus kulkeutuu; vrt. myös muistinumeron kulkeutuminen yhteenlaskulaitteessa.

5. Selitä kiikun toiminta tarkasti kahdessa tapauksessa: 1) piiri muistaa bitin M=1, 2) M:ään tallennetaan 0. Tässä tulee siis kertoa C:n, D:n, x:n ja y:n arvot ja osoittaa, että piiri toimii kohdissa 1) ja 2) kuvatulla tavalla.
   Huom. Tämä on vanha tenttitehtävä!

[5] Tämä on tähtitehtävä (ks. tämän oppaan s. 6)

VIIKKO 37

Sisältö: **Mikro-ohjelmoitava tietokone ja sen ohjelmointi.**

Oppimateriaali

Opintomoniste s. 130-139 puoliväliin saakka.

Itsenäisen työskentelyn avuksi

Alustus:

Tällä viikolla ‘rakennetaan’ hypoteettinen mikro-ohjelmoitava tietokone. Tämä osa kurssista on hyvin käytännönläheinen. Koneemme näyttää aluksi monimutkaiselta (vaikka se on hyvin pelkistetty ja alkeellinen) ja vaatiikin erityistä tarkkaavaisuutta, sisältäähän kone useita yksityiskohtia. Kokonaisuuden ymmärtäminen vaatii useiden yksityiskohtaisten asioiden hallitsemista, joten koko asian ymmärtäminen ei käy kädenkäänteessä. Senpä vuoksi sinun tulee tässä vaiheessa jaksaa lukea asiaa eteenpäin, vaikka kokonaisuus ei tunnu hahmottuvan. Kun kokonaisuus alkaa hahmottua, huomaat, että koko asia ei ole laisinkaan vaikea. Nimittäin tässähän on tarkoituksena kertoa tietokoneen toimintaperiaate, ja koska tietokone on laite, joka suorittaa mekaanisesti peräkkäin yksinkertaisia käskyjä, ei koneen toiminnan ymmärtäminen ole ylipääsemättömän vaikeata, eikä ainakaan teoreettista! Se vaatii kuitenkin aikaa ja pitkäjännitteisyyttä.

Jo nyt kannattaa kertoa mihin pyrimme. Tavoitteena on esittää kaikki ne vaiheet, joita tarvitaan korkean tason kielisen ohjelman suorittamiseksi tällä viikolla esitetyllä mikro-ohjelmoitavalla tietokoneella. Tarvitsemme tässä tietenkin useita välivaiheita, joita käsitellään seuraavien viikkojen aikana. Aluksi siirrymme kuitenkin suoraan hyvin matalalle tasolla eli tarkastelemme mikro-ohjelmoitavaa tietokonettamme. Kuva tietokoneestamme on esitetty kurssisivuilla olevassa liitteessä. Kuvaa on tietenkin mahdoton ymmärtää tässä vaiheessa, mutta vilkuile sitä aika ajoin lukiessasi seuraavaa materiaalia.

Aluksi harjoittelemme mikro-ohjelmien kirjoittamista keksittyjen esimerkkien avulla. Käytännössä mikro-ohjelmamuistissa on vain ns. mikro-ohjelmoitu tulkki, joka on mikro-ohjelma, joka osaa suorittaa päämuistissa olevaa konekielistä ohjelmaa. Siis meidän tulee **kirjoittaa tulkki vain kerran**, joka ‘poltetaan’ mikro-ohjelmamuistiin. Tämän jälkeen voimme unohtaa mikro-ohjelmoinnin, koska koneen ohjelmointiin käytetään sen jälkeen korkeamman tason kieliä.

Kone ja sen toiminta lyhyesti

Seuraavassa on aika paljon samoja asioita kuin opintomonisteessa mutta usein hiukan tarkemmin.

Koneessamme on useita rekistereitä (nopeita muisteja tiedon väliaikaiseen tallennukseen), kolme dataväylää, ohjausväylä, päämuisti (RAM), mikro-ohjelmamuisti (ROM), yhteenlaskulaite kahdelle yhteenlaskettavalle, vastalukuoperaation sekä ns. shiftleft-operaation (siirretään bittejä yksi askel vasemmalle, joka vastaa kahdella kertomista) suorittava piiri. Koneen sanankoko on 16 bittiä, jolloin dataväylät, päämuisti ja (useimmat) rekisterit ovat 16 bittisiä. Ohjausväylä on 22 bittinen, jolloin myös mikro-ohjelman säilytyspaikka MPM on 22-bittinen muisti. Mikro-ohjelma koostuu MPM:ssä peräkkäin olevista mikrokäskyistä, joista kukin on 22 bitin pituinen bittijono. Kukin mikrokäsky viedään aina vuorollaan erityiseen rekisteriin (MIR), joka on liitetty tietokoneen kelloon. Koneen ohjelmointi on hyvin alkeellista, sillä yksi mikrokäsky on 22 ohjausbitin jono c1,...,c22, jotka ohjaavat koneen piirien toimintaa. Sivulla 133 on jokaisen ohjausbitin vaikutus, jos kyseinen bitti on tilassa 1. Lisäsivulla on yhteenveto koneen toiminnoista kellon eri vaiheissa.

Lue ensin monisteen s. 130-134 ja sitten alla oleva, joka tarkentaa ko. tekstiosuutta (tai näitä voi lukea myös rinnakkain).

Help:

s. 133-134 Yhteenveto hiukan tarkemmin. Pähkinänkuoressa koneemme toiminta on seuraava: kello 1 ja 2 voidaan suorittaa tietynlainen asetuslause eli asettaa jokin arvo rekistereihin MAR, MDR, A, B, C tai D. Kello 3 on varattu päämuistin käsittelyyn (tiedon siirto päämuistista MM rekisteriin MDR tai päinvastoin). Kello 4 väylälle 3 (DC3) tulee saada seuraavaksi suoritettavan käskyn osoite mikro-ohjelmamuistissa (MPM). Tämä saadaan aikaan siirtämällä sopivat arvot väylille 1 ja 2, joiden summa tulee väylälle 3. Kello 5 tapahtuu **automaattisesti** aina seuraavaa: DC3:n sisältö viedään mikro-ohjelmalaskuriin MPC ja MPM:n ko. muistipaikassa oleva käsky siirretään mikrokäskyrekisteriin (MIR). Rekistereiden sekä rekistereiden ja päämuistin välillä ei voi suoraan siirtää tietoa, vaan tieto siirretään ensin väylille 1 ja 2 (DC1 ja DC2), joiden sisällöt menevät aina yhteenlaskulaitteen läpi (siis lasketaan aina yhteen). Tiedot kulkevat kellosyklin aikana aina kahdesti yhteenlaskulaitteen läpi: kellon vaiheissa 1 ja 2 sekä kellon vaiheessa 4. Jos väylille 1 tai 2 ei siirretä mitään, niin silloin niillä oletetaan olevan arvo nolla. Yhteenlaskulaitteen tulos voidaan siirtää väylältä 3 (DC3) mihin tahansa rekisteriin (useampaankin yhtä aikaa), mutta ei suoraan päämuistiin. Päämuistin käsittely tapahtuu aina seuraavalla tavalla: 1) asetetaan ensin rekisteriin MAR sen muistipaikan osoite, joka halutaan lukea/kirjoittaa päämuistista/päämuistiin, ja 2) tieto tallennetaan päämuistiin ja luetaan päämuistista aina rekisterin MDR kautta. Koska koneemme rekistereiden ulostulot ovat kytketty väyliin 1 ja 2, jotka menevät aina yhteenlaskulaitteen läpi, tulee kaikki ohjelmassa tarvittavat arvot (esim. luvun tallennus rekisteriin, tiedon siirto rekistereiden välillä ja osoitteen tallennus MAR:iin) muodostaa summaamalla. Esimerkiksi arvon nolla asetus rekisteriin A tehdään seuraavasti: asetetaan väylille 1 ja 2 arvo 0 (eli ei viedä väylille

mitään), jotka kulkevat yhteenlaskulaitteen läpi ja yhteenlaskun tulos viedään rekisteriin A.

Lue sen jälkeen lisäsivu (tiivistelmä koneen toiminnasta, joka on kurssisvuilla) vertaamalla sen tekstiä yllä olevaan alustukseen ja opintomonisteen tekstiin. Huomioi tässä seuraava huomautus.

Huomaa, että lisäsivulla käytetyt merkintätavat (esim. 1+B→B) *eivät ole ohjelmointikieltä*, vaan merkinnät kertovat, että asettamalla ohjausbitit sopivasti saadaan aikaan toiminta, jonka vaikutus koneeseen on ko. merkinnän mukainen. Esimerkiksi 1+B→B tarkoittaa seuraavaa: luku 1 viedään väylälle 1 (DC1) ja rekisterin B sisältö viedään väylälle 2 (DC2). Niiden välissä on +, koska väylän DC1 ja DC2 sisällöt viedään aina yhteenlaskulaitteen läpi. Yhteenlaskun tulos (väylän DC3 sisältö) viedään lopuksi rekisteriin B. *Lisäsivun tarkoituksena on kertoa lyhyesti miten konetta voi ohjelmoida eli minkä tyyppisiä komentoja (tiedon siirtoja) se osaa suorittaa* ja lisäksi annetaan merkintätavat näille komennoille ja kerrotaan millä ohjausbiteillä toiminto saadaan aikaan (käsinkirjoitetut ohjausbittien numerot). Näiden merkintöjen muuttaminen kontrollibiteiksi on suoraviivaista ja muunnos voidaan tehdä lisäsivun mukaan (voisimme tehdä helposti esim. Java-ohjelman, joka lukee kyseisiä merkintätapoja käyttävää ohjelmakoodia ja muuntaa ne vastaaviksi bittijonoiksi eli siis tietynlaisen kääntäjän!). Esimerkiksi −1+B→B vastaa bittiyhdistelmää: c5=1 (luku 1 väylään 1), c2=1 (B:n sisältö väylään 2), c7=1 (muutetaan väylän 1 sisältö eli 1 vastaluvukseen), c10=1 (yhteenlaskun −1+B tulos väylältä 3 viedään rekisteriin B) ja kaikki muut kellon 1 ja 2 vaiheiden bitit = 0. Periaatteessa s. 133 ohjausbittejä koskeva informaatio riittää, jotta osaisimme ohjelmoida tietokonettamme. Näin yksityiskohtainen tieto kuitenkin hämärtää itse ohjelmointiprosessia, joten on helpompaa ajatella itse ohjelmointiprosessia symbolisessa muodossa käyttäen lisäsivun mukaisia merkintöjä.

