Esipuhe

Tämä opintomoniste on tarkoitettu käytettäväksi opintojaksoilla, jotka käsittelevät algoritmien ja yleensä tietojenkäsittelytieteen perusteita, joskin jotkin käsiteltävät asiat menevät jo aika syvälle. Materiaali on laaja, joten opettaja voi keskittyä tärkeimpinä pitämiinsä asioihin.

Algoritmien esityksessä käytetään pseudokieltä eikä todellista ohjelmointikieltä. Pedagogisena perusteluna on se, että tarkoitus on opettaa yleisesti, millaisia rakenteita ja käsitteitä ohjelmoinnissa tarvitaan ja käytetään, eikä tarkoituksena ole oppia yhtä tiettyä todellista ohjelmointikieltä. Todellista kieltä käytettäessä syntaksi saattaa "varastaa shown" ja lisäksi todellisia ohjelmia yritetään saada toimimaan kokeilemalla, ja siten ei aina ymmärretä algoritmin toimintaa kunnolla. Tietysti ohjelmien suorittaminen oikeassa ympäristössä on myös opettavaista.

Opintomoniste perustuu osittain alla mainittuun melko vanhaan mutta silti vielä nykyäänkin erinomaiseen Goldschlagerin ja Listerin kirjaan sekä moniin aiempiin monisteisiin, joita ovat vuosien varrella olleet kirjoittelemassa ainakin Jorma Boberg, Martti Penttonen, Tapio Salakoski ja Jukka Teuhola. Tämän monisteen tueksi ovat vanhat kurssisivut: http://staff.cs.utu.fi/kurssit/JAJO/ ja http://staff.cs.utu.fi/kurssit/TP/, jotka sisältävät lisäesimerkkejä ja -materiaalia.

Uusin versio tästä monisteesta on saatavilla sivuilta: http://staff.cs.utu.fi/staff/jorma.boberg/Mat. Sivuilta löytyvät myös korjaukset monisteeseen sekä aiemmat versiot. Sivuilla on opintojen tueksi ohjelmisto Ville, joka havainnollistaa algoritmien toimintaa muistipaikkatasolla ja ohjelmisto, joka simuloi tämän materiaalin mikro-ohjelmoitavan tietokoneen ohjelmointia ja toimintaa. Lisäksi em. sivulla on lyhyt tiivistelmä imperatiivisen ohjelmoinnin peruskäsitteistä sekä tässä monisteessa käytetyn pseudokielen syntaksi.

Monisteesta löytyviä virheitä ja yleisiäkin kommentteja voi lähettää alla olevaan sähköpostiosoitteeseen. Kiitän Lasse Bergrothia viimeisistä kommenteista.

Turussa 25.6.2012

Jorma Boberg e-mail: boberg@utu.fi

Kirjallisuutta

Brookshear: *Computer Science: An Overview*. Addison-Wesley, 2011 tai vanhempi. Se sisältää hyvin samankaltaista asiaa kuin tämä moniste ja on **hyvä hankinta**! Kirjan vanhempi painos on myös suomennettu: Tietotekniikka, IT Press, Edita 2003 ja sitä löytynee myös kirjastoista.

Goldschlager, Lister: *Computer Science – A Modern Introduction*. Prentice-Hall, 1988.

Paananen: *Tietotekniikan peruskirja*, 6. laitos, Docendo Finland Oy, 2005. Hyvä ja laaja käytännön asioita ja terminologiaa sisältävä käsikirja, hyvin erilainen kuin tämä moniste.

3 Algoritmiteoriaa

Edellisessä luvussa tarkasteltiin algoritmien muodostamista ja niiden esittämistä. Tässä luvussa tarkastellaan algoritmien yleisiä teoreettisia ominaisuuksia, joiden perusteella tehtävien vaikeutta ja niihin laadittujen algoritmien hyvyyttä voidaan arvioida. Nämä ominaisuudet toimivat myös kriteereinä, joiden perusteella yhtäältä tehtäviä ja toisaalta samaan tehtävään laadittuja algoritmeja voidaan verrata keskenään. Käsiteltävät ominaisuudet ovat laskettavuus, kompleksisuus ja oikeellisuus.

3.1 Tehtävän algoritminen ratkeavuus

3.1.1 Laskettavuus

Edellisessä luvussa esiteltiin monia tehtäviä ja niiden ratkaisemiseksi laadittuja algoritmeja eli mekaanisia ohjeita. Esitetyt tehtävät selvästikin voidaan ratkaista tietokoneella ja ne ovat siis algoritmisesti ratkeavia. Tällaisia tehtäviä sanotaan ***laskettavissa***[4] (computable) oleviksi. Mutta onko olemassa tehtäviä, joita tietokone ei pysty suorittamaan, ts. tehtäviä, joiden ratkaisemiseksi ei ole algoritmia? Kyllä, on olemassa paljon tehtäviä, joiden suorittamiseksi ei ole vielä kyetty tekemään algoritmia. Mutta sen lisäksi on olemassa tehtäviä, joiden algoritminen ratkaiseminen on mahdotonta eli on osoitettu, että tehtävän ratkaisemiseksi ei kyetä ikinä kirjoittamaan algoritmia.

Laskettavissa olevien tehtävien määrä on äärettömän pieni algoritmisesti ratkeamattomien tehtävien määrään verrattuna. Useimpia tehtäviä tietokone ei siis osaa suorittaa! Onneksi suuri osa laskettavissa olevista tehtävistä on juuri sellaisia mielenkiintoisia tehtäviä, joita varten algoritmi halutaankin löytää. Toisaalta voidaan tietysti väittää, että tällaiset mielenkiintoiset ongelmat ovat mielenkiintoisia juuri siksi, että ne ovat algoritmisesti ratkaistavissa.

Idea algoritmista, ohjeista jonkin tehtävän suorittamiseksi, on tuhansia vuosia vanha. Hyvin pitkään uskottiin, että ei ole olemassa ongelmia, joille ei voida löytää algoritmista ratkaisua eli tehtävän ratkaisevaa algoritmia. Toisin sanoen uskottiin, että jos jollekin tehtävälle ei ratkaisua tunneta, se johtuu siitä, ettei sitä ole vielä keksitty. Ja jos jollekin tehtäville ei yrityksistä huolimatta algoritmia vielä osattu kirjoittaa, sen ajateltiin johtuvan vain siitä, että ongelmaa ei vielä ymmärretty riittävän hyvin. Ongelman algoritmista ratkeavuutta tai ratkeamattomuutta ei siis osattu pitää ongelman ominaisuutena. Mutta 1930-luvulla kyettiin osoittamaan, että on olemassa ongelmia, joille ei ole olemassa algoritmista ratkaisua. Saksalainen matemaatikko David Hilbert (1862-1943) esitti seuraavan ongelman ('Entscheidungsproblem'): voidaanko mielivaltaisesta kokonaislukuja koskevasta väitteestä algoritmisesti selvittää, pitääkö se paikkansa vai ei? Tšekkoslovakialaissyntyinen matemaatikko Kurt Gödel (s. 1906) julkaisi vuonna 1931 kuuluisan epätäydellisyyslauseensa ('Incompleteness Theorem'), jossa osoitettiin, että

[4] käsitteet 'algoritmisesti ratkeava' ja 'laskettavissa oleva' ovat samat, mutta tässä käytetään termiä laskettavuus, koska sitä käytetään yleisesti kirjallisuudessa.

Hilbertin ongelma ei ole algoritmisesti ratkaistavissa. Tulos oli melkoinen sensaatio, ja aiheutti suurta hämmennystä laajoissa tiedemiespiireissä (mikä sinänsä on mielenkiintoista, koska päinvastainen tulos – kaikki ongelmat ovat algoritmisesti ratkeavia – olisi itse asiassa tehnyt matemaatikot tarpeettomiksi!). Sittemmin Hilbertin ja Gödelin lisäksi mm. matemaatikot Alonso Church, Stephen Kleene, Emil Post ja Alan Turing esittivät muita algoritmisesti ratkeamattomia ongelmia. On erityisen huomion arvoista, että nämä kaikkia tietokoneita – niin nykyisiä kuin tuleviakin – koskevat tulokset todistettiin jo 1930-luvulla, vuosikausia ennen kuin ensimmäistäkään tietokonetta oli rakennettu![5]

3.1.2 Church-Turingin teesit

Ennen kuin jokin tehtävä voidaan osoittaa algoritmisesti ratkeamattomaksi, on tietysti tarkkaan määriteltävä, mikä algoritmi on. Se ei kuitenkaan ole yksinkertainen tehtävä. Algoritmin käsitteelle on esitetty mm. seuraavia määrittelyjä:

- rekursiivisten funktioiden teorian mukaiset säännöt uusien matemaattisten funktioiden muodostamiseksi olemassa olevista funktioista (Kleene 1935)
- erään symbolien manipulointiin tarkoitetun formalismin, ns. λ-kalkyylin mukainen määriteltävyys (Church 1936)
- eräälle hypoteettiselle koneelle, ns. Turingin koneelle annettava käskyjoukko (Turing 1937)
- ns. Markovin algoritmit (Markov 1951)

Esitetyt algoritmien määritelmät on osoitettu ekvivalenteiksi: jos ongelma voidaan ratkaista yhdellä tavalla määritellyllä algoritmilla, se voidaan ratkaista jokaisella muulla tavalla määritellyllä algoritmilla. Tähän liittyvät Churchin ja Turingin esittämät teesit:

1. Kaikki tunnetut, järkevät algoritmin määritelmät ovat keskenään ekvivalentteja.
2. Mikä tahansa järkevä algoritmin määritelmä ikinä keksitäänkään, se tulee olemaan ekvivalentti tunnettujen määritelmien kanssa.

Teeseistä ensimmäinen on osoitettu oikeaksi. Jälkimmäistä ei tietenkään voikaan todistaa oikeaksi, koska se koskee vielä tuntemattomia algoritmin määritelmiä. Teesit on kuitenkin yleisesti hyväksytty, ts. niitä pidetään tosina, vaikka niitä ei olekaan osoitettu oikeiksi. Teesien idea on siinä, että algoritmin käsite on intuitiivisesti selkeä ja hyvin määritelty, mutta sen luonteva formalisointi on vaikeaa. Niinpä me voimmekin lisätä algoritmin määritelmien joukkoon vielä yhden luonnehdinnan:

- Algoritmeja ovat tällä kurssilla algoritmien esitykseen käytetyn pseudokielen avulla kirjoitettavissa olevat toimenpidesarjat[6].

Tällä kurssilla käytetty pseudokieli voidaan todistaa tarkalleen yhtä vahvaksi kuin muutkin algoritmin määritelmät. Algoritmi voidaan siis määritellä miten hyvänsä, kunhan määritelmän mukaiset algoritmit voidaan suorittaa automaattisesti tietokoneella. Entä pitääkö tämä kone sitten määritellä, jotta algoritmin määritelmän järkevyydestä voitaisiin olla varmoja? Ei tarvitse, sillä voidaan osoittaa, että jokainen algoritmi, joka

[5] Tietokoneiden historian lasketaan yleensä alkaneen vasta toisen maailmansodan loppuvaiheissa.

[6] kunhan sovitaan täsmällisesti sallitut toimenpiteet

voidaan suorittaa jossakin tietokoneessa, voidaan suorittaa missä tahansa toisessa tietokoneessa. Tietokoneet voivat myös simuloida toisiaan, ts. jokaiselle tietokoneelle A on mahdollista kirjoittaa ohjelma, jonka avulla A pystyy suorittamaan minkä tahansa toisen tietokoneen B ohjelmia. Tällaista ohjelmaa nimitetään tulkiksi, simulaattoriksi tai universaaliohjelmaksi.

Tulkin olemassaolosta seuraa, että jokaista ohjelmointikieltä varten on tulkki ja että tulkin avulla voidaan suorittaa kaikki ne tehtävät, jotka millä tahansa toisella kielellä kirjoitettuja ohjelmia suorittava tietokone voi suorittaa. Tätä sanotaan algoritmien universaalisuudeksi. Mikä tahansa tietokone on ekvivalentti kaikkien toisten tietokoneiden kanssa: ne kaikki voivat suorittaa tarkalleen samat tehtävät. Käytännössä tietokoneiden välillä on toki suuriakin eroja: nopeus, muistikapasiteetti, ohjelmien saatavuus, ohjelmoinnin helppous jne.

3.1.3 Pysähtymisongelma ja muita algoritmisesti ratkeamattomia ongelmia

Algoritmin pysähtyminen annetuilla syötteillä ei ole suinkaan aina selvää, esimerkiksi silloin tällöin ohjelmat jäävät ikuiseen silmukkaan. Olisi erittäin hyödyllistä tietää etukäteen kunkin ohjelman kohdalla, pysähtyykö se annetuilla syötteillä vai ei.

Pysähtymisongelma: Olkoon annettu mielivaltainen ohjelma P ja sen mielivaltaiset syöttötiedot D. Pysähtyykö ohjelma P syötteellä D (siis P(D)) vai ei.

Osoitetaan seuraavaksi, että pysähtymisongelma ei ole laskettavissa. Todistus suoritetaan ns. vastaoletustekniikalla: oletetaan, että pysähtymisongelman ratkaiseva algoritmi on olemassa ja johdetaan sen avulla ristiriitainen (absurdi) tilanne, joka ei voi missään olosuhteissa olla voimassa. Tästä voidaan päätellä, että vastaoletus ei voi olla voimassa.

Oletetaan siis, että pysähtymisongelma olisi algoritmisesti ratkeava, eli olisi olemassa algoritmi, joka kertoisi pysähtyykö P syötteellä D vai eikö. Kutsutaan tätä algoritmia pysähtymistestiksi. Moduulilla on kaksi syötettä, ohjelma P ja sille syöttötiedot D:

```
MODULE pysähtymistesti(P, D) RETURNS kyllä / ei
(*  moduuli palauttaa arvon kyllä, jos P(D) pysähtyy ja arvon ei, jos P(D) ei pysähdy *)
ENDMODULE
```

Huomaa, että moduuli 'pysähtymistesti' itse aina pysähtyy ja palauttaa tuloksen. Koska pysähtymistesti testaa ohjelman P pysähtymistä kaikilla syötteillä D, sitä voidaan käyttää konstruoitaessa rajoittuneempi algoritmi, joka testaa P:n pysähtymistä, kun sille annetaan syötteenä P:n ohjelmakoodi (so. tekstitiedosto). Tässä ei ole mitään kummallista, onhan olemassa lukuisia ohjelmia, jotka saavat syötteenä ohjelmia, kuten ohjelmointikielten kääntäjät, editorit jne. Annetaan tälle uudelle algoritmille nimeksi pysähtyyitsellään:

```
MODULE pysähtyyitsellään(P) RETURNS kyllä / ei
        RETURN pysähtymistesti(P, P)
ENDMODULE
```

Konstruoidaan näiden moduulien avulla vielä uusi moduuli funny, joka on haettu ristiriitainen moduuli.

```
MODULE funny(P) RETURNS kyllä tai ei pysähdy
        IF pysähtyyitsellään(P) =  kyllä THEN
                REPEAT UNTIL 1=0    (*  ikuinen silmukka  *)
        ELSE RETURN kyllä
        ENDIF
ENDMODULE
```

Mutta miksi moduulin funny kaltaista moduulia ei voi olla olemassa? Tarkastellaan kutsua funny(funny). Moduulin funny määritelmän nojalla kutsun funny(funny) suoritus ei pysähdy, jos pysähtyyitsellään(funny)=kyllä. Mutta tämähän tarkoittaa sitä, että moduuli funny pysähtyy, kun sille annetaan syötteenä moduuli itse, ts. funny(funny) pysähtyy. Tällainen tilanne on mahdoton, joten asettamamme vastaoletuksen tulee olla väärä, eli moduulia pysähtymistesti ei voi olla olemassa.

On tietenkin olemassa ohjelmia, joiden kohdalla pysähtyminen on helppo todeta, mutta näin ei ole kaikkien ohjelmien kohdalla. Monen ohjelman pysähtymisen selvittämiseksi tarvitaan hyvin paljon luovuutta. Tästä on alla esimerkkinä Fermat-moduulin pysähtymisen tarkastelu. Pelkästään mekaaniseen päättelyyn (algoritmiin) perustuvaa yleistä ratkaisua pysähtymisongelmaan ei siis ole olemassa. Käytännön ratkaisu pysähtymisongelmaan on se, että ohjelmat, jotka eivät kohtuullisessa ajassa pysähdy, pysäytetään.