Lue s. 135 alku ja kaksi ensimmäistä yksinkertaista esimerkkiä ja sen jälkeen alla oleva lisäesimerkki.

Lisäesimerkki. Seuraavassa on, ennen s. 135-137 laajempaa esimerkkiä, kaksi hyvin yksinkertaista toimintoa (asetuslausetta), joista on jätetty MPC:n käsittely pois.

1) *Aseta rekisteriin A arvo nolla*. Tämä saadaan aikaan toiminnalla 0+0→A ; ; Tällöin kellon vaiheessa 1 tulee muodostaa 0+0. Koska väylillä 1 ja 2 on oletusarvoisesti nollat, niin kellon vaiheessa 1 ei tarvitse tehdä mitään, mutta kellon vaiheessa 2 tulee summan 0+0 arvo siirtää rekisterin A. Näin ollen asetetaan c9=1 ja muut bitit nolliksi.

2) *Kopioi rekisterin B sisältö A:han*. Tämä saadaan aikaan käskyllä 0+B→A ; ; eli asetetaan c2=c9=1 ja muut bitit nolliksi. Huomaa, että koska väylien 1 ja 2 sisällöt kulkevat aina yhteenlaskulaitteen läpi, niin tässäkin täytyy väylällä 1 olla arvo nolla, vaikka kyseessä on vain rekisterin B kopiointi rekisteriin A (siis asetuslause).

(*** lisäesimerkin loppu ***)

Kurssisivuilla on ohjelma **Simo**, joka **simuloi** konettamme ja sen toimintaa. Siinä voi kirjoittaa mikrokäskyjä, suorittaa ne ja katsoa miten rekisterien sisältö muuttuu. Sinun kannattaa kuitenkin tutustua seikkaperäisesti koneeseemme ennen kuin alat käyttää tätä ohjelmaa. Harjoitustehtävissä pitää yksi tehtävä tehdä käyttäen Simoa.

Siirrytään seuraavaksi s. 135-137 esimerkkiin. Huomio myös s. 137 lopussa oleva selostus ohjausbittien numeroinnista. Esimerkki on kirjoitettu hyvin seikkaperäisesti ja siitä käy hyvin ilmi mikro-ohjelman muodostamiseen tarvittavat näkökohdat ja huomiot. Lue se huolella. Aluksi algoritmi kirjoitetaan korkealla tasolla ja sen jälkeen mietitään, mitä siinä pitää muuttaa, että siitä saataisiin mikro-ohjelma. Tällöin tulee ymmärtää millaisia toimintoja koneemme osaa suorittaa. Sivun 126 keskellä algoritmi esitetään käyttäen lisäsivulla esitettyjä merkintöjä, jotka on suoraviivaista muuttaa binääriseksi mikro-ohjelmaksi. Kellon 5 sykliä on jaettu koneen toiminnan logiikan mukaan kolmeen vaiheeseen, joissa kussakin voi tapahtua jokin tietty toimintakokonaisuus:

- *vaihe 1*: kello 1 ja 2, jolloin suoritetaan asetuslause,
- *vaihe 2*: kello 3, jolloin tapahtuu muistinkäsittely, ja
- *vaihe 3*: kello 4 ja 5, jolloin määrätään mikä käsky suoritetaan seuraavaksi.

Nämä kolme vaihetta erotetaan lisäsivun mukaisessa symbolisessa esityksessä toisistaan *puolipisteillä* ja kirjoitetaan näkyviin, vaikka ko. vaiheessa ei tehtäisi mitään. Esimerkiksi ohjelman 1. käskyn vaiheessa 2 ei tehdä mitään, koska tällöin ei suoriteta päämuistin käsittelyä. Vaiheessa 3 määrätään siis seuraavan käskyn osoite, joka viedään MPC:hen. Jos ohjelmassa edetään peräkkäisesti, tulee MPC:n arvoa lisätä yhdellä (1+MPC→MPC). Tämä saadaan aikaan asettamalla c17=c22=1. Ehdollinen yhden lauseen ylihyppy voidaan suorittaa tutkimalla rekisterin A sisältöä bittien c19 ja c22 avulla. Jos taas halutaan suorittaa *ehdoton hyppy* (käskyt 5 ja 9), tulee seuraavan käskyn osoite siirtää MPC:hen (tämä on kuvattu s. 135).

Huom. Opintomonisteen s. 134 alalaidassa sanotaan, että kullekin väylälle siirretään yhden rekisterin sisältö kerrallaan. Hyppylauseen kohdalla tilanne on toinen. Ajatellaanpa, että haluamme kirjoittaa mikro-ohjelmaan hyppylauseen kohtaan 14. Tällöin meidän tulee asettaa c1...c8=00001110, jolloin c5=c6=c7=1. Tällöin väylälle 1 yritetään siirtää sekä luku 1 että rekisterin MDR sisältö samaan aikaan. Miten koneemme toimii tässä tapauksessa? Meille riittää tieto siitä, että kyseinen käsky 'ei kaada konettamme', koska tässä tapauksessa (hyppylause) kellon ajanjaksoina 1 ja 2 tapahtuvina toiminnoilla ei ole mitään merkitystä (ks. yllä oleva kohta 1.). Näin ollen kun kirjoitamme symbolista mikro-ohjelmakoodia, niin hyppykäskyssä emme kirjoita mitään kellon ajanjaksoina 1 ja 2, vaikka siihen tarkkaan ottaen pitäisi kirjoittaa ne toiminnot, jotka bitit c1...c8 saavat aikaan. Sen sijaan kirjoitamme kellon vaiheeseen kolme toiminnan, joka siirtää MPC:hen halutun osoitteen. Tarkastellaan esimerkkinä opintomonisteen s. 126 symbolisen mikro-ohjelman riviä 5. Tässä kirjoitetaan ; ;1010->MPC, joka tarkoittaa sitä, että c1...c8=00001010 eli kellon vaiheessa 1 siirretään väylälle 1 luku 1 (koska c5=1) ja se muunnetaan vastaluvukseen –1 (koska c7=1). Koska väylälle 2 ei siirretä mitään, siellä on nolla, jolloin väylälle 3 tulee –1, mutta sitä ei siirretä mihinkään rekisteriin, koska kellon vaiheen 2 mikään bitti ei ole päällä. Merkintä ; ;1010->MPC on siis hiukan harhaanjohtava, koska sen mukaan kellon

vaiheissa 1 ja 2 ei tapahdu mitään, joka muuttaisi tietokoneemme tilaa. *Näin ollen olisikin parempi kirjoittaa vain 1010->MPC*, joka ei ota kantaa siihen, mitä kellon vaiheissa 1, 2 ja 3 tapahtuu.

s. 137 loppu - s. 138. Huomaa, että mikro-ohjelman lause 4 sisältää kaksi toimintoa: A:n vähennyksen yhdellä ja hypyn lauseeseen 1. Tällainen temppuilu on tässä turhaa, joten selvempää olisi kirjoittaa kaksi mikrokäskyä. Huomaa, että ohjelmaa kirjoitettaessa tulee käyttää sellaisia käskyjä, jotka voidaan koneellamme toteuttaa.

Viikon tärkeimmät asiat

Mikro-ohjelmoitava koneen osat ja koneen ohjelmointi. Tentissä kysytään vain hyvin yksinkertaisia ohjelmanpätkiä eikä esim. opintomonisteen s. 136-137 pitkähköä ohjelmaa, joka on kuitenkin hyvin opettavainen ja kannattaa lukea huolella.

Harjoitustehtävät

1. Esitä seuraavat symboliset mikrokäskyt binäärisinä (22 bitin yhdelminä):

   MDR+C → MAR; (MAR) → MDR; 1 + MPC → MPC
   (1+A)×2 → MDR; MDR → (MAR); 1 + MPC → MPC
   MDR+B → A; ; (A<0) + MPC → MPC
   −1+D→ A; ; (A=0) + MPC → MPC
   10101010_2 → MPC

   Ohje. Käytä hyväksesi lisäsivua, jossa selitetään eo. symbolisten merkintöjen merkitys. Vaikka käskyt eivät saa peräkkäin suoritettuina mitään järkevää aikaan, niin mieti silti kunkin käskyn toiminta. Kirjoita osoitteet käyttäen 10-järjestelmän lukuja alkaen vaikkapa 100:sta (jolloin MPC:n arvo on aluksi siis 100).

2. -3. Mitä seuraava mikro-ohjelma tekee?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
100	1													1	1		1					1
101						1			1		1						1					1
102		1												1	1		1					1
103	1					1	1		1											1		1
104		1	1		1		1											1				
105			1										1				1					1

   Ohje: Kirjoita mikro-ohjelma ensin symbolisessa muodossa käyttäen samoja (lisäsivun mukaisia) merkintöjä kuin edellisessä tehtävässä. Mieti sen jälkeen ohjelman toimintaa esimerkillä, jossa ohjelman käsittelemissä rekistereissä on jotkut arvot.

4. Asenna koneeseesi kurssisivuilla oleva ohjelma Simo, joka simuloi koneemme toimintaa. Simuloi tehtävän 1 ensimmäistä käskyriviä ja totea että koneemme toimii. Aseta ensin MDR=2, C=3 ja aseta päämuistin muistipaikkaan 5 arvo 6.
   Ohje: Katso kurssisivuilta Simon käytön ohjeet. Ota muutama kuvaruutukaappaus, jolla voi todentaa tehtävän suoritetuksi.

5. -6. Konstruoi seuraavat mikro-ohjelmoitavassa tietokoneessa tarvittavat piirit:

   a) Piiri, joka tuottaa väylään 1 arvon 1, jos rekisterin A sisältö on 0, ja arvon 2, jos rekisterin A sisältö on nollasta poikkeava.

   b) Piiri, joka tuottaa väylään 1 arvon 1, jos rekisterin A sisältö on negatiivinen, ja muulloin arvon 2.

   Tällaista piiriä tarvitsemme, jotta voimme toteuttaa tietokoneellamme konekielemme (tarkastellaan myöhemmin) komennot JUMPNEG ja JUMPZERO, jossa suoritetaan ehdollinen hyppy sen mukaan onko akun (rekisteri A) sisältö negatiivinen tai nolla.
   Ohje: Kummallekin piirille on syötteinä rekisteristä A lähtevät johdot (joista otetaan käyttöön vain tarvittavat; esim. b-kohdassa vain A:n ylin bitti) ja ohjausbitti c19 (a-kohta) tai c20 (b-kohta) ja tuloksena on kaksi johtoa, jotka ovat kiinni DC1:n kahdessa alimpia bittejä vastaavissa johdoissa d_0 ja d_1 (tuloksena on luku 1 tai 2, jotka voidaan esittää kahdella bitillä). Rekisterissä A oleva luku on negatiivinen, jos sen ylin bitti on 1 (kahden komplementtiesitys!). Rekisterissä A oleva luku on nolla, jos luvun kaikki bitit ovat nollia ja vain tällöin myös niiden OR on nolla.

VIIKKO 38

Sisältö: Konekieli. Konekieltä tulkitseva mikro-ohjelma: mikrotulkki.

Oppimateriaali

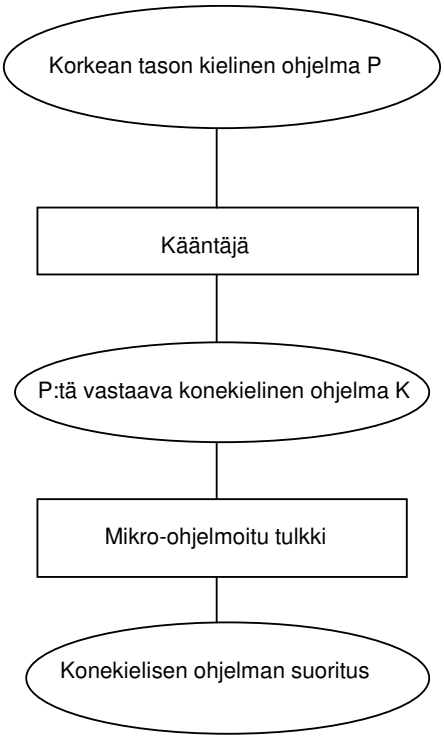
Opintomoniste s. 141 (luku 4.5) - 149 (lukuun 4.5.2 saakka)

Itsenäisen työskentelyn avuksi

Alustus:

Tähän mennessä olemme harjoitelleet tekemään mikro-ohjelmia. Kuitenkaan tarkoituksena ei ole suorittaa mikro-ohjelmointia kuin kerran: silloin kun tehdään mikro-ohjelmoitu tulkki, joka poltetaan MPM:ään. Tämän jälkeen voimme unohtaa koko mikro-ohjelmoinnin ja siirtyä korkeammalle tasolle.

Ennen viikon ohjelman tarkastelua puhumme siitä, mihin seuraavien viikkojen aikana pyritään. Tätä kuvaa lyhyesti alla oleva kuva sitä seuraava tarkentava teksti (kohdat 1. ja 2.). Nämä asiat ovat **tärkeitä**.



Tarkastellaan sitä miten saamme tietokoneemme suorittamaan päämuistissa[6] (eli keskusmuistissa) olevaa kirjoittamaamme korkean tason kielellä kirjoitettua ohjelmaa (vaikkapa opintomonisteen kielellä kirjoitettuja moduuleja). Yllä oleva kuva kertoo tässä tarvittavat vaiheet:

1. Ensin kirjoitetaan kääntäjäohjelma, joka lukee päämuistissa tai oheismuistissa olevaa korkeantasonkielistä ohjelmaa P ja tuottaa sitä vastaavan konekielisen ohjelman K päämuistiin. Tällainen kääntäjäohjelma tulee kirjoittaa vain kerran, jonka jälkeen sille voidaan antaa syötteenä mikä tahansa korkean tason kielinen ohjelma ja tuloksena saadaan sitä vastaava konekielinen ohjelma. Tällainen kääntäjäohjelma tehdään luvussa 5.3. Konekielinen ohjelma K on huomattavasti korkeamman tasoista kuin sitä vastaava mikro-ohjelma, mutta taas selvästi matalamman tasoinen kuin alkuperäinen korkean tason ohjelma P. Konekieli esitellään tämän viikon yhteydessä ja samalla harjoitellaan tekemään konekielisiä ohjelmia, jotta osaisimme kirjoittaa kääntäjän luvussa 5.3. Konekielinen ohjelma ei tietenkään vielä riitä, vaan meidän tulee toteuttaa konekielisen ohjelman K käskyt tietokoneessamme. Nimittäin tietokone saadaan toimimaan vain käyttäen mikrokäskyjä.
2. Sen vuoksi teemme mikro-ohjelman, joka lukee päämuistissa olevaa konekielistä ohjelmaa ja suorittaa konekielisen ohjelman vaatimat käskyt tietokoneellamme. Tätä ohjelmaa sanotaan mikro-ohjelmoiduksi tulkiksi konekielelle. Tämäkin ohjelma tehdään vain kerran ja se poltetaan MPM:ään. Kaikki tietokoneessa suoritettavat ohjelmat saavat aikaan tämän tulkkiohjelman käynnistyksen ja tämä mikro-ohjelma ohjaa tietokoneemme eri loogisten piirien toimintaa. Tämä tulkkiohjelma esitetään tällä viikolla.

Vaiheessa 1 tarvittava kääntäjäohjelma käsitellään kahden viikon päästä ja vaiheessa 2 tarvittava mikro-ohjelmoitu tulkki käsitellään tällä viikolla.

Päästäksemme konekielikoneeseen eli konekieliseen ohjelmointiin, nostamme koneen abstraktiotasoa. Konekielisen ohjelman säilytyspaikka on päämuisti (RAM), josta konekieliset käskyt tuodaan yksi kerrallaan tietokoneemme johonkin rekisteriin. Konekieli sisältää suppean komentojoukon (esim. LOAD, STORE, JUMP,...) ja kolme erikoisasemassa olevaa rekisteriä, joita kutsutaan nimillä PC (ohjelmalaskuri, joka kertoo seuraavaksi suoritettavan konekielisen käskyn osoitteen päämuistissa), IR (käskyrekisteri, johon haetaan päämuistista kukin suoritettava käsky vuorollaan) ja ACC (akku, joka on rekisteri, johon lähes kaikki konekieliset komennot kohdistuvat). Myöhemmin mikro-ohjelmoitua tulkkia kirjoittaessamme mietimme, miten nämä komennot voidaan toteuttaa koneellamme, ja samalla valitsemme jotkin koneemme todellisista rekistereistä vastaamaan abstrakteja rekistereitä PC, IR ja ACC.

Mikro-ohjelmoidun tulkin (yllä oleva vaihe 2) toiminta on yksinkertaistettuna seuraava. Oletetaan, että aluksi rekisterissä B on ensimmäisen konekielisen komennon osoite päämuistissa. Tulkkiohjelma toistaa seuraavia toimintoja:

[6] Päämuisti on sama kuin RAM ja MPM on sama kuin ROM.

1. Hae päämuistista muistipaikan, jonka osoite on B:ssä, sisältö rekisteriin MDR. MDR:ssä on tällöin jonkun konekielisen käskyn binäärikoodi.
2. Tutki MDR:n sisältö ja generoi ne mikrokäskyt, jotka vaaditaan ko. konekielisen komennon suorittamiseksi. Tällöin myös B:n sisältö muuttuu, jos MDR:ään ladataan hyppykäsky. Jos taas MDR:ssä ei ollut hyppykäskyä, niin B:n sisältöä kasvatetaan yhdellä.

Käyttämämme konekieli on melko alkeellinen, mutta käytännön konekielet tarjoavat useita muita ominaisuuksia, jotka helpottavat ohjelmointiprosessia. Näistä ominaisuuksista tarkastelemme erilaisia muistiosoitustapoja.

Help:

s. 142-143. Konekielinen ohjelma (ja myös ohjelmien käyttämä data) sijaitsee päämuistissa ja ohjelma koostuu 16-bittisistä käskyistä. Koska bittimuotoista konekielistä ohjelmaa on hankala lukea ja kirjoittaa, kirjoitamme konekielisen ohjelman symbolisessa muodossa. Esimerkiksi konekielen bittimuotoinen käsky 0001000000011111 kirjoitetaan symbolisessa muodossa LOAD 31. Käskyn toiminta on seuraava: lataa akkuun (=rekisteri A) päämuistin muistipaikan 31 sisältö. Muunnos bittimuotoisen ja symbolisen konekielen välillä on suoraviivainen: ohjelmoinnin kurssien perusteella voisimme kirjoittaa helposti tällaisen merkkijonoja käsittelevän metodin.