Esimerkki. Fermat'n suuri lause (last theorem). Ranskalainen matemaatikko Pierre de Fermat (1601 – 1665) esitti muistikirjansa reunassa väitteen, että ei ole olemassa sellaisia positiivisia kokonaislukuja a, b, c ja n, että $a^n + b^n = c^n$, kun n>2. Lisäksi hän kirjoitti "keksineensä väitteelle oivallisen todistuksenkin, joka ei kuitenkaan aivan mahtunut muistikirjan marginaaliin". Valitettavasti Fermat kuoli ennen kuin hän ehti julkaista teoreemaansa todistuksineen, joskin nykyään ei uskota sen todistuksen oikeellisuuteen. Fermat'n kuoleman jälkeen matemaatikot kaikkialla maailmassa ovat yrittäneet todistaa Fermat'n väitettä todeksi tai epätodeksi. Oliko Fermat oikeassa vai ei? Asian selvittämiseksi voidaan kirjoittaa algoritmi: käydään läpi kaikki kyseeseen tulevat lukunelikot, ja kokeillaan, toteutuuko yhtälö jonkin nelikon (a, b, c, n) kohdalla. On vain huolehdittava siitä, että kaikki nelikot todella tulevat joskus kokeiltua. Deterministisen algoritmin laatimiseksi lukunelikot (a,b,c,n) tulee järjestää jonoksi siten, että jokainen kombinaatio esiintyy jonossa ennemmin tai myöhemmin. Lukunelikot voi järjestää esimerkiksi kasvavan summan mukaan: ensin kaikki nelikot, joiden summa on kuusi, sitten kaikki nelikot, joiden summa on seitsemän, sitten kahdeksan jne. Siis (1,1,1,3), (1,1,1,4), (1,1,2,3), (1,2,1,3), (2,1,1,3), (1,1,1,5), (1,1,2,4), (1,1,3,3),...

```
MODULE Fermat(tarkasteltavien lukunelikoiden jono)
        REPEAT
                Ota jonosta seuraava nelikko (a, b, c, n)
        UNTIL a^n + b^n = c^n
        Tulosta luvut a, b, c ja n
ENDMODULE
```

Pysähtyykö tämä algoritmi? Ilmeisesti kokeellisesti emme kuitenkaan saa koskaan tietää sitä, sillä vaikka algoritmi olisi pyörinyt kuinka kauan, se saattaa silti pysähtyä juuri seuraavalla hetkellä. Mutta jos algoritmi pysähtyy, huomaamme Fermat'n olleen väärässä. Andrew Wiles todisti Fermat'n otaksuman todeksi vasta vuonna 1994 käyttäen erittäin järeätä korkeampaa matematiikkaa. Todistus oli yli sata sivua pitkä ja sen ovat tarkistaneet jo useat matemaatikot. Näin ollen edellä oleva algoritmi ei pysähdy, vaikka tietenkin algoritmin mukainen tietokoneohjelma pysähtyy ylivuotoon, kun tarkasteltavan lausekkeen arvo on liian suuri.

Vaikka edellä sanottiin, että monet mielenkiintoiset tehtävät ovat laskettavissa, myös monet tärkeät ongelmat on osoitettu algoritmisesti ratkeamattomiksi. Seuraavassa esitetään kolme ongelmaa, jotka eivät ole laskettavissa. Todistukset perustuvat siihen, että pysähtymisongelma ei ole laskettavissa. Yleinen todistusstrategia on seuraava: tehdään vastaoletus, että tarkasteltava ongelma olisikin laskettavissa eli on olemassa tehtävän ratkaiseva algoritmi. Sitten konstruoidaan tämän algoritmin avulla uusi algoritmi, joka ratkaisee pysähtymisongelman (tai jonkun muun ongelman, joka ei ole laskettavissa).

Totaalisuusongelma: Jos ohjelma P pysähtyy kaikilla syöttötiedoilla D, ohjelmaa P sanotaan totaaliseksi. Totaalisuusongelmassa kysytään, onko ohjelma P totaalinen.

Tämä ei ole sama ongelma kuin pysähtymisongelma, koska pysähtymisongelmassa kysytään pysähtyykö P annetulla syötteellä eikä kaikilla mahdollisilla syötteillä.

Ekvivalenssiongelma: Olkoon annettu kaksi ohjelmaa P1 ja P2. Ekvivalenssiongelman ratkaisu on ohjelma, joka ilmoittaa, ovatko P1 ja P2 ekvivalentit, ts. antavatko ne samoilla syöttötiedoilla aina saman tuloksen.

Ricen teoreema: Olkoon T tehtävä ja A algoritmi. Ei ole olemassa algoritmia, joka kertoisi, tekeekö algoritmi A tehtävän T.

3.1.4 Yhteenveto laskettavuudesta

- Jos on olemassa algoritmi, joka ratkaisee annetun tehtävän kaikilla mahdollisilla syötteillä, sanotaan, että tehtävä on laskettavissa.
- Jos sanotaan, että tehtävä ei ole laskettavissa, niin silloin pystytään todistamaan, että ei ole olemassa algoritmia, joka ratkaisee tehtävän kaikilla syötteillä. Sen vuoksi tällöin sanotaan myös, että tehtävä on ***algoritmisesti ratkeamaton***.
- Lisäksi tietenkin on tehtäviä, joiden laskettavuudesta ei tiedetä; eli ei ole vielä kyetty kirjoittamaan tehtävän ratkaisevaa algoritmia, mutta toisaalta ei ole kyetty todistamaankaan, että tällaista algoritmia ei ole olemassa.

3.2 Kompleksisuus

Parhaimmallakaan tietokoneella ei voida ratkaista kaikkia ongelmia. Ainoastaan häviävän pieni osa kaikista tehtävistä on laskettavissa. Laskettavissa olevien algoritmien osalta on tarpeellista tietää etukäteen, minkä verran ne kuluttavat suorituksensa aikana tietokoneen resursseja. Kompleksisuusteoria on tietojenkäsittelytieteen haara, joka hakee vastauksia kysymyksiin algoritmien suorittamiseen tarvittavien resurssien määrästä. Algoritmin kompleksisuudella mitataan siis sen tehokkuutta. Kompleksisuuden avulla voidaan vertailla eri algoritmien tehokkuutta samaan tehtävään. Huomaa erityisesti, että kompleksisuudella ei tarkoiteta algoritmin monimutkaisuutta ja ymmärrettävyyttä (toisin kuin arkikielessä).

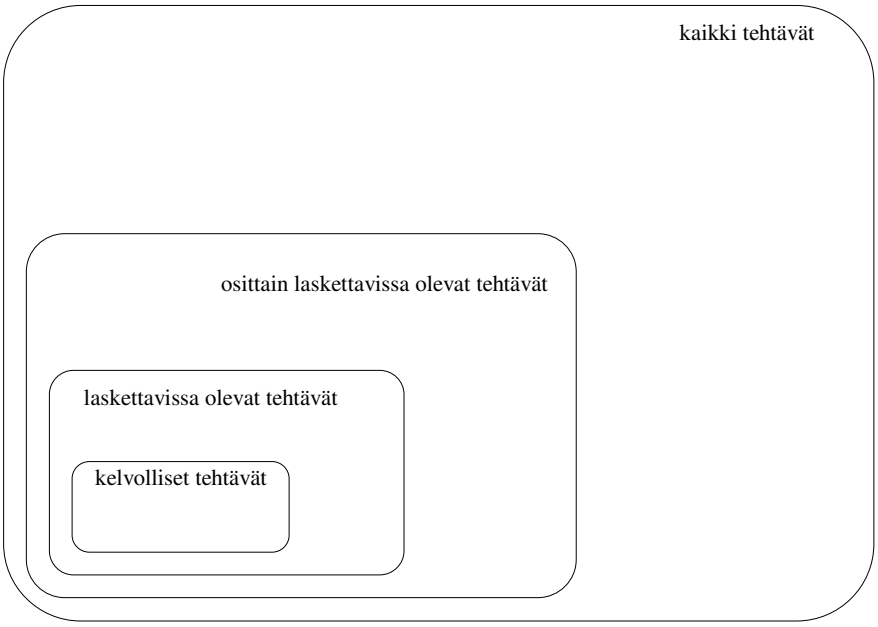
Esimerkki. Suurimman yhteisen tekijän määräämiseen esitettiin iteratiivinen algoritmi kappaleessa 2.6 ja rekursiivinen kappaleessa 2.7. Esitystapaeroista huolimatta moduulit määräävät suurimman yhteisen tekijän laskennallisesti tarkalleen samalla tavalla – vähennyslaskun avulla – ja ovat näin muodoin myös yhtä tehokkaita. Esimerkiksi lasku syt(75,6) etenee seuraavasti: syt(75,6) = syt(69,6) = syt(63,6) = syt(57,6) = ... = syt(9,6) = syt(3,6) = syt(3,3) = 3. Mutta syt voidaan määrätä myös tehokkaammin käyttäen jakojäännös- eli modulolaskua. Nimittäin jos luvuista toinen on paljon toista suurempi, on turhaa vähentää pienempää lukua suuremmasta monta kertaa peräkkäin – on tehokkaampaa vähentää pienempi luku suuremmasta heti niin monta kertaa kuin mahdollista. Mutta tämähän tarkoittaa juuri lukujen jakojäännöksen määräämistä. Tämän, ns. Euklideen algoritmin idea on siinä, että syt(x,y) pysyy muuttumattomana, kun x korvataan toistuvasti y:llä ja y puolestaan jakolaskun x/y jakojäännöksellä, jota merkitään: x mod y. Kun y tulee nollaksi, palautetaan x. Ohjelma perustuu siis seuraavaan rekursioyhtälöön:

$$\text{syt}(x,y) = \begin{cases} \text{syt}(y, x \bmod y), & \text{jos} \quad y > 0, \quad x > 0 \\ x, & \text{jos } y = 0, \quad x > 0. \end{cases}$$

```
(* Alkuehto: x ja y ovat positiivisia kokonaislukuja *)
MODULE syt(x,y) RETURNS x:n ja y:n suurimman yhteisen tekijän
        IF y=0 THEN
                RETURN x
        ELSE
                RETURN syt(y, x mod y)
        ENDIF
ENDMODULE
```

Nyt lasku syt(75,6) etenee seuraavasti: syt(75,6) = syt(6,3) = syt(3,0) = 3. Huomaa, että rekursiivisissa kutsuissa ensimmäinen argumentti (uusi x) on aina suurempi kuin toinen argumentti (uusi y). Jos aluksi x<y, niin luvut vain vaihtavat paikkaa ensimmäisessä rekursiivisessa kutsussa. Toimenpiteitä tarvitaan vain murto-osa aikaisempaan verrattuna. Täten moduuli on aiempaa selvästi tehokkaampi ratkaisu, jos mod voidaan laskea tehokkasti (kuten yleensä on laita).

Paitsi saman tehtävän eri ratkaisuja, kompleksisuuden avulla voidaan vertailla myös eri tehtäviä. Laskettavistakin tehtävistä valitettavasti vain hyvin pieni osa on resurssitarpeensa vuoksi käytännössä suoritettavissa. Vain ne algoritmit, joiden resurssitarve on kohtuullinen, ovat käyttökelpoisia. Tehtäviä, joiden ratkaisemiseksi on löydettävissä käyttökelpoinen algoritmi, sanotaan ***kelvollisiksi*** (feasible).



Tärkeimmät tietokoneresurssit ovat ***aika***, ***muisti (tila)*** ja ***laitteisto.*** Aika tarkoittaa algoritmin suoritukseen kuluvaa aikaa, muisti algoritmin tarvitsemaa tallennustilan määrää ja laitteisto tarvittavaa fyysistä laitteistoa algoritmin suorittamiseksi (esimerkiksi rinnakkaislaskennassa tarvittavien prosessorien määrää – rinnakkaisalgoritmeihin palataan myöhemmin).

Algoritmit käsittelevät syötettä, jonka koko (määrä) vaikuttaa algoritmin resurssitarpeisiin. Tehtävän ***koolla*** tarkoitetaan tarkasteltavaa ongelmaa kuvaavan tietorakenteen kokoa – luvun suuruutta, listan pituutta, puun solmujen määrää jne. Pienet ongelman tapaukset ratkeavat yleensä helposti, ja yleensä tehtävän vaikeus kasvaa syötteen koon kasvaessa. Usein ongelman tietyt tapaukset ovat vaikeampia kuin jotkin toiset, samankokoiset. Niinpä onkin tarkoituksenmukaista määritellä:

Määritelmä. ***Algoritmin kompleksisuudella*** tarkoitetaan algoritmin suoritukseen vaadittavien resurssien määrän riippuvuutta tehtävän koosta huonoimmassa tapauksessa. ***Tehtävän kompleksisuudella*** tarkoitetaan kompleksisuuden suhteen parhaan tehtävän ratkaisevan algoritmin kompleksisuutta.

Yleensä on pyrittävä valitsemaan se algoritmi, joka kuluttaa vähiten resursseja. Algoritmin valinnassa on kuitenkin otettava huomioon se, että vähennettäessä algoritmin jonkin resurssin (esimerkiksi ajan) tarvetta saattaa jonkin toisen resurssin (esimerkiksi muistitilan) tarve kasvaa. Se, minkä resurssin riittävyys kulloinkin muodostuu kriittiseksi, riippuu täysin sovelluksesta. Suurissa tietokantajärjestelmissä kuten henkilörekistereissä saattaa tilankäyttö muodostaa pullonkaulan, kun taas liikennevälineiden kuten autojen tai vaikkapa risteilyohjusten ohjausjärjestelmissä nopeus on usein ratkaiseva tekijä. On siis pyrittävä löytämään kulloiseenkin tilanteeseen sopiva tasapaino eri resurssien tarpeiden välillä.

Tässä monisteessa keskitytään kuitenkin tarkastelemaan yleisimmin tarkasteltua resurssia, aikaa. Algoritmin aikakompleksisuus eli aikavaativuus ilmoitetaan syötteen koon n funktiona T(n). Syötteen koko ei useimmiten tarkoita syöttöalkioiden lukumäärää. Esimerkiksi lajittelussa syötteen koko on lajiteltavien alkioiden lukumäärä, kun taas n-kertoman aikakompleksisuutta määrättäessä syötteen koko on luku n. Aikakompleksisuuden yksikkönä ei yleensä käytetä todellista aikaa, koska se riippuu niin paljon käytettävästä tietokoneesta, vaan laitteistosta riippumatonta keskeisten alkeisoperaatioiden määrää. Eri tehtävissä on tarkoituksenmukaista tarkastella eri operaatioita.

Esimerkki. Polynomin arvon laskeminen annetussa pisteessä. Yleinen n:nnen asteen polynomi on muotoa

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 \quad (a_i \in R)$$

Polynomi identifioidaan kertoimilla a_i, i=0,1,...,n, jotka tulee antaa moduulille parametreina. Lisäksi moduuli tarvitsee parametrina pisteen x, jossa polynomin arvo lasketaan. Todellisissa ohjelmointikielissä muuttujien nimiä ei voi indeksoida, kuten seuraavissa moduuleissa on tehty. Tällöin kertoimet a_i voitaisiin tallentaa esim. taulukkoon a.

```
MODULE P(a0 ,..., an , x) RETURNS Pn(x)
    summa := a0
    FOR i := 1,...,n DO
        (*  lasketaan ensin x^i ja tallennetaan se muuttujaan xpot  *)
        xpot := 1
        REPEAT i TIMES xpot := xpot * x ENDREPEAT
        summa := summa + ai * xpot
    ENDFOR
    RETURN summa
ENDMODULE
```

Keskeisiä alkeisoperaatioita algoritmissa näyttävät olevan kerto- ja yhteenlaskut. Tehtävän koko määräytyy polynomin asteen n mukaan: mitä suurempi n, sitä useampi silmukan kierros. Aikakompleksisuutta on siis tarkoituksenmukaista arvioida kerto- ja yhteenlaskujen määränä n:n suhteen. Algoritmissa suoritetaan kullakin FOR-silmukan kierroksella i+1 kertolaskua ja yksi yhteenlasku. Yhteensä algoritmissa suoritetaan siis $T_1(n) = 2+3+...+(n+1) = n(n+3)/2$ kertolaskua[7] ja $T_2(n) = n$ yhteenlaskua.

[7] Lausekkeen arvo saadaan suoraan taulukkokirjasta – kyseessähän on aritmeettisen sarjan summa.

Algoritmi on hyvin yksinkertainen, ja sitä voidaan helposti tehostaa. Luvun x i:ttä potenssia ei ole tarpeen laskea joka kierroksella alusta asti, vaan se saadaan laskettua kertomalla edellisen kierroksen potenssi x:llä. Siis:

```
MODULE P(a0 ,..., an , x) RETURNS Pn(x)
        summa := a0
        xpot := 1
        FOR i := 1,...,n DO
                xpot := xpot * x
                summa := summa + ai * xpot
        ENDFOR
        RETURN summa
ENDMODULE
```

Tässä versiossa kertolaskuja suoritetaan vain kaksi kullakin kierroksella, siis $T_1(n) = 2n$. Yhteenlaskujen määrä on muuttumaton eli $T_2(n) = n$.

Huomaa, että todellisissa ohjelmointikielissä ei voi käyttää indeksoituja muuttujia, joten polynomin kertoimet tulisi välittää moduulille esim. 1-ulotteisessa taulukossa A, jolloin algoritmi voisi näyttää esim. seuraavalta:

```
MODULE P(A, x) RETURNS P(x)
        summa := A[1]
        xpot := 1
        FOR i := 2,...,A.length DO
                xpot := xpot * x
                summa := summa + A[i] * xpot
        ENDFOR
        RETURN summa
ENDMODULE
```

3.2.1 Kompleksisuuden kertaluokat

Tarvittavien operaatioiden tarkan lukumäärän sijasta usein riittää tarkastella lukumäärän suuruusluokkaa. Tarkan suoritusaikalausekkeen laskeminen on usein vaikeaa, ja pienet erot algoritmien tehokkuudessa ovat yleensä merkityksettömiä. Tärkeämpiä ovat erot algoritmien muissa ominaisuuksissa, kuten luotettavuudessa ja ylläpidettävyydessä. Lisäksi pienet tehokkuuserot algoritmeissa hukkuvat usein jo tietokoneiden tehokkuuseroihin. ***Asymptoottinen*** kompleksisuustarkastelu kertoo kuinka algoritmi käyttäytyy, kun syötteen koko n kasvaa yhä suuremmaksi. Laskettaessa suoritusaikalausekkeita tarkastellaan yleensä vain niitä lausekkeiden osia, jotka suurilla n:n arvoilla dominoivat lauseketta. Dominoiva tekijä määrää täysin algoritmin asymptoottisen käyttäytymisen. Näin lausekkeita voidaan yksinkertaistaa tarkkuuden liiaksi kärsimättä.