Lisäesimerkki. Tarkastellaan seuraavaa yksinkertaista asetuslausetta[7]: (13)←(13)+(10), jolloin tarkoituksena on siis lisätä muistipaikan 13 sisältöön muistipaikan 10 sisältö. Sitä vastaava konekielinen koodi on seuraava (käskyt ovat päämuistin muistipaikoissa 0,1,2):

```
0       LOAD    13
1       ADD     10
2       STORE   13
```

(*** lisäesimerkin loppu ***)

Tarkennus sivulle 143. Materiaaleissa ei käytetä nuolia ← ja → mielivaltaisesti, vaan käytöllä on tietty merkitys. Nimittäin käytettäessä nuolta ← toiminto rinnastetaan tavalliseen asetuslauseeseen, jossa lasketaan ensin nuolen oikealla puolella olevan lausekkeen arvo, joka tallennetaan nuolen vasemmalla puolella olevaan muistipaikkaan. Sen sijaan käsiteltäessä mikro-ohjelmointia nuolella → tarkoitetaan sitä, mitä tietoa siirretään väylille 1 ja 2 (esiintyvät nuolen vasemmalla puolella ja niiden välissä on +, koska väylän 1 ja 2 sisällöt viedään aina yhteenlaskulaitteen lävitse) ja mihin rekisteriin tulos (väylän 3 sisältö) viedään eli esim. MDR+0→A.

[7] muistipaikan sisältöön viitataan kirjoittamalla muistipaikan numero sulkeiden sisälle

Huomaa, että JUMPSUBia s. 144-146 ei vaadita mutta s. 145 ensimmäinen esimerkkiä vaaditaan.

Kun kirjoitat konekielistä ohjelmaa, voit sijoittaa sen vapaasti eli voit aloittaa käskyjen numeroinnin haluamastasi muistipaikasta. Joustava **käytäntö** (joka poikkeaa opintomonisteesta tässä kohtaa) on kuitenkin seuraava: aloita numerointi 0:sta tai ykkösestä, sijoita data alkuun ja kirjoita ohjelma sen jälkeen peräkkäisiin muistipaikkoihin (alkaen esim. muistipaikasta 100). Tämä, ns. suhteellinen, viittaustapa on käytännöllinen esimerkiksi käännettäessä korkean tason ohjelma konekielelle ja myös käyttöjärjestelmän kannalta, joka viime kädessä muuntaa meidän valitsemat ohjelmaosoitteet todellisiksi muistiosoitteiksi.

s. 145 konekielinen ohjelma, joka laskee $\mathbf{2^n}$:n. Jokaisen rivin vasemman puoleisin luku tarkoittaa käskyn päämuistin osoitetta, mutta se voidaan tulkita myös rivinumeroksi. Kuten huomaat käskyt ja data kirjoitetaan ohjelmaan samalla tavalla: esim. ohjelman rivi 381 (= päämuistin muistipaikan 381 sisältö) tarkoittaa sitä, että päämuistin muistipaikassa 381 on n:n arvo. Tällöin komento 'LOAD 381' tarkoittaa sitä, että ‘hae akkuun päämuistin muistipaikan 381 sisältö’. Koska konekielessämme ei ole sellaista komentoa, jolla voidaan ladata akkuun lukuvakio, tulee tarvittava lukuvakio sijoittaa johonkin muistipaikkaan mp, jolloin se saadaan akkuun komennolla 'LOAD mp'. Tällaista menettelyä käytetään esimerkissä lukuvakioiden 0 ja 1 osalta. Myöhemmin konekieltä laajennetaan komennolla 'LOADI k', joka tuo akkuun luvun k (siis k:n arvon eikä sen osoittaman muistipaikan sisältöä, kuten komento 'LOAD k').

Lisäesimerkki s. 148 loppuun. Tarkastellaan seuraavaksi lähemmin konekielen käskyn **LOAD** toteutusta koneellamme eli vastaavaa mikro-ohjelmoidun tulkin ohjelmakoodia. Olkoon MDR:n sisältö esim. **'LOAD 14'**. Käskyn LOAD koodi alkaa tulkkiohjelman riviltä 13. Ensiksi suoritetaan MDR+0→MAR, jolloin MDR:n 12 alinta bittiä siirtyy rekisteriin MAR, koska MDR on 16 bittinen, mutta MAR vain 12-bittinen (tätä merkitään opintomonisteessa $MDR_{11\text{-}0}$+0→MAR). Siis MAR=14, jonka jälkeen suoritetaan toiminta (MAR)→MDR, joka saa aikaan sen, että päämuistin muistipaikan 14 sisältö tuodaan MDR:ään. Vielä pitää siirtää MDR:n sisältö rekisteriin A, joka tehdään tulkin rivillä 14. Näin on suoritettu komento 'LOAD 14'. Opintomonisteen s. 149 on selvitetty lisäksi ADD-käskyn ja JUMPZERO-käskyn suoritus.

s. 150. Käskyn **LOADI** x (jossa x voidaan tulkita lukuvakioksi tai osoitteeksi riippuen tilanteesta) lisäys konekieleen on tärkeä, koska meillä ei ole tähän saakka ollut käytössä suoraa komentoa mielivaltaisen lukuvakion x lataamiseksi akkuun.

Viikon tärkeimmät asiat

Konekieli ja esimerkit konekielisistä ohjelmista, mikro-ohjelmoitu tulkki konekielelle.

Harjoitustehtävät

1. Tee mikro-ohjelma, joka muuttaa rekisterissä A olevan luvun itseisarvokseen. *Ohje*: Jos A on negatiivinen, se tulee muuntaa vastaluvukseen ja muuten ei tarvitse tehdä mitään. Vie A:n sisältö MDR:ään, nimittäin MDR:n sisältö voidaan viedä väylälle 1 ja väylän 1 sisältö voidaan muuntaa vastaluvukseen ennen sen viemistä yhteenlaskuyksikköön. Tästä tulee todellakin näin lyhyt.

2. Kirjoita konekielinen ohjelman pätkä, joka vaihtaa muistipaikkojen 1 ja 2 sisällöt keskenään (swap). Voit käyttää apuna muistipaikkaa 3 ja olkoon ensimmäisen käskyn osoite 100.

3. Kirjoita lausetta IF x>y THEN x:=x-y ELSE y:=y-x ENDIF vastaavat konekieliset komennot. Oletetaan, että x on muistipaikassa 1 ja y muistipaikassa 2. Olkoon ensimmäisen käskyn osoite 100.

4-5. Olkoon päämuistin 5 ensimmäisen muistipaikan sisältö kuten alla on esitetty. Tässä muistipaikat ja niiden sisältö on kuvattu käskymuodossa (suluissa esimerkkinä kolmannen rivin vastaavat osat): ensin on päämuistin osoite, joka on kirjoitettu 10-järjestelmän lukuna (3). Seuraavaksi on muistipaikan sisällön neljä ensimmäistä bittiä muodostava operaatiokoodi, joka on kirjoitettu aina käskynä (STORE), ja sitä seuraavat 12 bittiä on kirjoitettu 10-järjestelmän lukuna (4). Mitä tapahtuu, jos ohjelmalaskuri on aluksi 1 ja aloitetaan suorittamaan seuraavaa konekielistä ohjelmaa.

osoite		
1	LOAD	5
2	ADD	2
3	STORE	4
4	ADD	2
5	SUBTRACT	2

Ohje: Ohjelman jokainen rivi on vain bittijono ja sen merkitystä ei tiedetä ennen kuin suoritusvaiheessa. Erityisesti etukäteen ei siis nähdä tulkitaanko jokin bittijono dataksi vai käskyksi. Esim. 1. käskyssä ladataan akkuun muistipaikan 5 sisältö, jolloin muistipaikan 5 sisältö tulkitaan datana eikä komentona, vaikka se on komennon muotoon tässä kirjoitettu. Tällöin akkuun ladataan siis bittijono, joka vastaa komentoa 'SUBTRACT 2' eli 0100 000000000010_2. Sen jälkeen akkuun lisätään muistipaikan 2 sisältö eli ADD 2, jota vastaa bittijono … jne.

Huomaa siis, että koneen sisällä sekä toiminnot (käskyt) että data esitetään aina bittijonoina, mutta niiden tulkinta määräytyy vasta suoritusvaiheessa.

LUENTOPÄIVÄ YLIOPISTOLLA 24.9.2016

Luentopäivä pidetään klo 10:15 - n. 14:30. Luentopäivän tiedot varmistetaan **kurssisivuilla hyvissä ajoin ennen luentopäivää**. Luentopäivänä ei käsitellä mitään uusia asioita, vaan luentopäivä toimii yhteenvedon omaisena opetuksena. *Ota luentopäivälle mukaasi kurssin materiaalit: opintomoniste, opiskeluopas ja harjoitustehtävien malliratkaisut*.

Luentopäivän aluksi tarkastelemme luvun 3 tärkeimpiä asioita. Sen jälkeen käymme ensin lyhyesti lävitse joitakin tietokoneessa tarvittavia komponentteja (esim. kiikku) ja tarkastelemme mikro-ohjelmoitavan tiekoneemme rakennetta ja sitä, miten voimme ohjelmoida sitä. Sen jälkeen tarkastelemme konekieltä ja vertaamme konekielistä ohjelmointia mikro-ohjelmointiin ja ohjelmointiin korkean tason kielellä. Lopuksi tarkastelemme mikro-ohjelmoitua tulkkia konekielelle (ks. tämän oppaan s. 32 oleva kuva ja siihen liittyvä teksti) ja käymme läpi alla olevan kokoavan esimerkin. Luentopäivän aikana läpikäytävä materiaali on sekä sivumääräisesti että sisällöllisesti laaja, joten valmistaudu siihen huolella kertaamalla luentopäivällä tarkasteltavat asiat. Mieti kysymyksiä, joihin haluat vastauksen luentopäivänä. Tarkastelemme myös kahden vanhan tentin kysymyksiä, jotka tulevat myös kurssisivuille.

Alla on tiivistelmä ohjelmoinnin eri tasoista: korkean tason kielinen ohjelma, konekielinen ohjelma ja sitä vastaava mikro-ohjelma. Kyseisiä asioita on selitetty yksityiskohtaisemmin tässä opiskeluoppaassa ja opintomonisteessa. Näitä asioita havainnollistetaan tarkastelemalla asetuslausetta x := x + y.

Ohjelmoinnin eri tasot (tiivistelmä)

MIKRO-OHJELMOINTI

Mikro-ohjelma sijaitsee tietokoneemme muistissa MPM, jonka jokainen muistipaikka on 22-bittinen, ja käskyn bittejä merkitään: c1, c2, c3, … , c22. Näillä biteillä ohjataan koneemme piirien toimintaa. Kukin mikro-ohjelmakäsky tuodaan MPM:stä aina vuorollaan kellon vaiheessa 5 rekisteriin MIR, joka on kytketty ohjausväylään CC. Tämän käskyn bitit toimivat koneen komponenttien ohjausbitteinä.

Loppujen lopuksi kirjoitamme (joka poltetaan pysyvästi) MPM:ään sellaisen mikro-ohjelman, joka osaa suorittaa konekielisen ohjelman jokaisen käskyn. Tämä ohjelma on nimeltään mikro-ohjelmoitu tulkki konekielelle.

Tulkin toiminta on seuraava: hae toistuvasti päämuistista MM seuraava suoritettava konekielinen käsky ja suorita käskyn vaatimat toimenpiteet. Näin ollen mikro-ohjelmoidussa tulkissa (=MPM:ssä oleva mikro-ohjelma) tulee olla toimintavaihtoehto jokaista konekielistä käskyä varten. Tämä sama on tarkemmin opintomonisteen s. 136.

Toteutettaessa mikro-ohjelmoitua tulkkia tarvitsemme koneemme todelliset rekisterit, jossa on aina seuraavaksi suoritettavan päämuistissa olevan konekielisen käskyn osoite ja lisäksi tarvitsemme rekisterin, johon konekielinen käsky tuodaan. Sovimme että koneemme rekisteri B (konekielikoneen abstrakti rekisteri PC, ohjelmalaskuri) toimii aina seuraavan suoritettavan päämuistissa oleva konekielisen käskyn osoitteen säilytyspaikkana, ja

konekielinen käsky haetaan päämuistista rekisteriin MDR (konekielikoneen abstrakti rekisteri IR, käskyrekisteri).

Huomaa siis, että mikro-ohjelman yksi käsky on aivan eri asia kuin konekielinen käsky. Useimmiten yhtä konekielistä käskyä varten tarvitsemme useita mikro-ohjelmakäskyjä. Huomaa lisäksi, että konekielikone ja konekielinen ohjelmointi on abstraktia: rekisterit akku, PC ja IR ovat loogisia nimiä tietyn tyyppiselle/tiettyä tehtävää edustavalle rekisterille ja meidän tulee valita konkreettisesta koneestamme tietyt rekisterit, jotka edustavat näitä rekistereitä.

KONEKIELINEN-OHJELMOINTI

Nyt siirrymme paljon korkeammalle abstraktiotasolle. Ensinnäkin valitsemme tietyn käskyjoukon (LOAD, STORE, JUMP, JUMPNEG,…), jota konekielessämme saa käyttää. Tarkoituksena on se, että konekielinen ohjelma sijaitsee päämuistissa, joka on 16-bittinen. Näin ollen jokainen konekielinen käsky tulee esittää 16 bitillä. Konekielisessä käskyssä 4 ylintä bittiä kertoo käskyn toiminnan ja loput bitit kertovat sen päämuistissa olevan muistipaikan osoitteen, johon toiminta kohdistetaan (ks. s. 131).

Konekielisessä ohjelmoinnissa on keskeisessä asemassa tietty rekisteri (jota sanotaan akuksi), johon kaikki operaatiot kohdistetaan. Olkoon se nimeltään ACC. Yllä tuli jo kerrottua, mitkä koneemme todelliset rekisterit vastaavat abstrakteja rekistereitä PC (ohjelmalaskuri) ja IR (käskyrekisteri). Lisäksi tulee valita todellinen rekisteri akulle. Koska konekielisessä ohjelmoinnissa tulee verrata akun sisältöä nollaan ja tutkia onko sen sisältö negatiivinen, niin meidän täytyy valita akuksi koneemme rekisteri A.

KORKEAN TASON KIELEN OHJELMAN SUORITUS TIETOKONEELLAMME

Yhtä korkean tason kielistä käskyä vastaa usea konekielinen käsky. Kääntäjäohjelma lukee korkean tason kielistä ohjelmaa ja tuottaa siitä vastaavan konekielisen ohjelman päämuistiin. Mikro-ohjelmoitu tulkki taas lukee päämuistissa olevan konekielisen ohjelman ja saa aikaan sen vaatimat toimenpiteet tietokoneellamme. Tämä on tarkemmin tässä oppaassa s. 33-.

KOKOAVA ESIMERKKI

Tarkastellaan lauseen

x := x + y

suorituksessa tarvittavia vaiheita.

Ensin se käännetään konekielelle, jossa tarvitaan vaiheet: selaaminen, jäsentäminen ja koodin generointi (tee harjoituksena!). Opintomonisteen s. 156 moduuli tuottaa tästä seuraavat konekieliset komennot:

```
LOAD x
ADD y
STORE x
```

Kun tässä korvataan symboliset osoitteet (x, y ja z) todellisilla muistiosoitteilla, voisi koneemme päämuisti MM näyttää esimerkiksi seuraavalta:

päämuisti:

osoite	sisältö	
6	x:n arvo	
7	y:n arvo	
…		
90	LOAD 6	(= 0001 0…0110)
91	ADD 7	(= 0011 0…0111)
92	STORE 6	(= 0010 0…0110)

Miten tietokoneemme suorittaa nämä komennot? Mikro-ohjelmoidun tulkin toiminta on pääpiirteissään s. 136 alussa. Sen tarkka versio eli mikro-ohjelma, joka on poltettu tietokoneemme MPM:ään muistipaikkoihin 0-72, esitetään samalla sivulla käyttäen symbolisia mikro-ohjelmakäskyjä, jotka ovat todellisuudessa 22-bittisiä ohjausbittien jonoja.

Seuraavassa esitetään mikro-ohjelmoidun tulkin suorittamat 'komennot' lauseiden LOAD 6 ja ADD 7 suorittamiseksi. Alla käytetään 10-järjestelmän lukuja ja symbolisia mikro-ohjelmakäskyjä (bittijonojen sijasta) kuten opintomonisteessakin. Ensimmäisen mikrokäskyn 0 suorituksessa kirjoitamme toiminnan suluissa (eli …) myös binäärisenä.

Aluksi MPC=0 ja B=90.

0 0+B -> MAR eli 90 -> MAR (eli luvun 90 binäärisesitys -> MAR) ;
(MAR) -> MDR eli LOAD 6 -> MDR (eli bittijono 0001 0…0110 -> MDR);
1+MPC -> MPC eli 1+0 -> MPC (eli 0…01 -> MPC)

1 1+B -> B eli 1+90=91 -> B; ;
$MDR_{15\text{-}12}$+MPC -> MPC eli 1+1=2 -> MPC

2 13 -> MPC

13 $MDR_{11\text{-}0}$+0 -> MAR eli 6+0=6 -> MAR;
(MAR) -> MDR eli x -> MDR;
1+MPC -> MPC eli 1+13=14 -> MPC

14 MDR+0 -> A eli x+0=x -> A; ;
0+0=0 -> MPC

Tähän saakka koneemme on suorittanut lauseen LOAD 6 eli: lataa rekisteriin A muistipaikan 6 sisältö.

0 0+B -> MAR eli 0+91=91 -> MAR;
(MAR) -> MDR eli ADD 7 -> MDR;
1+MPC -> MPC eli 1+0=1 -> MPC

1 1+B -> B eli 1+91=92 -> B; ;
$MDR_{15\text{-}12}$+MPC -> MPC eli 3+1=4 -> MPC

4 17 -> MPC

17 $MDR_{11\text{-}0}$+0 -> MAR eli 7+0=7 -> MAR;
(MAR) -> MDR eli y -> MDR;

1+MPC -> MPC eli 1+17=18 -> MPC

18 MDR+A -> A eli y+A -> A eli y+x -> A, koska A:ssa oli ennen tätä x ; ;
0+0=0 -> MPC

Tähän saakka koneemme on suorittanut lauseet LOAD 6 ja ADD 7 eli: lataa ensin rekisteriin A muistipaikan 6 sisältö ja lisää sen jälkeen rekisterin A sisältöön muistipaikan 7 sisältö. Jatkuu samaan tapaan …

VIIKKO 39

Sisältö: Tietokoneverkot. Systeemiohjelmisto: ohjelmointikielen tulkitseminen ja kääntäminen. Kieliopin määrittely. Kääntäjän toiminta: selaus, jäsennyspuun muodostaminen, konekielisen koodin generointi.

Oppimateriaali

Opintomoniste s. 153-164 kappaleeseen ”Osittava jäsentäminen” saakka.

Itsenäisen työskentelyn avuksi

Alustus:

Luvussa 5 käsitellään eri systeemiohjelmistoja, joista tässä opintomonisteessa käsitellään tarkemmin kääntäjät ja tulkit sekä käyttöjärjestelmä ja sen tärkeimmät tehtävät: erityisesti resurssien jako sekä keskus- ja levymuistin hallinta. Tällä viikolla käsitellään systeemiohjelmistoja yleisellä tasolla ja keskittyen lähinnä ohjelman tulkitsemiseen kääntämiseen konekielelle.

Jokaisella korkean tason kielellä on oma kielioppinsa, joka määrittelee hyvin tarkalla tasolla millaiset merkkijonot ovat oikein kirjoitettuja ohjelmia. Ohjelmointikielen jäsentäjä toimii kieliopin sääntöjen mukaisesti yrittäen päätellä onko ohjelma johdettavissa lähtösymbolista ja myönteisessä tapauksessa antaa tuloksena jäsennyspuun, jonka perusteella vastaava konekielinen ohjelma voidaan muodostaa automaattisesti. Konekielisen koodin generoi kääntäjäohjelma, jonka syötteenä on ohjelman jäsennyspuu. Koodin generointiin kuuluu muistin varaaminen ohjelman muuttujille (ns. symbolitaulun perusteella) ja konekielisten käskyjen muodostaminen. Kääntäjäohjelma sisältää jokaista syntaksiluokkaa (esim. <toistolause>) varten moduulin, joka tuottaa kyseistä syntaksiluokkaa vastaavat konekieliset komennot.