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.

Esimerkki. Eri suoritusaikalausekkeiden vertailua.

$\log_2 n$	n	n^2	$n^2 + 5n + 3$	2^n
0	1	1	9	2
n. 3	10	100	153	1024
n. 7	100	10 000	10 503	n. 10^{30}
n. 10	1 000	1 000 000	1 005 003	...
n. 13	10 000	100 000 000	100 050 003	
n. 17	100 000	10 000 000 000	10 000 500 003	
n. 20	1 000 000	1 000 000 000 000	1 000 005 000 003	

Taulukosta nähdään, että logaritminen lauseke kasvaa erittäin hitaasti, kun taas eksponentiaalinen lauseke kasvaa äärimmäisen nopeasti. Polynomilausekkeissa korkeimman n:n potenssin omaava termi dominoi, eivätkä muut termit juurikaan vaikuta lausekkeen arvoon suurilla n:n arvoilla; esimerkiksi $T(n) = n^2+5n+3$ on likimain n^2, kun n on suuri. Sanotaan, että T(n) on suuruusluokkaa n^2, ja merkitään $T(n) \sim n^2$. Myöskään korkeimman potenssin kerroin ei muuta lausekkeen asymptoottista käyttäytymistä merkittävästi; oleellista on, että kun n kasvaa, mikä tahansa n:n toisen asteen polynomi esim. n^2, n^2+5n+3 tai vaikkapa $100n^2$, kasvaa siihen nähden neliöllisesti: esimerkiksi n:n arvon kaksinkertaistuessa toisen asteen polynomin arvo nelinkertaistuu, ja n:n kymmenkertaistuessa se satakertaistuu.

Algoritmin asymptoottinen kompleksisuus esitetään usein muodossa: algoritmin kompleksisuus on samaa suuruusluokkaa kuin funktio f(n) tai antamalla vain yläraja: kompleksisuus on enintään f(n). Kompleksisuuden tarkka matemaattinen määrittely:

Määritelmä. Olkoot T ja f positiiviarvoisia funktioita. Sanotaan, että

- T(n) on ***suuruusluokkaa*** f(n), merkitään $T(n)=\Theta(f(n))$ tai $T(n) \sim f(n)$, jos on olemassa sellaiset positiiviset vakiot k, K ja n_0, että $k \cdot f(n) \leq T(n) \leq K \cdot f(n)$ aina kun $n>n_0$.
- ***yläraja*** T(n):lle on f(n), merkitään $T(n)=O(f(n))$, jos on olemassa sellaiset positiiviset vakiot K ja n_0, että $T(n) \leq K \cdot f(n)$ aina kun $n > n_0$. Tällöin sanotaan, että T(n) on **Ordo** f(n).

Tässä monisteessa rajoitutaan tarkastelemaan vain ensimmäisen kohdan mukaista asymptoottisesti tarkkaa rajaa käyttäen merkintää ~ . Seuraavassa luetellaan tärkeimpiä nimityksiä algoritmien aikakompleksisuudelle (huonommuusjärjestyksessä). Olkoon vakio $c > 1$. Algoritmi on

eksponentiaalinen,	jos $T(n) \sim c^n$,
polynomiaalinen,	jos $T(n) \sim n^c$
lineaarinen,	jos $T(n) \sim n$,
logaritminen,	jos $T(n) \sim \log_c n$,
vakioaikainen,	jos $T(n) \sim 1$ (aikakompleksisuus ei riipu syötteen koosta)

Huomattakoon, että logaritmin kantaluvulla ei asymptoottisessa tarkastelussa ole merkitystä – erikantaiset logaritmit eroavat toisistaan vain vakiokertoimen verran, koska $\log_a n=\log_c n/\log_c a$. Ellei logaritmin kantalukua ole merkitty näkyviin, sen oletetaan olevan kaksi.

Yleisesti voidaan osoittaa, että jos $T(n)=a_k n^k+ a_{k-1}n^{k-1}+...+ a_1 n^1+ a_0$, missä kaikki kertoimet a_i ovat reaalilukuja ja $a_k>0$, niin silloin $T(n) \sim n^k$

Esimerkki. Edellä olevan polynomin arvon laskeminen annetussa pisteessä. Sen ensimmäinen versio sisälsi n(n+3)/2 kertolaskua ja n yhteenlaskua. Tässä kertolaskut dominoivat, joten riittää tarkastella niitä eli pitää määrätä funktion T(n)=n(n+3)/2 suuruusluokka. Yllä olevan tuloksen nojalla $T(n) \sim n^2$.

Esimerkki. Vaihtolajittelun (kappale 2.8.2.2) aikakompleksisuus. Lajittelutehtävän koko on tietenkin järjestettävien alkioiden määrä. Lajittelutehtävässä keskeinen alkeisoperaatio on kahden alkion vertailu niiden keskinäisen järjestyksen määräämiseksi. Parittaisten vertailujen lisäksi tässä algoritmissa tehdään alkioiden paikan vaihtoja, mutta niiden määrä on riippuvainen vertailujen määrästä: vaihtoja tehdään korkeintaan yhtä monta kuin vertailuja. Lajittelutehtävässä riittääkin yleensä tarkastella parittaisten vertailujen määrää. Vaihtolajittelun aikavaativuus nähdään seuraavasti: Lajitteluun tarvitaan n–1 kierrosta (i osoittaa kierrokset). Kullakin kierroksella tehdään n-i vertailua (k osoittaa vertailut). Vertailuja tehdään kaikkiaan (n–1)+(n–2)+...+1 = n(n–1)/2 vertailua (kyseessä on jälleen aritmeettisen sarjan summa). Siis $T(n)=n(n–1)/2 \sim n^2$. Parhaimpien lajittelualgoritmien kompleksisuus on n log n, joista esitetään seuraavassa ns. limityslajittelu.

Esimerkki. Hanoin tornit on malliesimerkki siitä, miten vaikeaan ongelmaan on mahdollista kirjoittaa kompakti ja lyhyt rekursiivinen ratkaisu ja miten sen aikakompleksisuus $T(n) = 2^n – 1$ voidaan määrätä matemaattisesti. Hanoin tornit käsitellään kohdassa 3.2.3.

Esimerkki. Todellisia suoritusaikoja, kun prosessorin nopeus on 1 MHz.

n \ T(n)	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3 µs	10 µs	30 µs	100 µs	1 ms	1 ms
100	7 µs	100 µs	700 µs	10 ms	1 s	410^{22} a
1000	10 µs	1 ms	10 ms	1 s	17 min	10^{226} a
10^4	13 µs	10 ms	130 ms	100 s	12 d	
10^5	17 µs	100 ms	1,7 s	2,8 h	32 a	
10^6	20 µs	1 s	20 s	12 d	310^4 a	
10^7	23 µs	10 s	4 min	3,2 a	310^7 a	
10^8	27 µs	100 s	44 min	317 a	310^{10} a	
10^9	30 µs	17 min	8,3 h	310^4 a	310^{13} a	

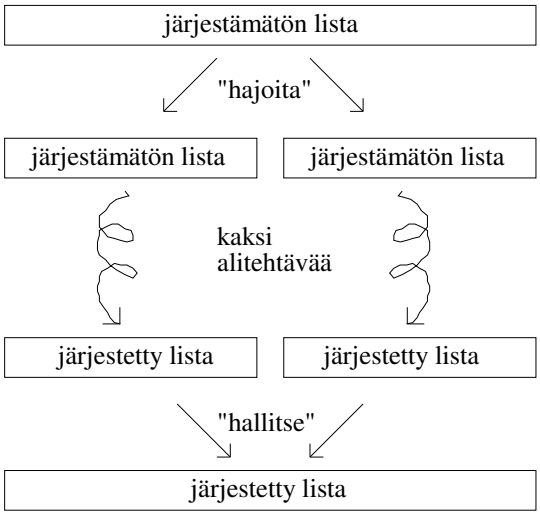
Taulukosta (a=vuosi, d=päivä) nähdään, että eksponentiaaliset algoritmit on mahdollista suorittaa kohtuullisessa ajassa vain hyvin pienillä n:n arvoilla, koska niiden suoritusaika kasvaa valtavan nopeasti n:n kasvaessa. Ne ovat useimmiten käytännössä kelvottomia. Prosessorin nopeuden kasvattaminenkaan ei sanottavammin auta: jos Hanoin tornien ongelmaa ratkotaan miljoona siirtoa sekunnissa, tunnissa ratkeaa 31 kiekon tehtävä. Vaikka prosessorin teho kasvaisi tästä miljoonakertaiseksi (jolloin sen kellotaajuus olisi 1 THz !), tunnissa ratkeaisi vain 51 kiekon tehtävä. Taulukon lukujen tarkoitus on havainnollistaa eri kompleksisuusluokkien eroja. On tietenkin selvää, että algoritmin vaatiman todellisen ajan määrään vaikuttavat kellotaajuuden lisäksi monet muut asiat; esim. välimuistin (cache) määrä, keskusmuistin koko, muistin käsittelyn nopeus (väylän leveys),…. Esimerkiksi, jos verrataan 1 GHz:n ja 2 GHz:n koneita, niin 2 GHz:n kone ei ole käytännössä kaksi kertaa niin nopea kuin 1 GHz:n kone.

Polynomiaaliset algoritmit ovat enimmäkseen käyttökelpoisia, varsinkin jos eksponentti on pieni. Polynomiaalisilla algoritmeilla käyttökelpoisuuden raja tulee vastaan vasta suurilla n:n arvoilla. Lineaariset algoritmit ovat nopeita myös suurilla n:n arvoilla, logaritmisten algoritmien suoritusnopeuteen tehtävän koko vaikuttaa vain hyvin vähän, ja vakioaikaisten algoritmien suoritusnopeus ei riipu lainkaan tehtävän koosta.

3.2.2 Hajota ja hallitse

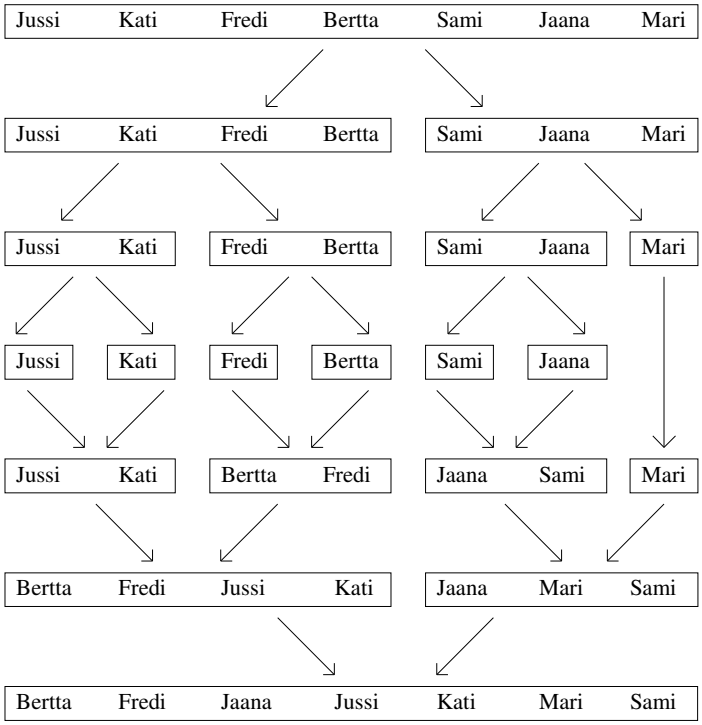
Algoritmin kompleksisuutta voidaan usein parantaa erilaisin menetelmin. Eräs tehokkaimpia tapoja on jo muinaisilta roomalaisilta tuttu ***hajota ja hallitse*** (lat. *divide et impera*; engl. divide and conquer) -periaate.

Edellä saatiin vaihtolajittelun asymptoottiseksi aikakompleksisuudeksi n^2, mutta lajittelutehtävän voi ratkaista tehokkaamminkin, reduktiivisesti hajota ja hallitse -periaatetta käyttäen. Nimilista voidaan lajitella jakamalla nimilista kahtia, lajittelemalla kumpikin puolisko jollakin tavalla, ja limittämällä lopuksi lajitellut puoliskot taas yhteen:



Jos keskellä kaksi pienempää lajittelutehtävää (alitehtävää) ratkaistaan jotenkin, menetelmä selvästikin toimii. Modulaarisuusperiaatteen mukaan alitehtävät voidaan ratkaista millä tahansa tavalla, joten erityisesti ne voidaan ratkaista samalla tavalla kuin koko tehtäväkin. Tällaista rekursiivista menetelmää sanotaan ***limityslajitteluksi***. Sen idea on siinä, että hyvin lyhyiden listojen järjestäminen on triviaalia, ja kaksi järjestettyä listaa on helppo limittää yhdeksi järjestetyksi listaksi.

Esimerkki. Tarkastellaan jo aiemmin esillä ollutta seitsemän nimen listan lajittelua:



Koska seitsemän ei ole kakkosen potenssi, ei nimiä voi aina jakaa tasan kahteen. Itse asiassa menetelmä toimii riippumatta siitä mistä kohtaa lista jaetaan, mutta paras tulos saavutetaan, kun jako suoritetaan mahdollisimman tasan. Rekursiiviset kutsut voidaan päättää tilanteeseen, jossa lajiteltava osa sisältää ainoastaan yhden nimen, jolloin tehtävä on triviaali. Kirjoitetaan proseduuri, joka järjestää parametrina saamansa listan limityslajittelun avulla:

```
MODULE limilaji(lista)
    IF listan pituus > 1 THEN
        Halkaise lista kahtia listoiksi lista1 ja lista2
        limilaji(lista1)
        limilaji(lista2)
        lista := limitä(lista1, lista2)
    ENDIF
ENDMODULE
MODULE limitä(järjestetyt listat lista1 ja lista2) RETURNS järjestetty lista
    (* Limitysmoduulin toteutus jätetään harjoitustehtäväksi *)
ENDMODULE
```

Koska limityslajittelu on rekursiivinen algoritmi, sen kompleksisuuden analysoiminen on helpointa palautuskaavan avulla. Kompleksisuutta laskettaessa voidaan olettaa, että n on kakkosen potenssi. Lista halkaistaan aina kahteen yhtäsuureen osaan ja halkaisun voidaan olettaa olevan vakioaikainen operaatio. Limityslajittelussa parittaisia vertailuja tarvitaan vain limitysvaiheessa. Limitys voidaan helposti toteuttaa limitettävien alkioiden kokonaismäärään nähden lineaarisessa ajassa. Merkitään n:n alkion lajittelussa tarvittavien parittaisten vertailujen määrää T(n):llä. Silloin

T(n)	= T(n/2)	listan alkuosan rekursiivinen lajittelu
	+ T(n/2)	listan loppuosan rekursiivinen lajittelu
	+ n	limitys
	= 2T(n/2) + n	

Soveltamalla palautuskaavaa uudestaan (sijoittamalla edelliseen n:n paikalle n/2) saadaan T(n/2) = 2T(n/4) + n/2, joten

$$
\begin{aligned}
T(n) &= 2(2T(n/4) + n/2) + n \\
&= 4T(n/4) + 2n \\
&= 2^2T(n/2^2) + 2n \\
&= 4(2T(n/8) + n/4) + 2n \\
&= 8T(n/8) + 3n \\
&= 2^3T(n/2^3) + 3n \\
&= \ldots \\
&= 2^mT(n/2^m) + mn
\end{aligned}
$$

Triviaali tapaus on yhden alkion lajittelu. Mutta mikä on m, kun $n/2^m = 1$? Logaritmin määritelmän mukaan

$$n/2^m = 1 \Leftrightarrow 2^m = n \Leftrightarrow m = \log n, \quad \text{(2–kantainen logaritmi)}$$

joten

$$
\begin{aligned}
T(n) &= 2^{\log n}\,T(1) + n \log n \\
&= nT(1) + n \log n \\
&= n \log n,
\end{aligned}
$$

koska T(1) = 0. Limityslajittelun aikakompleksisuus on siis $T(n) \sim n \log n$. Kuinka paljon parempi limityslajittelu on verrattuna vaihtolajitteluun (2.8.2.2), jolle T(n)=n(n–1)/2. Limityslajittelu on vaihtolajittelua parempi, kun

$$n \log n < n(n-1)/2 \Leftrightarrow n \geq 7.$$

Käytännössä raja on korkeampi, koska limityslajittelussa on enemmän erilaisia oheistoimenpiteitä (esimerkiksi rekursiivisten kutsujen toteuttaminen) kuin vaihtolajittelussa. Mutta mitä suurempi n, sitä parempi limityslajittelu on. Esimerkiksi kun n = 100 000 ja prosessori suorittaa miljoona parittaista vertailua sekunnissa, vaihtolajittelu kestää puolisentoista tuntia mutta rekursiivinen lajittelu alle sekunnin. Oleellista on, että ero kasvaa mielivaltaisen suureksi n:n kasvaessa. Jos esimerkiksi 1,2 miljardin ihmisen Kiinan valtion väestökirjanpitäjä käyttäisi vaihtolajittelua kansalaisten järjestämiseen henkilötunnuksen mukaan, kuluisi siihen aikaa noin 20 000 vuotta, kun taas limityslajittelulla hommasta selvitään noin kymmenessä tunnissa! Limityslajittelu onkin asymptoottisesti optimaalinen lajittelumenetelmä, ja se on käytännössäkin tehokkaimpia tunnettuja lajittelumenetelmiä ja yleisesti käytössä.