Ohjelman käännösprosessi sisältää siis ne vaiheet, joiden avulla korkean tason kielisestä ohjelmasta saadaan konekielinen ohjelma. Nämä vaiheet ovat: 1) ohjelman selaaminen ja 2) jäsennyspuun muodostaminen, jonka perusteella voidaan 3) generoida (tulostaa) ohjelmaa vastaava konekielinen koodi.

Huom. Viime viikolla käsiteltiin mikro-ohjelmoitu tulkki (ROM:ssa oleva mikro-ohjelma), joka lukee päämuistissa olevaa konekielistä ohjelmaa ja suorittaa sitä tietokoneellamme (eli saa aikaan vastaavat mikrokäskyt). Tällä viikolla käsittelemme puuttuvan vaiheen: miten korkean tason kielisestä ohjelmasta saadaan konekielinen ohjelma. Näin ollen olemme käyneet läpi kaikki ne vaiheet, joita tarvitaan, jotta korkean tason kielinen ohjelma voidaan suorittaa tietokoneellamme (ks. myös tämän oppaan s. 32 kuva ja siihen liittyvä teksti).

Help:

s. 161. Huomaa, että **produktiosäännön** vasemmalla puolella on aina nonterminaali ja oikealla puolella voi olla nonterminaaleja ja/tai terminaaleja. Produktiosääntöjen yksinkertaistamiseksi oikealla puolella käytetään merkkipareja **{ }** ja **[ ]** sekä merkkiä **|** ilmaisemaan seuraavaa:

- X→Y | U | V tarkoittaa, että X voidaan korvata joko Y:llä tai U:lla tai V:llä. Tämä on siis vain lyhennysmerkintä kolmelle produktiolle: X→Y, X→U ja X→V.
- Merkintä {Y} tarkoittaa, että kyseiseen kohtaan Y voidaan kirjoittaa 0, 1, 2, 3, ... kertaa. Huomaa erityisesti, että Y voi olla myös kokonaan pois. Esimerkiksi produktio X → {Y} voidaan kirjoittaa myös muodossa X → ε | Y X.
- Merkintä [Y] tarkoittaa, että kyseiseen kohtaan Y voidaan kirjoittaa 0 tai 1 kertaa. Merkinnällä ilmaistaan siis valinnaisuus: kyseinen osa on mukana tai se voi olla pois. Sen vuoksi merkintää käytetäänkin produktiosääntöjen oikealla puolella ilmaisemaan oikean puolen valinnaisia osia (esim. ELSE- ja OTHERS-haara valintalauseissa).

Huomaa, että nämä ovat siis oikeastaan lyhennysmerkintöjä pitemmille produktioille ja eivät ole siis välttämättömiä, mutta helpottaa produktioiden kirjoittamista.

Tarkastellaan sivun 161 jälkimmäistä esimerkkiä, jossa esitetään eräs **kielioppi opintomonisteen ohjelmointikielelle**. Ensimmäinen produktio sanoo, että ohjelma koostuu yhdestä tai useammasta peräkkäisestä moduulista (ensin tulee <moduuli>, jota seuraa 0 tai 1 tai 2 ... <moduuli>:ia). Toinen produktio määrää moduulin yleisen rakenteen määrittelemättä vielä minkälainen on moduulin runko (<lauselista>). Moduulin otsikko alkaa sanalla MODULE (joka on siis terminaali), jota seuraa moduulin nimi <tunnus>, jonka jälkeen tulee aina olla parametrilista, joka tulee ympäröidä kaarisulkeilla. Siis vaikka moduulilla ei olisikaan parametreja, tulee moduulin nimen perään kirjoittaa[8] ( ). Otsikon lopussa voi olla RETURNS-osa tai sitten ei. Jos RETURNS-osaa ei ole, niin moduuli on toiminnallisesti proseduuri ja muulloin se on funktio, jolloin tuloksen tyyppi tulee ilmoittaa RETURNS-sanan perässä. Mahdollisia tyyppejä tässä ovat INTEGER, REAL, BOOLEAN, CHAR ja LIST.

Parametrilista määritellään **rekursiivisesti**, joka on hyvin tavallista kielioppien yhteydessä. <parametrilista> on muotoa <tyyppi> <tunnus> <parametrilista> tai se on tyhjä. Vaihtoehto tyhjä vaaditaan siksi, että muutenhan kyseinen rekursiivinen määrittely ei päättyisi laisinkaan sovellettaessa sitä toistuvasti ja lisäksi parametrittomiakin funktioita kirjoitetaan. Parametrilista koostuu siis peräkkäisistä <tyyppi> <tunnus> pareista tai se on tyhjä (parametriton moduuli). Näin ollen esim. INTEGER x INTEGER y on tämän kieliopin mukainen parametrilista, kun taas esim. INTEGER x y ei ole. Myöskään pilkkuja ei saa siis käyttää erottamaan parametreja toisistaan. Parametrilista voidaan määritellä myös käyttäen {}-notaatiota:

<parametrilista> → {<tyyppi> <tunnus>}

[8] kuten Javassa

jolloin tämä olisi voitu sijoittaa suoraan <moduuli>:n produktiosääntöön (2. produktio). Edellä esitetty rekursiivinen määritelmä kuvastaa kuitenkin paremmin kääntäjäohjelman toimintaa.

Lauselista koostuu peräkkäisistä lauseista tai on tyhjä (tyhjän lauseen käsite on hyödyllinen: vrt. Java ja puolipisteiden käyttö). Ehtolause ja valintalause määritellään yleisesti: ELSE- tai OTHERS-haara on valinnainen, joka ilmaistaan [ ]-merkinnällä. Ehto määritellään tässä normaalia rajoittuneempana: se on muotoa <alkio> <relaatio> <alkio>, missä <alkio> voi olla vain tunnus tai luku. Esimerkiksi ehto a<b on kieliopin mukainen, kun taas ehto a+1<b ei ole. Lausekkeen käsite on myös rajoittunut, koska lausekkeessa ei voi olla osallisena kuin kaksi operandia; esim. b+c on kieliopin mukainen, mutta b+c+d ei ole tämän kieliopin mukainen lauseke. Näin ollen myöskään asetuslause a:=b+c+d ei ole tämän kieliopin mukainen.

Opintomonisteessa jäsennyspuut kirjoitetaan myös käyttäen **sulkumerkkinotaatiota** (*ei vaadita*): esimerkiksi merkintä p(p1,p2,p3) tarkoittaa puuta, jonka juuri on p ja poikapuut vasemmalta lukien ovat p1, p2 ja p3. Poikapuut esitetään vastaavasti käyttäen sulkumerkkinotaatiota. Yllä oleva puu voidaan kehittää seuraavasti (kirjoitetaan selvyyden vuoksi terminaalit hipsukoiden sisään):

- <toistolause>('WHILE',<ehto>(...),'DO',<lauselista>(...),'ENDWHILE')
- <toistolause>('WHILE',<ehto>(<alkio>(...),<relaatio>(...),<alkio>(...)),'DO', <lauselista>(<lause>(...), ε),'ENDWHILE')
- <toistolause>('WHILE',<ehto>(<alkio>(<tunnus>('x')),<relaatio>('>'), <alkio>(<luku>('1')),'DO', <lauselista>(<lause>(...), ε),'ENDWHILE')

jne.

Kuten huomaat sulkumerkkiesitys on epäselvä monien sulkeidensa takia, jos kaikki alipuut esitetään täydellisinä. Näin ollen kannattaakin piirtää aina jäsennyspuu.

Opintomonisteen sivuilla 166 (Koodin generointi) -171 käsitellään ohjelman jäsentämistä ja konekielisen koodin tuottamista hyvin tarkalla tasolla. Vaikka tätä osaa *ei vaadita*, niin tämä osuus kannattaa silmäillä läpi. Siellä esitetään tarkat algoritmit, joilla korkean tason kielisestä ohjelmasta voidaan tuottaa **automaattisesti** konekielinen ohjelma. Tässä määritellään kullekin lausetyypille oma moduulinsa, joka tulostaa vastaavat konekieliset komennot. Algoritmien toiminta vastaa samaa ajattelumallia kuin millä me kirjoitamme konekielisiä ohjelmia (esim. viime viikon tehtävä 2) ja tämä malli on suhteellisen helppo automatisoida eli kirjoittaa sitä varten algoritmi.

Viikon tärkeimmät asiat

Ohjelmointikielen tulkitseminen ja kääntäminen, kielioppi, kääntäjän toiminta.

Harjoitustehtävät

Tehtävissä 2-4 tarkastellaan s. 161-162 kielioppia.

1. Laadi konekielinen ohjelma, joka laskee kertoman N! = 1·2·3· ... ·(N−1)·N. Luku N on muistipaikassa 1, ja tulos viedään muistipaikkaan 2. Ohjelma ei saa muuttaa muistipaikan 1 sisältöä.
   Ohje: Kirjoita ohjelma alkaen muistipaikasta 100. Voit olettaa, että muistipaikassa 3 on luku 1 ja muistipaikkaa 4 voit käyttää apumuuttujana (kertoja).

2. Määrittele kielioppiin <tunnus>. Sovitaan, että tunnus alkaa aina kirjaimella ja sitä voi seurata kirjaimia ja/tai numeroita.