Esimerkki. Tarkastellaan toisena esimerkkinä hajota ja hallitse -periaatteesta kahta erilaista funktion $\exp(n)=2^n$ arvon laskevaa algoritmia. Helppo rekursiivinen toteutus saadaan suoraan eksponenttifunktion määritelmästä:

$$2^n = \begin{cases} 1, & n = 0 \\ 2 \times 2^{n-1}, & n \geq 1 \end{cases}$$

```
(* Alkuehto: n >= 0 *)
MODULE exp(n) RETURNS 2^n
        IF n = 0 THEN
                RETURN 1
        ELSE
                RETURN 2 * exp(n–1)
        ENDIF
ENDMODULE
```

Algoritmin kompleksisuus (suoritettujen kertolaskujen lukumäärä) n:n funktiona saadaan taas palautuskaavasta:

```
T(n)    = T(n–1) + 1
        = T(n–2) + 1 + 1
        = T(n–3) + 1 + 1+ 1
        = ...
        = T(0) + n
        = n,
```

koska T(0) = 0. Lasketaan seuraavaksi sama funktio hajota ja hallitse -periaatteella 'tasapainottaen'.

$$2^n = \begin{cases} 1, & n = 0 \\ 2^{n//2} \times 2^{n//2}, & n \geq 1 \text{ ja parillinen} \\ 2^{n//2} \times 2^{n//2} \times 2, & n \geq 1 \text{ ja pariton,} \end{cases}$$

missä merkintä // tarkoittaa sellaista jakolaskua, joka antaa tulokseksi jakolaskun kokonaisosan (esim. 5//2=2). Kirjoitetaan tämän pohjalta

```
(* Alkuehto: n >= 0 *)
MODULE exp_hh(n) RETURNS 2^n
        IF n = 0 THEN
                RETURN 1
        ELSE
                m: = exp_hh(n//2)
                IF n on parillinen THEN
                        RETURN m*m
                ELSE
                        RETURN 2*m*m
                ENDIF
        ENDIF
ENDMODULE
```

Tämän algoritmin kompleksisuuden (kertolaskujen ja jakolaskujen määrän) analysoimiseksi oletetaan ensin, että n on kakkosen potenssi. Silloin

```
T(n)  = 1               jakolasku n//2 (rinnastuu vaativuudeltaan kertolaskuun)
      + T(n/2)          rekursiivinen kutsu
      + 1               kertolasku m*m
      = T(n/2) + 2
      = T(n/4) + 2 + 2
      = T(n/2^2) + 2*2
      = T(n/2^3) + 2*3
      = ...
      = T(n/2^m) + 2m
```

Nyt T(1) = 3 ja $n/2^m = 1 \Leftrightarrow 2^m = n \Leftrightarrow m = \log n$, joten

```
T(n)  = T(1) + 2 log n = 3+2 log n
```

Silloin kun n ei ole kakkosen potenssi, tarvitaan aina välillä yksi kertolasku enemmän. Voidaan osoittaa, että yleisestikin $T(n) \leq 3 \log n$ eli joka tapauksessa

$$T(n) \sim \log n.$$

Vertaamalla algoritmien kompleksisuuslausekkeita voidaan todeta, että jälkimmäinen versio on nopeampi kuin edellinen. Pienillä n:n arvoilla ero on merkityksetön, mutta jo n:n arvolla 50 parempi versio on noin kolme kertaa nopeampi kuin ensin esitetty suoraviivainen algoritmi exp.

Algoritmien modifioimisessa paremman suorituskyvyn saamiseksi on kuitenkin oltava tarkkana. Esimerkiksi eksponenttifunktion laskennassa oleellista on juuri 'tasapainotus', kertaalleen lasketun arvon exp_hh(n//2) hyväksikäyttö. Pelkkä keskeltä halkaisu ei tuota toivottua tulosta, vaan päinvastoin huonontaa tilannetta: määrittelyn

$$2^n = \begin{cases} 1, & n = 0 \\ 2, & n = 1 \\ 2^{n//2} \times 2^{(n+1)//2}, & n > 1 \end{cases}$$

pohjalta kirjoitetusta algoritmista tulee noin kolme kertaa algoritmia exp hitaampi!

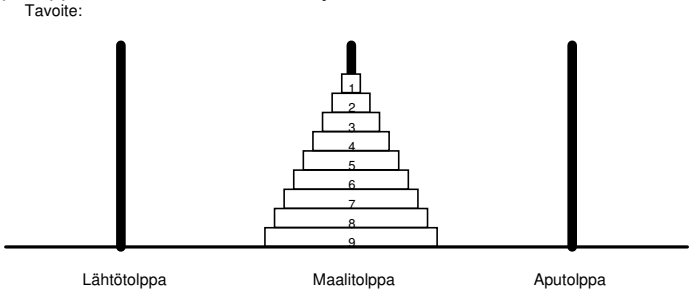
3.2.3 Hanoin tornien ongelma

Seuraavaksi esitetään esimerkki, joka ratkeaa luonnollisesti käyttäen rekursiota ja johon tehtävän luonteen vuoksi on hankalahkoa keksiä iteratiivista algoritmia.

Esimerkki. Hanoin tornit. Hanoin tornien ongelma on saanut alkunsa Hanoissa sijaitsevaan munkkiluostariin liittyvästä legendasta. Luostaria ympäröi muuri, jossa on kolme tolppaa. Yhdessä tolpassa on 64 reiällistä erisuurta kivikiekkoa päällekkäin, jotka on ladottu tolppaan suuruusjärjestyksessä suurin kiekko alimmaiseksi, ja kaksi muuta tolppaa ovat tyhjiä. Legendan mukaan luostarin Buddha-munkit saivat jumalallisessa ilmestyksessä tehtäväkseen ratkaista, miten siirtää yhden tolpan kiekot toiseen tolppaan kiekko kerrallaan. Kiekkoja saa kasata alkuperäisen tolpan ja tavoitteena olevan tyhjän tolpan lisäksi yhteen aputolppaan väliaikaiseksi aputorniksi. Lisäksi taivaallisen isonveljen pitää pystyä tarkkaan vahtimaan suoritusta, joten milloinkaan ei suurempi kiekko saa olla pienemmän päällä missään tolpassa. Munkkien käsityksen mukaan kiekkojen siirräntä on hengellisesti kohottavaa toimintaa. Pyhä missio on muutenkin varsin kunnioitettava, sillä legendan mukaan pappien saadessa tehtävänsä päätökseen koittaa maailmanloppu. Lukijan ei kannata kuitenkaan vielä ahdistua, vaan malttaa mielensä monisteen seuraavan luvun alkuun.

Ongelman hengellisistä ulottuvaisuuksistaan riisuttu formaalinen perusasetelma on seuraava: Kolmessa tolpassa on kiekkoja. Kiekkoja voidaan siirtää yksitellen mistä tolpasta tahansa johonkin muuhun tolppaan sillä rajoituksella, että suurempi kiekko ei saa milloinkaan olla pienemmän yläpuolella. Aluksi kaikki kiekot ovat lähtötolpassa suuruusjärjestyksessä:

(Piirrosteknisistä syistä kuvan tehtävässä on vain yhdeksän kiekkoa.) Tavoitteena on siirtää kaikki kiekot maalitolppaan käyttäen apuna aputolppaa siten, ettei mainittua rajoitusta hetkellisestikään rikota:

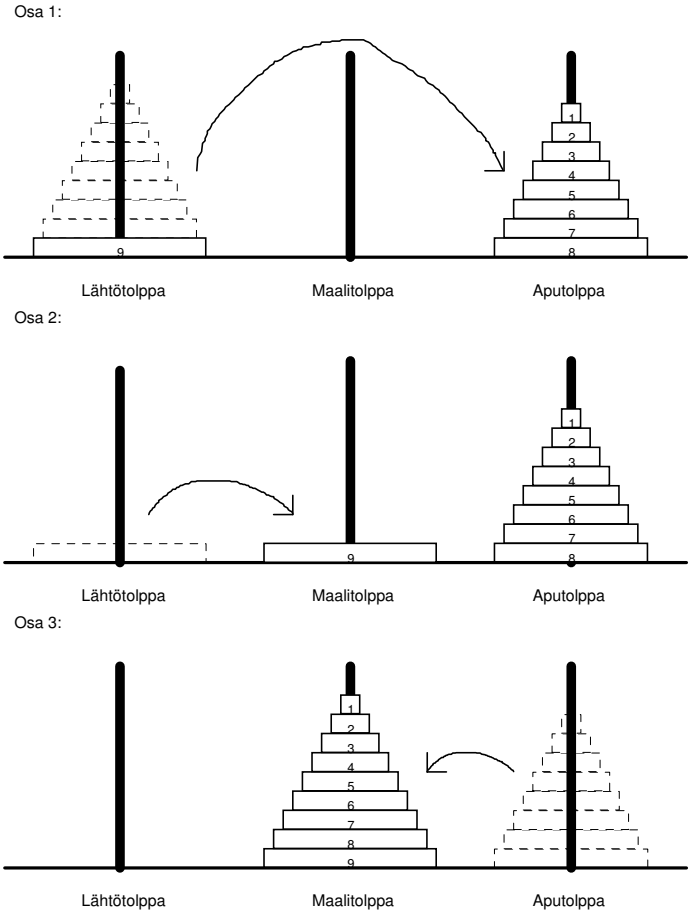


Tehtävä on vaikea. Se on vaikea siinäkin mielessä, että ratkaisualgoritmi saattaa aluksi tuntua vaikealta keksiä, mutta etenkin siinä mielessä, että ratkaisun tuottaminen on hyvin monimutkaista. Jos kiekkojen määrä on vähänkin suurehko, niin siirtojen määrä kasvaa niin suureksi, että yksittäisen siirron yhteyttä kokonaisuuteen on äärimmäisen vaikea nähdä. Juuri siksi on niin vaikeata keksiä iteratiivinen ratkaisu: on kovin vaikea nähdä miten yksittäinen siirto vie tilannetta kohti ratkaisua.

Sen sijaan reduktiivinen ajattelutapa tuottaa tulosta tässä tapauksessa suorastaan hämmästyttävän helposti. Tulee vain huomata, että tehtävän jokaisen ratkaisun – olkoon erilaisia ratkaisuja kuinka monta tahansa – on välttämättä kuljettava seuraavien vaiheiden kautta:

1. Siirrä tavalla tai toisella 63 päällimmäistä kiekkoa lähtötolpasta aputolppaan.
2. Siirrä alimmainen kiekko lähtötolpasta maalitolppaan.
3. Siirrä aputolppaan siirretyt 63 kiekkoa maalitolppaan.

Seuraavassa kuvassa on esitetty yhdeksän kiekon ongelman ratkaisun kulku vastaavien vaiheiden kautta.



Erityisesti tulee huomata, että tehtävään liittyvää rajoitusta – suurempi kiekko ei saa olla pienemmän päällä – ei milloinkaan rikota. Itse asiassa suurin kiekko voidaan ensimmäisessä ja kolmannessa vaiheessa unohtaa tyystin; koska se on suurin, ei mainittua rajoitusta mitenkään voida rikkoa pelkästään muita kiekkoja siirtelemällä. Niinpä osatehtävät voidaan samastaa alkuperäiseen ongelmaan, vain kiekkojen määrä on nyt muuttunut. Ratkaisu voidaan yleistää n kiekolle seuraavasti. Olkoot kiekot numeroitu juoksevasti pienimmästä suurimpaan.

```
MODULE Hanoi(n kiekkoa)
    Hanoi(n–1 kiekkoa)
    Siirrä kiekko numero n Lähtötolpasta Maalitolppaan
    Hanoi(n–1 kiekkoa)
ENDMODULE
```

Algoritmi on selvästi rekursiivinen, ja rekursiokutsussa esiintyvät alitehtävät ovat aidosti alkuperäistä yksinkertaisempia, väheneehän siirrettävien kiekkojen määrä aina yhdellä. Mutta algoritmi ei suinkaan ole vielä valmis: triviaali tapaus puuttuu. Jos tehtävässä on vain yksi kiekko, on tehtävä helppo – siirretään vain ainoa kiekko Lähtötolpasta Maalitolppaan. Yhden kiekon Hanoi onkin aivan kelvollinen triviaali tapaus. Mutta vieläkin helpompaa tehtävän ratkaiseminen on, jos kiekkoja ei ole ollenkaan – nollan kiekon siirtämiseksi kun ei tarvitse tehdä mitään!

```
MODULE Hanoi(n)
    IF n > 0 THEN
        Hanoi(n–1)
        Siirrä kiekko numero n Lähtötolpasta Maalitolppaan
        Hanoi(n–1)
    ENDIF
ENDMODULE
```

Moduuli ei kuitenkaan aivan toimi. Nimittäin, siinähän siirretään kiekkoja ainoastaan Lähtötolpasta Maalitolppaan, ja yllä olevasta kuvasta selvästi nähdään, että kiekkoja tulee siirtää toisinaan myös Lähtötolpasta Aputolppaan ja Aputolpasta Maalitolppaan. Yksinkertaistimme tehtävää siis hieman liikaa – unohdimme tolppien muuttuvat roolit. Tarkennetaan moduulia hieman:

```
(* Moduuli siirtää n kiekkoa tolpasta Lähtö tolppaan Maali käyttäen apuna tolppaa Apu niin, että missään
vaiheessa missään tolpassa ei isompi kiekko ole pienemmän kiekon päällä.
Alkuehto: n>=0 *)
MODULE Hanoi(n, Lähtö, Maali, Apu)
    IF n > 0 THEN
        Hanoi(n–1, Lähtö, Apu, Maali)
        Siirrä kiekko numero n tolpasta Lähtö tolppaan Maali
        Hanoi(n–1, Apu, Maali, Lähtö)
    ENDIF
ENDMODULE
```

Siinä se! 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). Tässä on hyvä huomata, että komennossa Siirrä … (eo. kuvan osa 2) kiekko numero n on Lähtötolpan päällimmäinen kiekko ja että rekursiivisissa kutsuissa parametrien järjestys kertoo niiden loogisen merkityksen: esim. Hanoi(n-1, Lähtö, Apu, Maali) saa aikaan sen että (osa 1) siirretään n–1 kiekkoa tolpasta Lähtö tolppaan Apu käyttäen apuna tolppaa Maali. Huomaa, että kutsu Hanoi(0, X, Y, Z) ei tee yhtään mitään ja kutsu Hanoi(1, X, Y, Z) saa aikaan vain yhden tolpan siirron X:stä Y:hyn.

Esimerkiksi 9 kiekon tehtävä redusoituu kolmeen osaan:

Hanoi(9, Lähtö, Maali, Apu)

1. Hanoi(8, Lähtö, Apu, Maali)
2. Siirrä kiekko numero 9 tolpasta Lähtö tolppaan Maali
3. Hanoi(8, Apu, Maali, Lähtö)

Tällöin ensimmäisessä kutsussa (osa 1) ratkaistaan Hanoin tornien ongelma kahdeksalle tehtävälle: siirretään alkuperäisen Lähtötolpan 8 päällimmäistä kiekkoa tolppaan Apu käyttäen apuna tolppaa Maali. Kun tämä tehtävä on suoritettu kokonaisuudessaan (tästähän syntyy puolestaan kaksi 7 kiekon tehtävää), suoritetaan keskimmäinen lause eli yhden kiekon siirto (osa 2). Seuraavaksi suoritetaan jälleen 8 kiekon tehtävä: siirretään tolpasta Apu tolppaan Maali käyttäen apuna tolppaa Lähtö (osa 3).

Huom. Moduulissa Lähtö, Maali ja Apu ovat muodollisia parametreja (muuttujia), joilla on eri arvot algoritmin eri vaiheissa. Näin ollen nimeämmekin tolpat esim. merkkijonoilla ”Tolppa 1”, ”Tolppa 2 ja ”Tolppa 3”, jolloin esim. 5-kiekon tehtävä ratkeaa kutsulla:

Hanoi(5, ”Tolppa 1”, ”Tolppa 2”, ”Tolppa 3”)

Tämä kertoo (ne voi esim. tulostaa) kaikki tarvittavat siirrot, kun siirretään 5 kiekkoa Tolpasta 1 Tolppaan 2. Tällöin esim. Java-ratkaisussa Lähtö, Maali ja Apu ovat tyyppiä String.

Algoritmin aikakompleksisuutta on tarkoituksenmukaista mitata tarvittavien siirtojen määrällä kiekkojen lukumäärän funktiona. n:n kiekon ongelma redusoituu kolmeen osaan: kahteen (n–1):n kiekon ongelmaan ja yhteen triviaaliin siirtoon. Merkitään n:n kiekon ongelman ratkaisuun tarvittavien siirtojen määrää T(n):llä. Silloin (n–1):n kiekon ongelman ratkaisemiseksi tarvitaan tietenkin T(n–1) siirtoa. Aikakompleksisuudelle voidaan tämän perusteella kirjoittaa rekursiivinen palautuskaava:

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1)+1, \quad n>1. \end{aligned}$$

Kyseessä on rekursiivinen palautuskaava, mutta miten tällaisesta ratkaistaan T(n). Kaavasta voidaan ratkaista T(n) kehittämällä kaavaa edelleen. Sijoittamalla yo. kaavaan n:n paikalle n-1, saadaan T(n-1) = 2T(n-2) + 1 jne, joten:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 4T(n-2) + 2 + 1 \\ &= 4(2T(n-3) + 1) + 2 + 1 \\ &= 8T(n-3) + 4 + 2 + 1 \end{aligned}$$

Seuraavaksi tulee miettiä noudattaako saatu lauseke jotain yleistä säännönmukaisuutta. Kirjoittamalla viimeinen lauseke muotoon $2^3T(n-3) + 2^2 + 2^1 + 2^0$ nähdään, että lausekkeen loppuosaan muodostuu geometrinen sarja, jonka suhdeluku on 2:

$$\begin{aligned} T(n) &= 2^3T(n-3) + 2^2 + 2^1 + 2^0 \\ &= 2^4T(n-4) + 2^3 + 2^2 + 2^1 + 2^0 \\ &= \ldots \\ &= 2^{n-1}T(1) + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 \end{aligned}$$

Koska yhden kiekon ongelman ratkaisemiseksi tarvitaan yksi siirto, T(1) = 1, joten saadaan edelleen[8]

$$\begin{aligned} T(n) &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 \\ &= 2^n - 1. \end{aligned}$$

Esitetty algoritmi on paras mahdollinen. Huomaa, että kyseinen kaava $T(n) = 2^n - 1$ on helppo todistaa todeksi induktiolla, kunhan ko. kaava ensin keksitään! Täten Hanoin tornien ongelman kompleksisuus on eksponentiaalinen, $T(n) \sim 2^n$. Alkuperäisessä ongelmassa kiekkoja oli 64 kpl. Jos yhteen siirto-operaatioon kuluu aikaa yksi sekunti, 64 kiekon siirtämiseen kuluu $T(64) = 2^{64} - 1$ s ≈ 585 miljardia vuotta! Maailmanloppua ei siis ole tarpeen pelätä vielä vähään aikaan. Vaikka tietokone tekisi miljardi siirtoa sekunnissa, niin silti aikaa kuluisi n. 585 vuotta. Huomaa, että algoritmin oikeellisuus voidaan todeta samalla tavalla käyttäen induktiota.

3.2.4 Kelvottomia ongelmia

Church-Turingin teesin mukaan laskettavuus on tietokoneista riippumaton ominaisuus eli kaikki laskettavissa olevat ongelmat ovat ratkaistavissa kaikilla tietokoneilla. Kuitenkin ***kelvollisia*** ongelmia ovat vain sellaiset tehtävät, jotka voidaan ratkaista polynomiaalisessa ajassa. ***Peräkkäislaskennan teesi*** (sequential computation thesis) on vahvempi kuin Church-Turingin teesi. Se sanoo, että kaikki kelvolliset ongelmat ovat ratkaistavissa kaikilla koneilla polynomiaalisessa ajassa. Toisin sanoen sen lisäksi, että tietokoneet voivat simuloida toisiaan, ne voivat tehdä sen kohtuullisen tehokkaasti. Näin ollen ongelman kelvollisuuskaan ei riipu tietokoneesta.

Kompleksisuudeltaan eksponentiaaliset ongelmat ovat yleensä ***kelvottomia*** (infeasible), vaikka riittävän pienillä syötteillä nekin saattavat ratketa kohtuullisessa ajassa. Käyttökelpoisuuden raja vaihtelee tietenkin tehtävittäin. Olemme jo tutustuneet joihinkin kelvottomiin ongelmiin, kuten Hanoin tornien ongelmaan. Mutta myös monet arkipäiväisemmät tehtävät ovat algoritmisesti kelvottomia.

---

[8] Geometrisen sarjan summa löytyy taulukkokirjasta. Se voidaan myös ajatella seuraavasti: $2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0$ vastaa binääriluvun 11...11 (n ykköstä) arvoa. Tätä seuraava binääriluku on 100...00 (ykkönen ja n nollaa), jonka arvo on siis 2^n. Niinpä alkuperäisen lausekkeen arvo on yhtä pienempi.

Esimerkki. Muun muassa seuraaville todellisen elämän ongelmille ei ole (vielä) löydetty polynomiaalista ratkaisua, vaan kaikki tähän mennessä esitetyt algoritmit ovat olleet eksponentiaalisia:

- Shakin pelaaminen.
- *Pakkausongelmassa* (bin-packing) kysytään, voidaanko pakata *n* kpl kuorma-autoja, joilla on tietty kantavuus, siten, että *k* kpl eripainoisia kontteja voidaan kuljettaa samanaikaisesti.
- *Kauppamatkustajan ongelma* (traveling salesperson problem) on selvittää, voidaanko käydä *n* kaupungissa, kun ajomatkan pituus saa olla enintään K kilometriä siten, että jokaisessa kaupungissa käydään vain kerran ja lopuksi palataan lähtökaupunkiin. Usein tarkastellaan myös lyhyimmän reitin löytämistä.
- *Hamiltonin polku* (Hamiltonian cycle) -ongelma on sama kuin kauppamatkustajan ongelma, mutta ajomatkan pituutta ei ole rajoitettu eli kysytään voidaanko ylipäänsä löytää sellaista reittiä, että jokaisessa kaupungissa käydään tarkalleen kerran palaten lopuksi lähtökaupunkiin.
- *Lukujärjestysongelmassa* (timetabling problem) on annettu joukko kursseja, joukko kursseille osallistuvia opiskelijoita sekä joukko mahdollisia kurssin pitoaikoja (kellonaikoja). Tehtävänä on ajoittaa kurssit niin, ettei kenenkään opiskelijan tarvitsisi olla kahdella kurssilla yhtä aikaa.

Kelvottomien ongelmien nopeaa ratkaisemista voidaan kuitenkin yrittää erilaisin keinoin tinkimällä algoritmin perusvaatimuksista. Esimerkiksi tingitään yleisyydestä rajoittumalla ongelman joihinkin nopeasti ratkeaviin erikoistapauksiin. Voidaan myös yrittää löytää algoritmi, joka voidaan suorittaa nopeasti keskimääräisellä syötteellä mutta joka kuitenkin pahimmassa tapauksessa olisi eksponentiaalinen. Voidaan myös tinkiä algoritmin oikeellisuudesta sallimalla likimääräinen ratkaisu tai virheitä pienellä todennäköisyydellä.

- ***Likimääräisalgoritmeilla*** (approximative algorithms) lasketaan vaatimustasoa ja pyritään likimäärin oikeaan tulokseen, jolloin voidaan löytää tehokaskin ratkaisu. Esimerkiksi lukujärjestysongelmassa voidaan sallia muutamia yhteentörmäyksiä, mikä on parempi kuin se, että jäädään kokonaan ilman lukujärjestystä.
- ***Todennäköisyysalgoritmeja*** (probabilistic algorithms) käytettäessä tyydytään algoritmiin, joka on nopea, mutta ei aina toimi oikein. Tällöin virheen todennäköisyyden tulee kuitenkin olla hyvin pieni. Esimerkiksi alkulukutestiin ei ole olemassa polynomiaalista algoritmia (eksponentiaalinen algoritmi kyllä löytyy). Sallimalla algoritmissa virhe, jonka esiintymistodennäköisyys on esimerkiksi 2^{-20}, alkulukutestaukseen voidaan esittää polynominen algoritmi.
- ***Epädeterministisissä*** tehtävissä ratkaisun tiedetään löytyvän joltakin alueelta, ja tiedetään ne keinotkin, joilla ratkaisu löytyy. Ei vain tiedetä, mitä keinoa pitäisi milloinkin soveltaa, eikä kaikkia keinoja ole mahdollista kokeilla joka tapauksessa. Epädeterministisiin tehtäviin kehitettyjä ***heuristisia*** algoritmeja, joissa kulloinkin suoritettavat toimenpiteet valitaan jonkinlaisella peukalosäännöllä – eli hyvällä arvauksella – tarkastellaan lähempää monisteen kuudennessa luvussa.

3.2.5 *P* ja *NP*

Kun puhutaan kompleksisuudesta, symboleilla *P* ja *NP* on tärkeä rooli. Symboli *P* on oikeastaan jo tuttu, se tarkoittaa kaikkien kelvollisten tehtävien joukkoa. Seuraavaksi määritellään nämä joukot:

Määritelmä:

a) Merkitään symbolilla *P* niiden ongelmien joukkoa, joille on olemassa deterministinen polynomiaalinen ratkaisualgoritmi.

b) Merkitään symbolilla *NP* niiden ongelmien joukkoa, joille on olemassa deterministinen polynomiaalinen algoritmi, jolla voidaan tarkastaa ratkaisuehdokkaan oikeellisuus.

Itse asiassa joukko *NP* (nondeterministic polynomial problems) on niiden ongelmien joukko, jotka voidaan ratkaista polynomiaalisessa ajassa epädeterministisellä algoritmilla. Tietenkin kaikki kelvolliset ongelmat (=*P*) kuuluvat joukkoon *NP* eli $P \subseteq NP$. Edellä esitetyt pakkausongelma, kauppamatkustajanongelma, Hamiltonin polku -ongelma ja lukujärjestysongelma kuuluvat joukkoon *NP*. Nimittäin annetun ratkaisuehdokkaan oikeellisuuden testaaminen voidaan suorittaa polynomiaalisessa ajassa, koska riittää vain tarkastaa täyttääkö ratkaisu annetun tehtävämäärittelyn eikä esimerkiksi tarvitse tutkia ratkaisun optimaalisuutta. Kuitenkaan ei tiedetä, kuuluvatko ne joukkoon *P*. Tietojenkäsittelytieteen suuria avoimia kysymyksiä on: onko *P=NP* vai onko *P NP*:n aito osajoukko?

NP-täydelliset (*NP*-complete) ongelmat ovat *NP*-joukon sellaisia (vaikeimpia) ongelmia, joista yhdenkin ratkeaminen polynomiaalisessa ajassa aiheuttaisi sen, että kaikkiin joukon *NP* ongelmiin löydettäisiin polynomiaalinen ratkaisu. Esimerkiksi edellä mainitut ongelmat on osoitettu *NP*-täydellisiksi.

Yleinen käsitys on, että *NP*-täydelliset ongelmat ovat ja pysyvät kelvottomina. Vaikkakaan tätä ei ole todistettu, näin on syytä uskoa. Monet eri alojen asiantuntijat ovat yrittäneet löytää oman erityisalansa *NP*-täydellisiin ongelmiin polynomiaalisia algoritmeja tuloksetta. Toisaalta kukaan ei ole onnistunut todistamaan *NP*-täydellisiä ongelmia kelvottomiksi.

Yhteenvetona peräkkäisalgoritmien kompleksisuudesta voidaan todeta, että kelvollisiin *P*-joukon ongelmiin voidaan löytää aina nopea algoritmi, joihinkin *NP*-vaikeina pidettyihin ongelmiin saatetaan vielä joskus löytää nopea algoritmi, mutta *NP*-täydellisiin ongelmiin nopeaa algoritmia ei uskota koskaan löydettävän.

3.3 Oikeellisuus

Tässä kohdassa tarkastellaan algoritmien oikeellisuutta: oikeellisuuden merkitystä, oikeellisuustyyppejä ja menetelmiä algoritmin toimivuuden ja oikeellisuuden osoittamiseen.

Useimmissa ohjelmissa on virheitä (bugs). Jopa sellaiset ohjelmat, jotka ovat olleet vuosikausia käytössä, voivat tietyissä olosuhteissa toimia väärin. Yleensä suurin osa ohjelmoijan työajasta kuluukin ohjelmien ylläpitoon: virheellisten ohjelmien korjailuun ja muuttuneiden tarpeiden aiheuttamien modifikaatioiden tekemiseen. Virheiden etsintää ja korjausta nimitetään 'debuggaukseksi' (debugging). Hyvin usein virheiden syynä ovat (liian) monimutkaiset algoritmit, joiden toimintaa ei ymmärretä täydellisesti. Tehtävien huolellinen abstrahointi ja selkeä, modulaarinen toteutus ovat mahdollisimman vähävirheisten ohjelmien tuottamisen perusedellytyksiä.

Virheettömien tai mahdollisimman vähävirheisten ohjelmien aikaansaamiseksi on olemassa kaksi perusmenetelmää: ***testaus*** (testing) ja ***oikeaksitodistaminen*** (proving).

3.3.1 Testaus

Tärkein keino ohjelman oikeellisuuden selvittämisessä on ohjelman huolellinen testaus. Testaus suoritetaan tarkasti valitulla testiaineistolla joko suorittamalla ohjelmaa tietokoneessa tai testaamalla ohjelmaa manuaalisesti ns. pöytätestauksella. Testiaineiston tulee olla kattava: tyypillisten tapausten tutkiminen ei riitä, vaan erityisesti on otettava huomioon erilaiset erikoistapaukset. Testiaineisto olisikin hyvä suunnitella ja laatia etukäteen jo ohjelmointivaiheessa, ennen varsinaisen testauksen aloittamista.

Ohjelmissa esiintyvät virheet voivat olla monentyyppisiä:

- Käyttövirheet: ohjelmaa sovelletaan tapaukseen, joka on sen määrittelyalueen ulkopuolella. Käyttövirhe on tietenkin periaatteessa ohjelman käyttäjän tekemä virhe, mutta usein sen syntymiseen myötävaikuttaa ohjelman epäselvä tai epäluonnollinen tehtävänmäärittely. Parannus: mahdollisimman luonnollinen ja tarkasti jäsennelty tehtävänmäärittely. Käyttäjälle pitää käydä selväksi, mihin ohjelma on tarkoitettu.
- Testausvirheet: ohjelma toimii jossakin (erikois)tilanteessa väärin. Tällainen virhe on yleensä helppo havaita, jos vain kyseinen erikoistapaus sisältyy testiaineistoon. Testauksen tulisikin olla mahdollisimman kattava, ja sen suorittamiseen pitää varata riittävästi aikaa, monissa tapauksissa jopa enemmän kuin itse ohjelmointiin.
- Määrittelyvirheet: jotakin (erikois)tapausta ei ole lainkaan otettu huomioon, jolloin ohjelman toiminta kyseisessä tapauksessa on tuntematon. Jos tehtävän määrittelyyn suhtaudutaan yliolkaisesti, voi määrittelyyn jäädä 'aukkoja'. Jokaisen moduulin tehtävä olisikin mietittävä huolellisesti, kiinnittäen erityistä huomiota toisaalta triviaaleina pidettyihin tapauksiin ja toisaalta erilaisiin epätavallisiin tai patologisiin, jopa absurdeihin tapauksiin.

Ohjelman käyttäytymisestä testauksen aikana ja sen tuottamista tuloksista päätellään, toimiiko ohjelma oikein. Testaus varmistaa, että ohjelma toimii valitulla syöteaineistolla oikein, mutta ei takaa sitä, että ohjelma toimii oikein kaikilla syötteillä. Testaus ei siis todista ohjelmaa oikeaksi. Vuosikausia moitteettomasti toiminut ohjelma saattaa osoittautua virheelliseksi. Suurten järjestelmien, kuten esimerkiksi tietokoneiden käyttöjärjestelmien, täydellinen virheettömyys onkin käytännössä mahdotonta. Kun havaittu virhe korjataan, voi muutettu tai lisätty ohjelman osa aiheuttaa uuden virheen jossain toisessa yhteydessä. Suurissa ohjelmistoissa saavutetaankin ennen pitkää dynaaminen tasapainotila, missä jokainen virheiden vähentämiseksi tehty muutos aiheuttaa suunnilleen yhtä monta uutta virhettä.

3.3.2 Oikeaksitodistaminen

Jos ohjelman oikeellisuudesta halutaan ehdoton varmuus, täytyy ohjelma todistaa oikeaksi. Oikeaksitodistamisessa osoitetaan, että ohjelma toimii oikein kaikilla sallituilla syöttötiedoilla. Tarkasteltaessa moduulin, tai sen osan, oikeellisuutta moduulin määrittelyn tulee olla mahdollisimman täsmällinen. Määrittelyssä kerrotaan, kuinka

moduulia käytetään, mitkä ovat sen parametrit rajoituksineen ja se, mitä moduuli tekee. Moduuleja kirjoitettaessa määrittely olisi hyvä kirjoittaa jo ennen ensimmäistäkään koodiriviä. Määrittelyyn kuuluu

- ***alkuehto*** (esiehto, pre-condition), jossa esitetään vaatimukset parametreille. Alkuehto kertoo kaikki kelvolliset alkutilat (so. parametrien arvot), joilla ohjelman on suunniteltu toimivan oikein.
- ***loppuehto*** (jälkiehto, post-condition), joka kertoo, mitkä ehdot ovat voimassa lopuksi, jos alkuehto oli voimassa, kun moduulia lähdettiin suorittamaan.