3. Piirrä lauseen IF x>y THEN x:=x-y ELSE y:=y-x ENDIF jäsennyspuu.

4. a) Miten alla olevan moduulin syt otsikkoriviä tulee muuttaa, jotta se olisi kieliopin mukainen.
   b) Moduulissa ei voi olla moduulin kutsua (miksi?). Laajenna kielioppia niin, että esimerkiksi opintomonisteen rekursiivinen moduuli syt-algoritmi (alla) olisi kieliopin mukainen.

```
MODULE syt(x, y) RETURNS x:n ja y:n suurin yhteinen tekijä
        IF x = y THEN RETURN x ENDIF
        IF x > y THEN RETURN syt(x-y, y) ENDIF
        IF x < y THEN RETURN syt(x, y-x) ENDIF
ENDMODULE
```

VIIKKO 40

Sisältö: Epädeterministisyys, hakuprobleemat: syvyys- ja leveyshaku, pelien pelaaminen, rinnakkaisalgoritmit.

Oppimateriaali

Opintomoniste s. 185-194 (sivulta 194 vain sen ensimmäinen esimerkki).

Itsenäisen työskentelyn avuksi

Alustus:

Tällä viikolla käsittelemme lukua 6, joka on eräänlainen 'mitä muita tärkeitä asioita tietojenkäsittelyssä tarvitaan ja tarkastellaan'. Asiat eivät kuitenkaan muodosta 'sekametelisoppaa', vaan liittyvät rinnakkaisuutta lukuun ottamatta ns. **tekoälyn** (artificial intelligence) käsitteeseen. Tutustumme lyhyesti epädeterministisyyteen, yleisiin hakuongelmiin, ongelman tilaesitykseen, pelien pelaamiseen. Tässä ei kuitenkaan puhuta tekoälystä, vaan käsitellään näitä periaatteita ja asioita erikseen lähinnä muutaman esimerkin avulla. Tällä viikolla käsitellään tehtävän tilaesitys ja sen myötä esitellään tärkeät hakumenetelmät: syvyys- ja leveyshaku. Lopuksi käsitellään hyvin lyhyesti rinnakkaisalgoritmeja, jolloin meillä on käytössä useita prosessoreita.

Pelien pelaaminen yhdistetään hyvin vahvasti tekoälyyn, jossa usein nostetaan aivan tarpeettomasti esille vastakkainasettelu: ihminen vastaan kone. Viime vuosina on kiinnitetty suurta huomiota esimerkiksi siihen, kun Kasparov (shakin ykkösnimi) ja IBM:n kehittämä - ja jatkuvan kehittämisen kohteena oleva - ohjelma ottelivat. Pelit ovat olleet hyvin tasaväkisiä, mutta viime aikoina kone on useimmiten päihittänyt ihmisen, joskin hyvin epätavanomainen strategia saattaa aiheuttaa koneelle hankaluuksia. On kuitenkin hyvä huomata, että shakkiin kehitetty ohjelma ei osaa tehdä mitään muuta, joten ei ole pelottavaa tai outoa, jos tällainen ohjelma voittaa ihmisen shakinpeluussa. On oikeastaan aika ihmeellistä, että tällaisen shakinpeluuohjelman kehittämiseen on kulunut näinkin kauan.

Tällä viikolla tutustutaan pelien pelaamiseen hyvin lyhyesti tarkastellen kahden pelaajan pelejä; esimerkkinä jätkänshakki (risti-nolla). Kuitenkin jo tämä esimerkki tuo esille sen, että pelien pelaaminen on mitä vaikein tehtävä. Pelien pelaaminen perustuu usein ns. *arviofunktioon*, joka liittää jokaiseen pelitilanteeseen (ja näin myös jokaiseen mahdolliseen siirtoon) numeerisen hyvyysarvon. Sen vuoksi tällaista funktiota kutsutaan myös hyvyysfunktioksi. Siirto suoritetaan tietenkin siihen tilaan, jonka hyvyysarvo on paras. Näin ollen pelien pelaaminenkin voidaan ajatella hakuongelmana. Pelien pelaamisessa on tyypillistä, että siirron hyvyyttä ei mitata yhden siirron päähän vaan usean siirron päähän. Tässä on tietenkin ongelmana se, että tällöin meidän täytyy myös olettaa jotain vastustajan pelaamisesta ja usein lähdetään tietenkin siitä, että myös vastustaja pelaa hyvin (käyttää esim. samaa strategiaa).

Rinnakkaisalgoritmeissa meillä on käytössä useita prosessoreita, jolloin alkuperäinen tehtävä jaetaan näiden prosessorien suoritettavaksi samanaikaisesti. On tietenkin selvää, että ei-laskettavat tehtävät eivät muutu laskettaviksi rinnakkaisuuden myötä, mutta joissakin tapauksissa rinnakkaisuuden avulla saadaan algoritmin vaatima suoritusaika putoamaan kohtuulliseksi.

Help:

s. 185. **Tilaesityksen** käyttö on hyvin yleistä tietojenkäsittelyssä. Esimerkiksi ohjelman suoritusta voidaan kuvata täsmällisesti tilaesityksen avulla. Tällöin tiloja vastaa muistin eri sisällöt eli ohjelmassa käytettyjen muistipaikkojen sisällöt (muuttujien arvot) ohjelman suorituksen aikana. Tällaista tilaesitystä käsiteltiin jo ohjelmoinnin jaksoilla. Jokainen ohjelman lause saattaa muuttaa muistin tilaa. Sivulla 187-188 käsitellään tien löytämistä ulos labyrintista. Tässä alkutila on labyrintin ruutu 4 ja lopputila ruutu 3. Muut tilat ovat kaikki mahdolliset sijainnit labyrintissa eli ruudut 1,2,5,6,7,8 ja 9. Mahdolliset siirrot tarkoittavat taas mahdollisia siirtymisiä labyrintissa: esim. 4→5 on yksi tämän ongelman siirto, mutta esim. 4→2 ei ole. Seuraavaksi esitämme kaksi menetelmää, joiden avulla voidaan löytää tie läpi labyrintin.

s. 186-187 **leveys**- ja **syvyyshaku**. Tässä yhteydessä tarvitaan opintomonisteen alussa (jakson TTP I-opintomoniste) käsiteltyjä abstrakteja tietotyyppejä (ADT) pino ja jono. Nyt ei ole oleellista se, miten nämä rakenteet on tallennettu tietokoneen muistiin (siis mikä on niiden talletusrakenne), vaan se miten ne käyttäytyvät alkion lisäyksen ja poiston suhteen eli mikä on näiden abstraktien tietotyyppien **abstrakti malli**. Tätähän käsiteltiin jo opintojen alussa jaksolla TTP I.

- Jonolle on luonteenomaista se, että jonosta poistetaan aina jonossa kauimmin ollut alkio.
- Pinosta taas poistetaan vähiten aikaa pinossa ollut alkio (siis viimeksi lisätty)

Tällainen käytös saadaan aikaan siten, että jonossa lisäys ja poisto tehdään aina listan **eri** päihin (lisäys listan loppuun ja poisto listan alusta), kun taas pinossa lisäys ja poisto tehdään aina listan **samaan** päähän.

Tämä tulee huomioida, kun tarkastellaan leveyshakualgoritmin komentoja: ‘Ota jonosta tila s’ ja ‘Lisää tilat s_1, ..., s_k jonoon’ ja vastaavia syvyyshaun komentoja. Algoritmien komennot ‘Ota jonosta tila s’ määräävät haun suunnan eli voidaan ajatella, että algoritmin tässä kohtaa suoritetaan aina toiminto: mene tilaan s. Esim. labyrinttiongelmassa tämä vastaisi tilannetta: siirry ruutuun s. On erityisen tärkeätä huomata, että **syvyys- ja leveyshakualgoritmien ainoa ero on käytetty listarakenne (pino/jono)**. Algoritmeissa esiintyvä komento ‘Aseta osoittimet tiloista s_1, ..., s_k tilaan s’ tehdään vain siksi, että saadaan piirrettyä hakupuu ja oikea ratkaisupolku, jos niin halutaan. Huomaa kuitenkin, että puussa nuolet (algoritmin asettamat osoittimet) kulkevat alhaalta ylös. Jos syvyys- ja leveyshakualgoritmit tuntuvat vaikeasti ymmärrettäviltä, niin ne varmaankin selviävät, kun suoritamme s. 187 labyrinttitehtävän algoritmien mukaisesti.

Huom. Kummassakin algoritmissa tulee huomata, se että jos toistorakenteessa on tilanne, jossa ei löydetä tutkimattomia tiloja, niin lauseet Aseta … ja Lisää …eivät saa aikaan mitään eli mennään testaamaan WHILE-lauseen ehtoa.

s. 187-188 esimerkki **labyrintissa** kulkemisesta. Tarkastellaan **leveyshakua** annetulla heuristiikalla p, i, e, l. Heuristiikka määrää ilmansuuntien kokeilujärjestyksen jokaisessa tilassa (siis aina yritetään edetä ensin pohjoiseen, seuraavaksi itään jne.) Jos annettu ilmansuunta johtaa umpikujaan, peräännytään sellaiseen tilaan, jossa on vielä ilmansuuntia, joita ei ole kokeiltu. Esitetään jonon alkiot peräkkäin pilkuilla erotettuina siten, että vasemmanpuoleisin alkio on jonon ensimmäinen alkio. Tällöin lisäys jonoon tehdään lisäämällä alkiot listan perään ja poisto ottamalla alkiot pois listan alusta. Ensin jonoon laitetaan alkutila 4 (merkitään: jono=4), jonka jälkeen alkaa WHILE-silmukka. Seuraavassa on silmukan runko-osan suorituksen aikaansaamat toimenpiteet (eli alla selitetään s. 188 alussa olevaa jonon kehitystä):

1. Otetaan jonosta pois tila s=4 (jonka jälkeen se on tyhjä) ja katsotaan, mihin siitä voi edetä. Mahdolliset vaihtoehdot ovat 1,5,7 ja juuri tässä (paremmuus)järjestyksessä annettua heuristiikkaa käyttäen. Nämä alkiot lisätään jonoon eli jono=1,5,7. Sen jälkeen asetetaan osoittimet: 1→4, 5→4, 7→4, jonka jälkeen aloitetaan silmukan uusi kierros, koska jono ei ole tyhjä ja ratkaisua (=3) ei ole vielä löydetty.
2. Otetaan jonosta tila s=1, jolloin jono=5,7. Nyt ykkösestä ei päästä mihinkään tutkimattomaan tilaan, joten suoritetaan silmukan runko uudelleen.
3. Otetaan jonosta tila s=5, jolloin jono=7. Ruudusta 5 päästään ruutuun 2, joka lisätään jonon perään eli jono=7,2. Asetetaan osoitin 2→5.
4. Otetaan jonosta tila s=7, jolloin jono=2. Ruudusta 7 päästään ruutuun 8, joka lisätään jonon perään eli jono=2,8. Asetetaan osoitin 8→7.
5. ...

s. 188. **Lisäesimerkki: syvyyshaku labyrintissa**. Seuraavassa on pinon kehitys syvyyshakua käytettäessä heuristiikalla p, i, e, l opintomonisteen s. 187 labyrintissa. Pinossa alkion lisäys ja poisto tehdään aina listan samaan päähän. Kirjoitetaan pinon alkiot vaakatasossa niin, että vasemmanpuoleisin alkio on pinon huippu, jolloin lisäys ja poisto tapahtuvat kumpikin listan vasempaan päähän.