Suunniteltaessa suuria ohjelmia jaetaan ongelma osaongelmiin, jotka ratkaistaan erikseen käyttäen itsenäisiä, toisistaan riippumattomia moduuleja. Myös ohjelmien oikeaksitodistamisessa moduulijaosta on hyötyä. 'Epävirallinen' todistus syntyy jo moduulia suunniteltaessa: moduuli täyttää sille asetetut toiminnalliset vaatimukset sekä syöttö- ja tulostiedoille asetetut määritykset: alku- ja loppuehdot.

Alkuehto kuvaa laskun tilan ennen moduulin suoritusta. Loppuehto kuvaa taas laskun tilan moduulin suorituksen jälkeen. Loppuehdossa kuvataan siis se tila, mihin moduulin tulisi loppua, jotta algoritmi toimisi oikein. Tarkoituksena on todistaa loppuehdon paikkansapitävyys, kun alkuehto oletetaan todeksi. Usein loppuehdon toteutuvuutta ei voida todeta suoraan, ja tällöin sijoitetaan myös moduulin sisälle ‘kriittisiin kohtiin’ ***väitteitä*** (assertions), joiden avulla pyritään osoittamaan moduulin oikeellisuus.

Hyvään ohjelmointityyliin kuuluu kirjoittaa jokaiselle moduulille alkuehto. Joissakin kielissä alkuehdon paikkansapitävyyden tarkastus suoritetaan automaattisesti, mutta useimmissa ei. Tällöin alkuehto kirjoitetaan moduulin otsikkoriville kommenttina ja tällöin alkuehto toimii informaationa moduulin käyttäjälle siitä, milloin (millä parametrien arvoilla) moduuli toimii oikein. Jos moduuli toimii oikein kaikilla parametrien arvoilla, alkuehtoa ei kirjoiteta.

Esimerkki. 1) n-kertoman palauttavan moduulin kertoma(n) alkuehto on muotoa: n>=0. 2) Taulukon T alkioiden keskiarvon palauttavan moduulin alkuehto on muotoa: T:n pituus >0 eli T.length>0.

Ohjelman oikeaksitodistaminen suoritetaan paloittain tarkastelemalla ohjelman osia (esim. pieni moduuli, toistorakenne); todistamalla osien oikeellisuus pyritään todistamaan koko ohjelman oikeellisuus. Oikeaksitodistaminen on huomattavasti vaikeampaa kuin testaaminen: se on äärimmäisen tarkkaa ja huolellista työtä, ja menetelmät ovat sangen monimutkaisia. Koska valitettavasti myös oikeaksitodistuksessa on mahdollista tehdä virheitä, pysyy täydellinen virheettömyys utopiana ainakin hyvin suurten ohjelmistojen osalta. Ohjelman oikeaksitodistaminen ei ole laskettavissa oleva ongelma, joten todistusprosessia ei ole voitu mekanisoida.

Oikeaksitodistamisen käytännön merkitys onkin kahtaalla:

- **Ennalta ehkäisy**: oikeellisuuden todistamismenetelmien periaatteiden tunteminen ja niiden mielessä pitäminen ohjelman laatimisen aikana ehkäisevät ennalta virheiden syntyä. Alku- ja loppuehtojen laatiminen jo sinällään pakottaa perehtymään tarkemmin annettuun tehtävään.
- **Kriittiset moduulit**: monimutkaisen ohjelmistokokonaisuuden keskeiset tai elintärkeät osat voidaan todistaa oikeiksi, mikä ei toki todista kokonaisuutta

oikeelliseksi, mutta lisää merkittävästi vakuuttautuneisuutta kokonaisuuden oikeellisuudesta.

Seuraavaksi käsitellään oikeellisuustodistuksissa käytettyjä metodeja: induktio, invarianssit ja oikeaksi todistaminen peräkkäisten väitteiden avulla. Induktio ja induktiivinen päättely yleensä ovat keskeisessä asemassa oikeaksitodistamisessa. Selvitetään kuitenkin ensin itse matemaattisen induktion periaate.

3.3.2.1 Induktioperiaate

Olkoon n_0 jokin kokonaisluku ja P jokin kokonaisluvuilla parametrisoitu totuusarvoinen väite. Tarkastellaan väitteen P(n) voimassaoloa kokonaislukujen n, $n \geq n_0$, suhteen. Induktioperiaate on yleinen todistusmenetelmä, jolla voidaan todistaa annetun väitteen P(n) voimassaolo kaikilla kokonaisluvuilla $n \geq n_0$. Tarkasteltava väite voi liittyä hyvin monenlaisiin asioihin. Induktiolla voidaan todistaa esimerkiksi matematiikassa usein esiintyvä aritmeettista sarjaa koskeva tulos: 1+2+…+n=n(n+1)/2, n≥1 (vrt. Matematiikan pk.). Tietojenkäsittelyssä induktiota käytetään usein algoritmien kompleksisuusanalyysissä (kuten seuraavassa esimerkissä käsitelty Hanoin tornien aikakompleksisuus) ja moduulien oikeellisuuden todistamisessa (moduulin tulostapuu oikeellisuus).

Induktioperiaate. Jos seuraavat kaksi ehtoa (i) ja (ii) toteutuvat

(i) $P(n_0)$ on tosi, missä n_0 on kokonaisluku

(ii) Olkoon $k \geq n_0$. Kun oletetaan, että P(k) on tosi, niin sen avulla voidaan todistaa, että myös P(k+1) on tosi

niin silloin

(iii) P(n) on tosi **kaikilla** n:n arvoilla $n \geq n_0$.

Tarkasteltavissa väitteissä usein n_0=0 eli tarkastellaan väitteitä P(0), P(1), ...

Induktiotodistus. Induktioperiaatetta käytetään todistuksissa seuraavasti. Formuloidaan väite P ja parametri n, jonka suhteen todistus suoritetaan. Ensin osoitetaan, että väite pitää paikkansa pienimmällä tarkasteltavalla n:n arvolla n_0:

- ***Lähtökohta***: $P(n_0)$ on tosi (kohta (i))

Sitten osoitetaan kohta (ii) todeksi tekemällä induktio-oletus ja todistamalla sen nojalla induktioväite todeksi.

- ***Induktio-oletus***: oletetaan, että $k \geq n_0$ ja että P(k) on tosi.
- ***Induktioväite***: P(k+1) on tosi.

Lopuksi voidaan induktioperiaatteen nojalla tehdä

- ***Johtopäätös*** (kohta (iii))

Esimerkki. Osoitetaan induktiolla, että Hanoin tornien ongelman aikakompleksisuus on $T(n) = 2^n - 1$, $n \geq 0$. Itse todistus on lyhyt ja yksinkertainen. Vaikeampi ongelma on keksiä todistettava kaava. Väitteenä P(n) on eo. kaavan voimassaolo n:n arvoilla n=0,1,2,... Nyt $n_0 = 0$ eli induktion lähtökohdassa n=0.

Lähtökohta: Kun kiekkoja ei ole yhtään, ei ole mitään tehtävää eli $T(0)=0=2^0 - 1$.
Siis kaava pätee, kun n=0.

Induktio-oletus: $T(k) = 2^k - 1$ $(k \geq 0)$ (eli kaava pätee, kun kiekkoja on k kpl)

Induktioväite: $T(k+1) = 2^{k+1} - 1$.

Induktioväitteen todistus: Tarkastellaan k+1 kiekon tehtävää. Algoritmin mukaan

$$\begin{aligned} T(k+1) &= T(k) + 1 + T(k) \\ &= 2^k - 1 + 1 + 2^k - 1 \\ &= 2^{k+1} - 1. \end{aligned}$$

Johtopäätös: $T(n) = 2^n - 1$ kaikilla $n \geq 0$.

Joskus – kuten seuraavassa esimerkissä – induktioperiaatetta käytetään muodossa, jossa vaihe (ii) on korvattu seuraavalla (miksi voimassa?).

(ii') Olkoon $k \geq n_0$. Kun oletetaan, että $P(n_0)$, $P(n_0+1)$, ... , P(k) ovat tosia, niin silloin voidaan todistaa, että myös P(k+1) on tosi.

Esimerkki. Osoitetaan, että seuraava välijärjestystä noudattava moduuli tulostaa lajitellun binääripuun solmut (nousevassa) suuruusjärjestyksessä.

```
MODULE tulostapuu(binääripuu p)
    IF p ei ole tyhjä puu THEN
        tulostapuu(p.vasen)
        tulosta(p.arvo)
        tulostapuu(p.oikea)
    ENDIF
ENDMODULE
```

Nyt induktiotodistuksessa tarvittavaa parametria n, jonka suhteen todistus suoritetaan, ei ole suoraan annettu. Tähän sopii usein tehtävän koko sopivalla tavalla mitattuna. Tehtävän kooksi sopii tässä tapauksessa hyvin puun solmujen lukumäärä. Osoitetaan algoritmin oikeellisuus induktiolla puun solmujen lukumäärän suhteen eli tarkasteltava väite P(n) on seuraava: algoritmi tulostapuu tulostaa lajitellun binääripuun solmut nousevassa järjestyksessä, kun puussa on n solmua kaikilla n:n arvoilla n=0, 1, 2, ...

Lähtökohta: Jos puun solmujen määrä on nolla, puu on tyhjä. Algoritmi toimii oikein (se ei tee mitään, koska puussa ei ole mitään tulostettavaa).

Induktio-oletus: Oletetaan, että algoritmi toimii oikein, kun puun koko on enintään k (siis käytetään induktio-oletuksen muotoa (ii)').

Induktioväite: Moduuli tulostapuu toimii oikein, kun lajitellussa binääripuussa p on k+1 solmua.

Induktioväitteen todistus: Olkoon lajitellussa binääripuussa p k+1 solmua. Silloin

1) puun p.vasen kaikkien solmujen arvot ovat pienempiä kuin p.arvo ja
2) puun p.oikea kaikkien solmujen arvot ovat suurempia kuin p.arvo.

Puussa p.vasen on enintään k solmua, joten induktio-oletuksen mukaan algoritmin rekursiivinen kutsu tulostapuu(p.vasen) tulostaa p:n vasemman poikapuun solmut suuruusjärjestyksessä. Sitten algoritmi tulostaa puun juuren, mikä on aivan oikein kohtien 1 ja 2 nojalla. Lopuksi algoritmi tulostaa oikean poikapuun solmut induktio-oletuksen mukaan oikeassa järjestyksessä. Siten koko puu tulostetaan suuruusjärjestyksessä.

Johtopäätös: Induktioperiaatteen mukaisesti päätellään, että algoritmi tulostaa jokaisen lajitellun binääripuun alkiot suuruusjärjestyksessä.

Huomaa, että induktio-oletus lausuttiin muodossa "kaikille *enintään* k:n suuruisille puille". Tämä on välttämätöntä, koska kumpikaan (k+1)-solmuisen puun poikapuista ei välttämättä ole tarkalleen k-solmuinen.

Näin on todistettu tulostapuu-moduulin osittainen oikeellisuus. Totaalisen oikeellisuuden osoittamiseksi täytyy vielä osoittaa, että algoritmi terminoituu välttämättä. Mutta tämä on ilmeistä, koska jokainen rekursiivinen kutsu pienentää puuta, eikä tyhjä puu aiheuta enää lisäkutsuja.

Tulostapuu-moduulin tapauksessa induktio sujui oivallisesti puun solmujen lukumäärän suhteen. Toisinaan puita koskevat todistukset on kuitenkin helpompi suorittaa puun solmujen lukumäärän sijasta puun korkeuden suhteen.

Edellisen esimerkin algoritmin toiminta oli ilmeistä, eikä sen oikeellisuudesta ole vaikea vakuuttautua pelkästään algoritmia tutkimallakaan. Kokenut ohjelmoija näkee tulostapuu-algoritmin oikeellisuuden suoraan vertaamalla sitä järjestetyn binääripuun määritelmään. Mutta induktiota hyväksi käyttäen voidaan todistaa algoritmin oikeellisuus myös silloin, kun sen toiminta on hyvinkin monimutkaista, eikä oikeellisuus ole millään muotoa ilmeistä.

3.3.2.2 Invarianttien käyttö oikeaksitodistamisessa

Induktiolla voidaan suoraan osoittaa vain 'pienten' ja luonteeltaan matemaattisten algoritmien tai niiden osien oikeellisuus. Mutkikkaammissa algoritmeissa toiminta voidaan kuvata sopivilla väitteillä, joiden paikkansapitävyys voidaan osoittaa induktiolla. Tällaisten väitteiden käyttöön perustuvassa menetelmässä algoritmin toiminnan keskeinen idea esitetään väitteenä, jonka muuttumattomuus eli ***invarianssi*** ohjelman suorituksen aikana osoitetaan induktiolla. Tällaista muuttumatonta väitettä sanotaan ***invariantiksi*****.** Menetelmä sopii erityisesti luonteeltaan iteratiivisten ja rekursiivisten algoritmien tai niiden osien oikeellisuuden todistamiseen. Invariantti ilmaistaan ohjelman tilana – eli ohjelman muuttujien arvoina – suorituksen tiettynä hetkenä. Alkutila tarkoittaa tilaa (tilannetta), kun algoritmin suoritus aloitetaan ja lopputila vastaavasti tilaa algoritmin lopussa. Käytännössä tällä menetelmällä ei todisteta koko ohjelman oikeellisuutta, vaan rekursiivisten moduulien ja toisto-rakenteiden oikeellisuus.

Invariantin käyttöön perustuva menetelmä on seuraavanlainen:

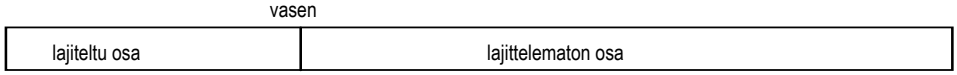
1. Muodostetaan väite, jolla on seuraavat ominaisuudet:
   a) väite pätee alkutilassa
   b) jos väite pätee lopputilassa, niin algoritmin suoritus on virheetön.
2. Osoitetaan, ettei algoritmin suoritus vaikuta väitteen totuusarvoon, ts. että väite on invariantti algoritmin suorituksen suhteen.
3. Osoitetaan, että lopputila saavutetaan välttämättä, ts. että algoritmi terminoituu.

Haettu väite tulee siis laatia siten, että väite on invariantti algoritmin suorituksen suhteen ja että se takaa algoritmin oikeellisuuden. Nimittäin vaatimuksen 2 mukaan väite pätee myös lopputilassa. Usein vaihe 2 todistetaan induktiivisesti eli jos tarkasteltava algoritmi sisältää toistorakenteen (tai rekursiivisen kutsun), niin todetaan, että väite on voimassa jokaisen silmukan suorituksen (rekursiivisen kutsun) jälkeen.

Esimerkki. Vaihtolajittelun (2.8.2.2) oikeellisuus. Kun vaihtolajittelualgoritmi kirjoitetaan oikeellisuustodistusta silmällä pitäen käyttäen havainnollisia muuttujan nimiä, käy menetelmän toimivuus ilmeiseksi. Näin oikeellisuuden todistamisen periaatteiden tuntemus myötävaikuttaa algoritmin laatuun jo sen laatimisvaiheessa.

```
MODULE vaihtolajittelu (vektori V)
        pituus := V.length
        FOR vasen := 1,...,pituus –1 DO
                FOR oikea := vasen+1,...,pituus DO
                        IF V[vasen] > V[oikea] THEN vaihda (V[vasen], V[oikea]) ENDIF
                ENDFOR
        ENDFOR
ENDMODULE
```

Vaihtolajittelu sisältää kaksi sisäkkäistä toistorakennetta. Liitetään uloimpaan (vasen) silmukkaan seuraavanlainen väite. Väitetään, että vektori V on lajiteltu indeksiin vasen saakka ja että sen oikealla puolella oleva osa on lajittelematon ja sisältää alkioita, jotka ovat kaikki suurempia tai yhtäsuuria kuin vasemmalla - jo lajitellulla - puolella olevat alkiot. Osoitetaan, että tämä väite on voimassa ulomman silmukan lopussa silmukkalaskurin vasen arvoilla 1, 2, ..., pituus–1. Olkoon tämä väite nimeltään P(vasen). Väitettä kutsutaan ns. silmukkainvariantiksi.



1. Alkutilassa vektorin alkioiden parittaisia vertailuja ei ole suoritettu lainkaan, joten väite on triviaalisti voimassa alkutilassa. Oletetaan, että väite pitää paikkaansa lopputilassa. Tällöin vasen=pituus-1, joten vektorin V alkiot V(1), V(2), ... , V(pituus–1) ovat järjestyksessä ja alkio V(pituus) on vähintään kaikkien muiden alkioiden suuruinen. Täten vektori V on järjestetty ja algoritmi toimii oikein.
2. Osoitetaan väitteen voimassaolo induktiolla. Kun vasen=1, niin vektorin ensimmäistä komponenttia verrataan vektorin komponentteihin 2, 3, ... ,pituus ja suoritetaan vaihto, jos löydetään väärä järjestys. Tämän jälkeen ensimmäisenä komponenttina on välttämättä pienin, joten P(1) pätee. Oletetaan nyt, että P(k) on tosi. Tällöin $V(k) \geq V(k-1) \geq ... \geq V(1)$ ja $V(j) \geq V(i)$ kaikilla j=k+1,k+2,...,pituus ja i=1,2,...,k. Kun nyt ulompi silmukka suoritetaan silmukkalaskurin vasen arvolla k+1, tulee algoritmin nojalla paikkaan V(k+1) pienin alkioista V(k+1),...,V(pituus). Induktio-oletuksen nojalla taas $V(k+1) \geq V(i)$, i=1, 2, ... ,k, joten $V(k+1) \geq V(k) \geq ... \geq V(1)$. Näin ollen vektorin k+1 ensimmäista alkiota ovat jo järjestyksessä. Täten myös P(k+1) pätee.
3. Kyseessä on definiitti toisto, joka päättyy tarkalleen n(n–1)/2 kierroksen (katso kappaleen 3.2.1 loppu) jälkeen, joten lopputila saavutetaan välttämättä.

3.3.2.3 Oikeaksitodistaminen peräkkäisten väitteiden avulla

Usein yksittäisen invarianssin löytäminen, tai sen todistaminen induktiolla, ei ole helppoa. Silloin voidaan käyttää peräkkäisiä väitteitä seuraavalla tavalla. Ensin muodostetaan alku- ja loppuehdot. Ohjelmaan sijoitetaan väitteitä ohjelman 'kriittisiin kohtiin' siten, että näiden avulla voidaan todistaa ohjelman toimivuus kulkemalla alkuehdosta väitteiden kautta loppuehtoon ohjelman kontrollin mukaisesti. Erityisesti jokaisen toistorakenteen ja rekursiivisen kutsun jälkeen tulee sijoittaa väite. Tarkoituksena on todistaa, että kun algoritmin jokaisesta kohdasta, jossa on väite, edetään ohjelman kontrollin mukaan kohtaan, jossa on seuraava väite, niin myös tämä väite pitää paikkansa. Jos väitteiden avulla saadaan silta alku- ja loppuehtojen välille, on osoitettu algoritmin oikeellisuus. Lopuksi tulee tietenkin vielä osoittaa, että ohjelma terminoituu. Yksittäisten väitteiden todistuksessa voidaan käyttää induktiota tai muita invariantteja tarpeen mukaan.

3.3.2.4 Terminoituvuus

Algoritmi voidaan todistaa päättyväksi tarkastelemalla esimerkiksi algoritmin toistorakenteita ja niihin liittyviä ehtoja, joissa oleva lauseke joskus tulee epätodeksi/todeksi, tai rekursiivisten kutsujen päättymistä. Algoritmin suorituksen päättyminen on helppo todeta, jos algoritmi sisältää ohjausrakenteinaan pelkästään peräkkäisyyttä ja valinnaisuutta, mutta vaikeampaa, jos algoritmi sisältää toistorakenteita tai jos algoritmi on rekursiivinen. Edellä on esitetty useita algoritmeja, joiden päättyminen on ilmeistä. Joskus algoritmin päättymistä voi olla hyvin vaikea osoittaa. Olemme jo aiemmin tarkastelleet esimerkkiä tällaisesta algoritmista Fermat'n suuren lauseen yhteydessä.

Esimerkki. Ackermannin funktio. Jotta algoritmin terminoituvuudesta voitaisiin vakuuttautua, täytyy algoritmin toiminta ymmärtää hyvin. Seuraavan esimerkkialgoritmin toimintaa on hyvin vaikea ymmärtää täydellisesti, mutta se on mahdollista ymmärtää riittävän hyvin, jotta voidaan osoittaa sen terminoituvan. Ns. Ackermannin funktio määritellään ei-negatiivisille kokonaisluvuille seuraavasti:

$$A(x,y)=\begin{cases} y+1, & x=0,\ y\geq 0 \\ A(x-1,1), & x>0,\ y=0 \\ A(x-1,A(x,y-1)), & x>0,\ y>0 \end{cases}$$

Tämä voidaan kirjoittaa suoraviivaisesti algoritmiksi:

```
MODULE Ack(x, y) RETURNS A(x, y)
        IF x = 0 THEN
                RETURN y + 1
        ELSE
                IF y = 0 THEN
                        RETURN Ack(x – 1, 1)
                ELSE
                        RETURN Ack(x – 1, Ack(x, y – 1))
                ENDIF
        ENDIF
ENDMODULE
```

Voidaan päätellä, että algoritmi päättyy ei-negatiivisilla kokonaislukuarvoilla x ja y, koska kahdessa rekursiivisessa kutsussa x pienenee yhdellä ja kolmannessa, x:n pysyessä muuttumattomana, y pienenee yhdellä. Viimeksi mainitussa rekursiivisessa kutsussa y saavuttaa joskus arvon nolla, jolloin alkaa uusi rekursio, jossa x pienenee yhdellä. Kaikki rekursiiviset kutsut johtavat siihen, että x pienenee aina kohti arvoa nolla, jolloin algoritmin suoritus päättyy.

Algoritmin terminoituminen on periaatteessa selvä. Käytännössä funktion arvon laskeminen kestää jo pienilläkin x:n ja y:n arvoilla hyvin kauan. Kiinnostunut lukija voi koettaa laskea esimerkiksi arvoja A(2,2) tai A(3,1). Myös funktion arvo kasvaa valtavan nopeasti, nopeammin kuin vaikkapa funktioiden $f(x,y) = x^y$ tai $g(x,y) = x^{x^{\cdots^x}}$, missä x:iä on y kpl. Jo varsin pienillä x:n ja y:n arvoilla funktion arvon esittämiseen tarvittavien numeroiden määrä ylittää, sanokaamme vaikkapa, maailmankaikkeuden atomien määrän 10^{86}. Yksistään jo tästä syystä algoritmi on käytännössä kelvoton, vaikka Ackermannin funktion arvon määrääminen annetuilla syötteillä onkin periaatteessa laskettavissa oleva tehtävä.

Esimerkki. Otetaan lopuksi esimerkki algoritmista, joka näyttää yksinkertaiselta, mutta jonka toimintaa kukaan ei täysin ymmärrä. Terminoituuko seuraava moduuli?

```
MODULE tricky(x)
        WHILE x > 1 DO
                IF x on parillinen THEN
                        x := x/2
                ELSE  (* x on pariton *)
                        x := 3*x + 1
                ENDIF
        ENDWHILE
ENDMODULE
```

Moduulin suorituksen jäljittäminen osoittaa, että algoritmi päättyy monilla x:n arvoilla, mutta päättyykö se kaikilla x:n arvoilla? Vastausta ei tiedetä.

4 Tietokoneen rakenne ja toiminta

Edellisissä luvuissa on keskeisenä tarkastelun kohteena ollut itse algoritmi: algoritmien suunnittelu ja esittäminen, algoritmien ominaisuudet sekä algoritmisen ongelmanratkaisun mahdollisuudet ja sen rajoitukset. Algoritmien valtava käytännön merkitys perustuu niiden automaattiseen suorittamiseen. Tietokone on yleissuoritin, joka pystyy suorittamaan hyvin erilaisia algoritmeja.

Lähes kaikki nykyiset tietokoneet perustuvat John von Neumannin 1940-luvulla muodostamiin käsitteisiin. Tätä ***von Neumannin konemallia*** karakterisoi muisti (joukko samankaltaisia muistipaikkoja) ja prosessori, jolla on käytettävissään joukko rekistereitä. Prosessori voi ladata tietoa muistista rekistereihin, suorittaa aritmeettisia ja loogisia operaatioita rekistereille sekä tallentaa rekisterien arvot takaisin muistiin. Koneen ohjelma koostuu käskyjoukosta, jolla em. operaatioita tehdään, sekä kontrollikäskyistä, joilla määrätään seuraava käsky.

Tässä luvussa tarkastellaan, millä periaatteella tietokoneen prosessori toimii ja sitä, miten se pystyy lukemaan, 'ymmärtämään' ja suorittamaan algoritmien käskyjä.

4.1 Matemaattiset perusteet

Digitaalisten tietokoneiden toiminta perustuu ns. ***kahden olotilan periaatteelle***: luotettavasti ja kohtuullisin kustannuksin on mahdollista rakentaa kone, joka tunnistaa kaksi toisistaan poikkeavaa olotilaa. Useampien tilojen käyttö ei tuo juurikaan etuja, mutta mutkistaa koneen rakennetta kohtuuttomasti. Aikaisemmin käytettiin myös jonkin verran rakenteeltaan analogisia tietokoneita, jotka perustuvat jatkuvan signaalin käsittelyyn, mutta myös analogisten signaalien käsittelyn digitaaliset menetelmät ovat kehittyneet, ja nykyään käytännöllisesti katsoen kaikki tietokoneet käyttävät binäärijärjestelmää toimintansa pohjana. Sana ***bitti*** (bit) tulee sanoista BInary digiT, joka tarkoittaa kaksijärjestelmän numeroa. Bitti on pienin informaation yksikkö.

Tiedon ***koodauksella*** tarkoitetaan tiedon esittämistä jonkin järjestelmän mukaisina merkkeinä tai symboleina. Koska tieto on yleensä jo ennen koodaustakin esitetty jotenkin, on useimmiten kyseessä tiedon esitystavan muuttaminen. Alkuperäisen esitystavan palauttamista sanotaan ***dekoodaukseksi***. Koodauksen tekemiseksi on kaksi perustapaa: ***tiedoittainen*** ja ***merkeittäinen*** koodaus. Tiedoittaisessa koodauksessa tiedon alkuperäinen merkitys muunnetaan kokonaisuutena uuteen esitysmuotoon, kun taas merkeittäisessä koodauksessa tiedon alkuperäisen esitystavan merkit muutetaan yksitellen uuteen järjestelmään. Tiedoittaista koodausta käytetään esimerkiksi liikennemerkeissä, henkilötunnuksessa tai vaikkapa vakioveikkauksessa. Merkeittäistä koodausta puolestaan edustavat esimerkiksi morseaakkoset.

Tietokoneessa esitettävä tieto voidaan jakaa merkkitietoon (aakkosellinen ja aakkosnumeerinen tieto) ja lukuihin (numeerinen tieto). Näiden erona on se, että merkkitiedossa tiedon merkitystä ei voi nähdä tiedon esityksestä, kun taas luvun arvo määräytyy suoraan luvun esityksestä, kun käytetty lukujärjestelmä tunnetaan. Niinpä aakkosellisen tiedon esittämiseksi tietokoneessa ei ole muuta mahdollisuutta muin merkeittäinen koodaus biteiksi: jokaiselle merkille varataan oma yksikäsitteinen bittiyhdistelmänsä.

Muunnokseen käytetään useita järjestelmiä, joista tunnetuimpia ovat 7- tai 8-bittinen ASCII (American Standard Code for Information Interchange) ja Unicode. Unicode on ohjelmistotalojen kehittämä laaja merkistöstandardi, joka kattaa suurimman osan maailman kirjoitettujen kielten käyttämistä merkeistä. Unicode määrittelee yksilöivän koodiarvon yli 90 000 erilaiselle kirjoitusmerkille. Sen avulla voidaan esittää lähes kaikki maailman kielten käyttämät merkit. Mikrotietokoneiden lisääntymisen myötä eräänlaisen standardin asemaan pääsi ASCII-koodi, joskin monet nykypäivän ohjelmat käyttävät tai ainakin tukevat Unicode-koodausta. ASCII-koodin voi tyhjentävästi esittää yksinkertaisena muunnostaulukkona, siinä ei ole mitään mielenkiintoista.

Aakkosista muodostuvan aakkosellisen tiedon lisäksi myös numeerinen, siis numeromerkeistä koostuva tieto, voidaan esittää merkkikoodina. Esimerkiksi osoitteissa, nimissä, puhelinnumeroissa, henkilötunnuksissa ja erilaisissa asiakasnumeroissa esiintyvät numerot eivät yleensä tarkoita mitään lukua, vaan ne on tarkoituksenmukaisinta tulkita pelkästään numerosarjoiksi vailla arvoa. Tällöin on kyseessä aakkosnumeerinen tieto. Lukuja esittävien numeroiden esittämistä tarkastellaan lähemmin tuota pikaa.

Yleisesti on voimassa, että n-bittisellä koodisanalla voidaan esittää 2^n erilaista koodia. Aikaisemmin käytettiin paljon 7-bittistä ASCII-koodia – riittäähän 128 koodisanaa isojen ja pienten kirjainten, numeroiden ja välimerkkien esittämiseen. Mutta kansainvälisissä yhteyksissä koodi on käynyt pieneksi, koska kaikkien eri maissa esiintyvien erikoismerkkien (kuten å, ä ja ö!) esittäminen ei ole mahdollista. 256 koodisanan 8-bittisellä ASCII-koodilla tämä onnistuu, ja lisäksi on mahdollista esittää erilaisia ei-kirjoittuvia merkkejä, joita käytetään tietokoneen ja sen oheislaitteiden ohjaukseen.

4.1.1 Lukujärjestelmät

Luku (number) on matemaattinen objekti, jolla on arvo. ***Numerolla*** (digit) puolestaan tarkoitetaan mm. lukujen esittämiseen käytettävää merkkiä. Silloin kun numerot esittävät lukuja, on myös niiden tiedoittainen koodaus mahdollista. Jos luvuilla aiotaan laskea, on lukujen arvon koodaaminen paljon käytännöllisempää kuin lukujen numeroiden koodaaminen. Tietokoneessa luvut esitetään kaksikantaisen lukujärjestelmän lukuina. Tarkastellaan lukujärjestelmiä hieman tarkemmin.

Lukujärjestelmää (number system), jonka kantaluku on k, sanotaan k-järjestelmäksi eli k-ariseksi järjestelmäksi. k-järjestelmässä luvun esittämiseen on käytettävissä k numerosymbolia d_0, d_1, ..., d_{k-1}. Yleisesti käytetyissä positionaalisissa järjestelmissä luvun esityksessä olevan numeron merkitys riippuu paitsi numerosymbolista itsestään, myös sen sijainnista muihin numeroihin nähden.

Määritelmä. k-järjestelmän esitys

$$a_{n-1}a_{n-2}...a_1a_0.a_{-1}a_{-2}...a_{-(m-1)}a_{-m}$$

missä kukin a_i (i = n-1,...,1,0,-1,...-m) on jokin d_j (j = 0,1,...,k-1) tarkoittaa lukua

$$a_{n-1}k^{n-1} + a_{n-2}k^{n-2} + ... + a_1k + a_0 + a_{-1}k^{-1} + a_{-2}k^{-2} + ... + a_{-(m-1)}k^{-(m-1)} + a_{-m}k^{-m}$$

Jos käytettävä lukujärjestelmä ei käy muuten selväksi, kantaluku voidaan merkitä luvun esityksen perään alaindeksiksi: $(a_{n-1}a_{n-2}...a_1a_0.a_{-1}a_{-2}...a_{-(m-1)}a_{-m})_k$.

Usein käytettyjä lukujärjestelmiä ovat:

- 2-järjestelmä eli binäärijärjestelmä
  kantaluku k = 2
  numerosymbolit: 0, 1
- 8-järjestelmä eli oktaalijärjestelmä
  kantaluku k = 8
  numerosymbolit: 0, 1, 2, 3, 4, 5, 6, 7
- 10-järjestelmä eli desimaalijärjestelmä
  kantaluku k = 10
  numerosymbolit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- 16-järjestemä eli heksadesimaalijärjestelmä
  kantaluku k = 16
  numerosymbolit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Muunnoskaava. Ottamalla kantaluku k yhteiseksi tekijäksi saadaan luvun esityksen arvo laskettua myös seuraavan kaavan avulla. Kaavaa

$$(a_{n-1}a_{n-2}...a_1a_0.a_{-1}a_{-2}...a_{-(m-1)}a_{-m})_k$$

$$= (...(a_{n-1}k + a_{n-2})k + ... + a_1)k + a_0 + (...(a_{-m}k^{-1} + a_{-(m-1)})k^{-1} + ... + a_1)k^{-1}$$

voidaan soveltaa kumpaankin suuntaan, joten sen avulla saadaan luvun esitys muunnettua järjestelmästä toiseen peräkkäisten jako- tai kertolaskujen avulla. Koska laskeminen muussa kuin kymmenjärjestelmässä on kymmenjärjestelmään tottuneelle vaikeata, muunnokset on helpointa tehdä tarvittaessa kymmenjärjestelmän kautta.

Esimerkki.