4
1,5,7 (katso alla oleva Huom1.)
5,7
2,7 (huom. lisäys ja poisto pinon päälle eli listan alkuun)
7
8
9
6
3

Näin ollen kulkujärjestys on seuraava: 4,1,5,2,7,8,9,6,3, joka nähdään taas ensimmäisistä alkioista. Syvyyshakua käytettäessä ruutuja kokeillaan eri järjestyksessä kuin leveyshaussa, ja näin ollen hakupuukin piirretään eri järjestyksessä. Molemmissa tapauksissa tutkitaan kuitenkin labyrintin kaikki ruudut. Tämä johtuu

siitä, että annettu heuristiikka ei ole paras mahdollinen. Opintomonisteen s. 188 esitetään parempi heuristiikka: e, i, l, p, jolla löydetään lyhin reitti.

Huom1. Pinon käsittelyssä on huomattava myös se, että kun alkiot laitetaan pinoon yksitellen (esim. jos tämä ohjelmoitaisiin 'oikeasti'), tulee ne laittaa pinoon **huonommuusjärjestyksessä**, koska tällöin paras jää päällimmäiseksi. Näin ollen vaiheessa 2 laitetaan tilat pinoon järjestyksessä 7,5,1, jolloin pinon pohjalla on 7 ja päällä 1.

Huom2. Kun syvyyshaussa ei löydetä uusia tutkimattomia tiloja, palataan edelliseen tilaan. Tämä tarkoittaa sitä että peräännytään edelliseen tilaan. Näin ollen syvyyshaulle voidaan kirjoittaa helposti labyrintissa kulkemista animoiva ohjelma käyttäen jaksolla JIT käsiteltyä kilpikonnagrafiikkaa, jollainen on toteutettu esim. ohjelmointikielessä Logo.

s. 190-191. **Jätkänshakki.** Tarkastellaan alkutilan jälkeistä tilaa s=5 eli tilannetta jossa ruudukossa on vai yksi symboli: X keskellä ruudukkoa. Silloin tämän tilan hyvyys arvioidaan arvolla $f_X(5)$, joka määräytyy seuraavasti. Jos X täyttää ruudukon omalla merkillään X, on ruudukko täynnä X:iä. Tällöin X:n kolmen suorien lukumäärä on 8. Sen sijaan tässä tilassa O:lla on vain 4 mahdollista kolmen suoraa (reunat).

O	O	O
O	X	O
O	O	O

Siis $f_X(5)=8-4$. Vastaavasti lasketaan kaikki muut arvot $f_X(i)$, missä i=1,2,3,4,6,7,8,9. Kaikkia arvoja ei tietenkään tarvitse laskea; esimerkiksi tilat s=1,3,7 ja 9 ovat symmetrian nojalla yhtä hyviä. Samoin tilat s=2,4,6,8. Näissä tiloissa f_X:n arvo on pienempi kuin tilassa 5, joten X:n paras siirto tyhjälle ruudulle on ruudukon keskusta (tähänhän mekin aina panemme aluksi, jos saamme aloittaa! Valinta perustuu intuitioon: siinä on eniten mahdollisuuksia). Seuraavaksi on O:n siirtovuoro. Tässä oletamme, että myös O pelaa samanlaista strategiaa käyttäen, eli

$f_O(s)$ = O:n kolmen suorien lukumäärä, kun tilassa s tyhjät ruudut täytetään O:lla − X:n kolmen suorien lukumäärä, kun tilassa s tyhjät ruudut täytetään X:llä.

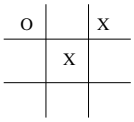
Symmetrian nojalla $f_O(s)=-f_X(s)$, joten yhtä hyvin O:n siirto voidaan arvioida funktion f_X avulla, mutta tällöin tulee se minimoida valittaessa O:n kannalta paras siirto.

Tarkastellaan tilannetta s = "X on keskellä ja O vasemmassa ylänurkassa" eli

O		
	X	

Kun tässä tilassa ruudukko täytetään X:llä, on X:llä 5 suoraa. Kun tässä tilassa ruudukko täytetään O:llä, on O:lla 4 suoraa eli $f_X(1)=5-4=1$, jne.

s. 192. Huomaa, että s. 191 algoritmissa tarkastellaan tilannetta, jossa X on tekemässä päätöstä kun taas s. 192 tarkastelemme O:n päätöstä. Sivun 192 esimerkissä f tarkoittaa f_X:ää eli O käyttää X:n arviofunktiota. Selvitetään esimerkin laskuja, koska useimmiten opiskelijat ovat 'oikaisseet laskuissa'. Laskut ovat ihmiselle aika monimutkaisia, mutta koneelle helppoja, kunhan algoritmi on ohjelmoitu oikein! Tarkastellaan tilaa



ja on O:n vuoro. Tällöin s_{24} tarkoittaa tilaa kun yllä olevassa ruudukossa O sijoittaa ruutuun 2 ja X ruutuun 4, jolloin ruudukossa on 5 merkintää. Kun tässä tilassa X täyttää ruudukon X:llä, tulee 4 suoraa. Kun tässä samassa tilassa s_{24} O täyttää ruudukon O:lla, tulee 1 suora. Näin ollen $f_X(s_{24})=4-1=3$. O käyttää siis X:n arviofunktiota f_X ja näin voidaan tehdä, koska $f_O(s)=-f_X(s)$. Sen vuoksi O:n tulee määrätä algoritmissa maksimien minimi:

> Laske kullekin s_i:lle luku $g(s_i) = \max \{f_X(s_{ij}) \mid s_i \rightarrow s_{ij}$ on vastustajan mahdollinen siirto$\}$
> Valitse siirto siihen tilaan s_i, jolle $g(s_i)$ on pienin.

Jos haluamme käyttää algoritmia O:n oman arviofunktion kanssa (joka on luonnollisempaa koska onhan O:n vuoro!), niin silloin laskemme arvon $f_X(s_{24})$ sijasta arvon $f_O(s_{24})=-3$. Tällöin algoritmissa lasketaan minimien maksimi:

> Laske kullekin s_i:lle luku $g(s_i) = \min \{f_O(s_{ij}) \mid s_i \rightarrow s_{ij}$ on vastustajan mahdollinen siirto$\}$
> Valitse siirto siihen tilaan s_i, jolle $g(s_i)$ on suurin.

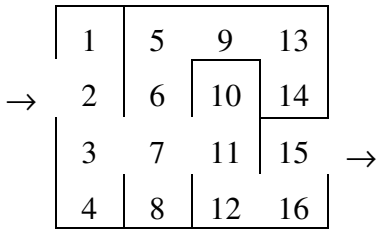
s. 193. **Minmax**-tekniikka. Tämäkään metodi ei välttämättä huomaa s. 193 ylälaidassa olevaa tavallaan triviaalia tilannetta, jossa on vain yksi järkevä siirto X:n kannalta. Mutta tämä on helppo korjata, nimittäin tutkitaan joka tilassa myös se, että onko kyseinen tila vastustajan voitto. Myönteisessä tapauksessa liitetään tällaiseen tilaan sellainen hyvyysarvo, joka ei voi tulla valituksi (tässä miinus ääretön $-\infty$ eli jonkin 'äärettömän pienen' luvun). Vastaavasti tilaan, jossa on oma voitto, tulee liittää mahdollisimman hyvä arvo ($+\infty$)

Viikon tärkeimmät asiat

Syvyys- ja leveyshaku ja niiden soveltaminen (esim. labyrintissa). Pelien pelaamisen periaatteet ja siinä käytettävä arviofunktio.

Harjoitustehtävät

Ratkaise seuraava labyrinttiongelma

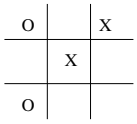


Tehtävä 1. leveyshaulla

Tehtävä 2. syvyyshaulla

kun heuristiikkana on p, i, e, l. Tässä pitää kirjoittaa myös tarkasteltavien tietorakenteiden sisältö eikä antaa vain hakupuita. Miten siirtojen kokeilujärjestyksen määräävä heuristiikka vaikuttaa lopputulokseen? Mikä tässä olisi parempi heuristiikka (ja miksi)?

Tarkastellaan lopuksi jätkänshakin pelaamista minmax-tekniikalla. Esitä miten X siirtäisi tilanteessa



Tehtävä 3-4. käyttäen arviofunktiota f_X,

*Tehtävä 5**. käyttäen parannettua arviofunktiota f^*_X.

*Tehtävä 6**. Tehtävät 3-5 kun oletetaan että onkin O:n siirtovuoro.

Arviofunktiot on määritelty opintomonisteessa.
Ohje: Katso yllä oleva huomautus koskien s. 192 esimerkkiä.