$11010_2 = 1\cdot2^4 + 1\cdot2^3 + 0\cdot2^2 + 1\cdot2 + 0 = (((\underline{1}\cdot2 + \underline{1})\cdot2 + \underline{0})\cdot2 + \underline{1})\cdot2 + \underline{0} = 26_{10}$

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

$E1A_{16} = E\cdot16^2 + 1\cdot16 + A = (\underline{14}\cdot16 + \underline{1})\cdot16 + \underline{10} = 3610_{10}$

$3610_{10} = 451\cdot8 + 2 = (56\cdot8 + 3)\cdot8 + 2 = ((\underline{7}\cdot8 + \underline{0})\cdot8 + \underline{3})\cdot8 + \underline{2} = 7032_8$

$374_{11} = 3\cdot11^2 + 7\cdot11 + 4 = (\underline{3}\cdot11 + \underline{7})\cdot11 + \underline{4} = 444_{10}$
$= 111\cdot4 + 0 = (27\cdot4 + 3)\cdot4 + 0 = ((6\cdot4 + 3)\cdot4 + 3)\cdot4 + 0 = (((\underline{1}\cdot4 + \underline{2})\cdot4 + \underline{3})\cdot4 + \underline{3})\cdot4 + \underline{0}$
$= 12330_4$

Desimaaliluvun muunnos 10-järjestelmästä k-järjestelmään voidaan suorittaa seuraavaa menetelmää käyttäen: Olkoon luvun kokonaisosa $x = (a_{n-1}a_{n-2}...a_1a_0)_{10}$ ja desimaaliosa $y = (0.a_{-1}a_{-2}...a_{-(m-1)}a_{-m})_{10}$, missä siis $0 \le y < 1$. Jaetaan x toistuvasti k:lla (ottamalla osamäärän kokonaisosa aina uudeksi jaettavaksi), kunnes jaettava tulee nollaksi. Peräkkäiset jakojäännökset muodostavat k-järjestelmän luvun kokonaisosan numerot oikealta vasemmalle. Desimaaliosan muunnos tehdään siten, että y *kerrotaan* toistuvasti k:lla (ottamalla tulon desimaaliosa aina uudeksi kerrottavaksi). Peräkkäisten tulojen kokonaisosat muodostavat k-järjestelmän 'desimaaliosan' numerot *vasemmalta oikealle*. On huomattava, että vaikka desimaaliluku olisi 10-järjestelmässä päättyvä, se saattaa olla k-järjestelmässä päättymätön (ja päinvastoin), joten kertolasku kannattaa lopettaa, kun riittävän tarkka likiarvo on saavutettu. Tulos on tarkka, jos desimaaliosa jossain kertolaskussa tulee nollaksi.

Esimerkki. Muunnetaan luku 123.703125_{10} 4-järjestelmään.

Kokonaisosa:	$\lfloor 123 / 4 \rfloor = 30$, jakojäännös = **3**
	$\lfloor 30 / 4 \rfloor = 7$, jakojäännös = **2**
	$\lfloor 7 / 4 \rfloor = 1$, jakojäännös = **3**
	$\lfloor 1 / 4 \rfloor = 0$, jakojäännös = **1**
Desimaaliosa:	$0.703125 \cdot 4 = \mathbf{2}.8125$
	$0.8125 \cdot 4 = \mathbf{3}.25$
	$0.25 \cdot 4 = \mathbf{1}.00$
Lopputulos on siis	1323.231_4.

Yleinen käsitys on, että kantaluvun kymmenen valinta on aikoinaan tapahtunut ihmisen sormien lukumäärän perusteella. Yhtä hyvin siis saattaisimme nykyään käyttää kymmenjärjestelmän sijasta viisikantaista 'kämmenjärjestelmää'[9]. Muinaiset sumerilaiset tosin käyttivät 12- ja 60-kantaisia järjestelmiä, jotka vieläkin sinnittelevät hengissä ajanlaskun yhteydessä. Siinä missä matemaatikon mielestä desimaalijärjestelmän asemesta olisi kannattanut valita järjestelmä, jonka kantaluku on alkuluku, tietojenkäsittelijä sadattelee sitä, että kymmenen ei ole kakkosen potenssi. Binäärilukujen yhteydessä käytetään desimaalijärjestelmän rinnalla tai sijasta usein oktaali- tai heksadesimaalijärjestelmää, koska näiden kantaluvut ovat kakkosen potensseja. Muunnos k-järjestelmän ja k^j-järjestelmän välillä on tavallista helpompi, koska se voidaan tehdä j numeron lohkoissa.

Esimerkki.

$1011010_2 = 1 \mid 011 \mid 010 = 132_8$

$E1A_{16} = 1110 \mid 0001 \mid 1010 = 111000011010_2$

$20120_3 = 2 \mid 01 \mid 20 = 216_9$

4.1.2 Kokonaislukujen esittäminen

Positiivisen luvun esitystä kaksijärjestelmässä edellä esitettyyn tapaan sanotaan luvun ***puhtaaksi binääriesitykseksi***. Kokonaislukujen esittämiseksi puhdas binääriesitys ei riitä: luvun absoluuttisen suuruuden lisäksi pitää myös sen etumerkki esittää jotenkin. Yksinkertaisin tapa on lisätä luvun esitykseen bitti, joka kertoo etumerkin. Luvun esityksen vasemmanpuoleisin eli eniten merkitsevä (most significant) bitti on nolla, jos luku on positiivinen, ja ykkönen, jos luku on negatiivinen. Koska yleisesti on voimassa, että n bitillä voidaan esittää (enintään) 2^n binäärilukua, voidaan n bitillä siis esittää kokonaisluvut -2^{n-1}, ..., 0, ..., $2^{n-1} - 1$.

[9] Koska 5 on alkuluku, niin p-adisten lukujen teorian perusteella tiedämme, että 5-kantainen matematiikka toimisi 'kauniimmin' kuin nykyinen 10-kantainen.

Esimerkki. Kokonaislukujen esittäminen, kun n=4. Lukualue on siis 0...15 tai –8...+7.

Luku	Etumerkki + itseisarvo	1:n komplementti	2:n komplementti	Excess-8
+7	0 111	0 111	0 111	1111
+6	0 110	0 110	0 110	1110
+5	0 101	0 101	0 101	1101
+4	0 100	0 100	0 100	1100
+3	0 011	0 011	0 011	1011
+2	0 010	0 010	0 010	1010
+1	0 001	0 001	0 001	1001
+0	0 000	0 000	0 000	1000
-0	1 000	1 111	-	-
-1	1 001	1 110	1 111	0111
-2	1 010	1 101	1 110	0110
-3	1 011	1 100	1 101	0101
-4	1 100	1 011	1 100	0100
-5	1 101	1 010	1 011	0011
-6	1 110	1 001	1 010	0010
-7	1 111	1 000	1 001	0001
-8	-	-	1 000	0000

Etumerkki+itseisarvo-esityksessä kokonaisluvun esitys saadaan lisäämällä merkkibitti luvun itseisarvon puhtaan binääriesityksen eteen. Taulukosta nähdään, että menettelyssä on kaksi ongelmaa: ensinnäkin positiivisten ja negatiivisten binäärilukujen muodostamat lukusuorat kulkevat eri suuntaan. Nimittäin jos lasketaan tavallisella kynällä ja paperilla allekkain -tavalla yhteen lukujen –2 ja +3 etumerkki+itseisarvo-esitykset, saadaan tulokseksi –5:

```
–2 =  1010
+3 =  0011
      1101   = –5
```

Ongelma on korjattu ns. ***yhden komplementtiesityksessä***, jossa negatiivisten lukujen itseisarvo-osat on komplementoitu, nollat on muutettu ykkösiksi ja ykköset nolliksi. Tämä tarkoittaa lukusuoran suunnan vaihtamista: esityksen arvo kasvaa nyt samaan suuntaan niin negatiivisilla kuin positiivisillakin luvuilla. Tarkastellaan äskeistä esimerkkilaskua uudelleen, nyt yhden komplementtiesitystä käyttäen:

```
–2 =  1101
+3 =  0011
      0000   = (+)0
```

Tulos ei ole vieläkään oikein, vaan se heittää yhdellä. Syynä on sama heikkous, joka on etumerkki+itseisarvo-esityksen toinen ongelma. Nimittäin nollalla on kaksi esitystä: plus nolla ja miinus nolla. Mutta tästä ongelmasta päästään helposti eroon siirtämällä lukujen esityksiä yhdellä pykälällä niin, että nollalle jää vain yksi esitys (+0). Samalla esitettävissä oleva lukualue laajenee yhdellä negatiiviseen suuntaan. Tällaista esitystapaa sanotaan ***kahden komplementtiesitykseksi***. Tarkastellaan esimerkkilaskua vielä kerran:

```
–2 =  1110
+3 =  0011
      0001   = +1
```

Nyt kokonaislukujen mekaaninen yhteenlasku sujuu oikein myös negatiivisilla luvuilla, ja edelleenkin esityksen ensimmäinen bitti kertoo luvun etumerkin. Etujensa ansiosta kahden komplementtiesitys onkin käytössä käytännöllisesti katsoen kaikissa tietoko-

neissa. Kahden komplementtiesitys voidaan lukusuoralla tulkita siten, että positiiviset luvut ovat oikealla paikallaan, mutta negatiiviset luvut on siirretty niiden oikealle puolelle lisäämällä niihin 2^n (yllä 16). Niinpä kahden komplementti voidaankin formaalisesti määritellä seuraavalla tavalla:

Määritelmä. Ei-negatiivisen kokonaisluvun $m \leq 2^{n-1}-1$ n-bittinen kahden komplementtiesitys on sama kuin m:n puhdas binääriesitys. Negatiivisen kokonaisluvun $-m \geq -2^{n-1}$ n-bittinen kahden komplementtiesitys on (positiivisen) kokonaisluvun $-m+2^n$ puhdas binääriesitys.

Esimerkki. Luvun 2 nelibittinen kahden komplementtiesitys on 0010, ja luvun –2 esitys on $-2+2^4 = 14_{10} = 1110_2$. Vastaavasti 8-bittiset esitykset ovat 00000010 ja $-2+2^8 = 254_{10} = 11111110_2$.

Käytännössä negatiivisten lukujen kahden komplementtiesitykset saadaan helpommin itseisarvoltaan vastaavien positiivisten lukujen esityksistä vaihtamalla nollat ykkösiksi ja päinvastoin sekä lisäämällä tulokseen luku 1. Tämä voidaan osoittaa seuraavasti. Olkoon positiivisen luvun m kahden komplementtiesitys $(m_{n-1}m_{n-2}...m_0)_2$. Silloin

$$m = \sum_{i=0}^{n-1} m_i \, 2^i$$

ja

$$\sum_{i=0}^{n-1} \left(1-m_i\right) 2^i + 1 = \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} m_i \, 2^i + 1 = \left(2^n - 1\right) - m + 1 = 2^n - m$$

Määritelmän perusteella nähdään, että muunnossääntö tuotti oikean tuloksen. Laskutoimitukset sujuvat siis kahden komplementtiesitystä käytettäessä oikein myös negatiivisilla luvuilla. Tietenkin esitettävissä oleva lukualue asettaa rajoituksensa: jos laskutoimituksen tulos ei pysy tämän alueen sisällä, ei tulos kahden komplementtiesityksessäkään voi olla oikein, vaan syntyy ns. ***ylivuoto***. Esimerkiksi:

```
+4 =  0100
+5 =  0101
      1001  = –7
```

Jos n = 16, voidaan esittää kokonaisluvut –32 768 ... +32 767. Jos n = 32, voidaan esittää kokonaisluvut –2 147 483 648 ... +2 147 483 647. Mutta kahden komplementtiesityksessä tulos ei ylivuodonkaan sattuessa ole mitä sattuu. Tulos on aina oikein modulo 2^n, missä n on esityksen pituus. Lukujen *x* ja *y* sanotaan olevan kongruentteja modulo m, merkitään $x \cong y \ (mod\ m)$, jos x voidaan kirjoittaa muodossa $x = y + km$, missä *k* on kokonaisluku. Esimerkiksi $-7 \cong 9$ (mod 16). Kahden komplementtiesityksessä lukusuora onkin siis lukusykli: Laskutoimitukset tapahtuvat esityksen pituuden n määräämän lukualueen puitteissa. Toisistaan $k*2^n$ verran eroavat luvut ovat siis samassa kohden rengasta, mutta eri 'kierroksella'.

Komplementtiesityksen lisäksi on vielä eräs yleisesti käytetty kokonaislukujen esitystapa, jota kutsutaan ***excess-koodaukseksi***. Sen idea on hyvin yksinkertainen: siirretään lukualue –a...b kokonaan positiiviselle puolelle lisäämällä *kaikkiin* lukuihin a. Jos esimerkiksi n=4 ja lukualue –8...7, puhutaan excess-8-koodauksesta, koska kaikkiin lukuihin lisätään $2^{4-1} = 8$. Nytkin ensimmäinen bitti edustaa etumerkkiä, mutta toisin kuin edellä, 0 tarkoittaa negatiivista ja 1 positiivista lukua. Vaikka aritmetiikka ei sujukaan yhtä kätevästi kuin kahden komplementtiesityksessä, on excess-koodauksella

eräitä teknisiä etuja, mistä syystä sitä käytetään mm. liukulukujen esityksessä. Eräänä etuna voidaan mainita se, että koodaus säilyttää lukujen suuruusjärjestyksen (puhtaina binäärilukuina), toisin kuin kahden komplementti.

4.1.3 Liukulukujen esittäminen

On selvää, että tietokoneessa ei voida esittää mielivaltaisia reaalilukuja, koska niissä voi olla binääripisteen jälkeen ääretön määrä bittejä. Jos bittimäärä on äärellinen, on lukujen esittämiseen kaksi päävaihtoehtoa:

1. ***Kiinteän pisteen esitys***: Käytetään aina samaa bittimäärää pisteen jälkeen, jolloin voidaan soveltaa kokonaislukujen koodaustapoja. Ongelmana on tarkkuuden häviäminen hyvin suurissa ja hyvin lähellä nollaa olevissa reaaliluvuissa.
2. ***Liukulukuesitys***: Siirretään ('liu'utetaan') binääripiste sopivaan kohtaan siten, että luvut voidaan aina esittää samalla määrällä merkitseviä bittejä.

Liukuluvut ovat näistä selvästi yleisemmin käytettyjä, joten seuraavaksi tarkastellaan niiden hivenen yksinkertaistettua esitystapaa. Merkitsevät bitit esitetään ns. mantissana (m), joka yleisen käytännön mukaan yhdenmukaistetaan siten, että piste siirretään luvun ensimmäisen ykkösen vasemmalle puolelle. Mantissa sijoittuu siis aina välille [0.5,1), ja sen koodauksessa riittää esittää binääripisteen oikealla puolella olevat bitit. Eksponentti (k) ilmoittaa kuinka paljon pistettä siirrettiin (vasemmalle: eksponentti positiivinen; oikealle: eksponentti negatiivinen). Alkuperäinen luku on siis $m \cdot 2^k$.

Esimerkki.

$10.11 = 0.1011 \cdot 2^2$, joten mantissa = 1011_2 ja eksponentti = 2_{10}
$0.0011 = 0.1100 \cdot 2^{-2}$, joten mantissa = 1100_2 ja eksponentti = -2_{10}

Käytännössä esiintyy suurta kirjavuutta mantissan ja eksponentin koodaustavoissa ja bittimäärissä. IEEE on määritellyt standardit erikseen normaaleille ja kaksoistarkkuuden liukuluvuille:

1. Lyhyet liukuluvut: 4 tavua = 32 bittiä:
   - merkkibitti
   - eksponentti: 8 bittiä, koodaus excess-128 (eksponentin arvoalue siis –128...127)
   - mantissan itseisarvo: 23 bittiä
2. Pitkät liukuluvut: 8 tavua = 64 bittiä:
   - merkkibitti
   - eksponentti: 11 bittiä, koodaus excess-1024 (eksponentin arvoalue –1024...1023)
   - mantissan itseisarvo: 52 bittiä

Liukulukuoperaatiot voidaan kirjoittaa algoritmeina, joissa käytetään kokonaisluku-aritmetiikkaa. Tehokkuussyistä nämä algoritmit toteutetaan usein laitteistollisesti.

4.1.4 Boolen algebra

Kaikki tietokoneen piirit sekä ohjelmoinnissa käytetyt loogiset lausekkeet voidaan esittää Boolen algebran lausekkeina, joilla voidaan laskea ja joita voi sieventää. Näin

ollen ’piirejä voidaan manipuloida matematiikan avulla’. Sen vuoksi seuraavaksi määritellään Boolen algebra ja sen laskulait.

Kahden alkion, yhteen- ja kertolaskun sekä ns. komplementtioperaation muodostamaa matemaattista struktuuria $B = \langle \{0,1\}, +, \times, \bar{} \rangle$ sanotaan Boolen algebraksi. Sen avulla mallinnetaan laskutoimituksia kaksijärjestelmässä. Boolen algebra eroaa tavallisesta lukujen algebrasta alkioiden rajoitetun lukumäärän suhteen, mikä aiheuttaa eroja myös operaatioiden käyttäytymisessä. Määritelmän mukaan Boolen algebran ainoat alkiot ovat 0 ja 1. Alkioiden välille on määritelty binääriset operaatiot, joita merkitään merkeillä + ja ×. Lisäksi algebrassa on unaarinen (voidaan siis kohdentaa vain yhteen arvoon) operaatio komplementti, joka merkitään yläviivalla symbolin päällä. Operaatiota lasketaan vasemmalta oikealle kuitenkin niin, että operaation × presedenssi on korkeampi kuin operaation +. Esimerkiksi lausekkeessa x+x×y (eli x+xy kuten seuraavassa lyhyesti merkitään) lasketaan ensin xy. Huomaa, että operaatioilla + ja × ei ole mitään tekemistä yhteen- ja kertolaskun kanssa, ja ne käyttäytyvätkin eri tavalla: esim. x+xy=x, kuten alla olevasta käy ilmi.

Seuraavassa on esitetty Boolen algebran laskulait, joita innokas lukija voi verrata tavanomaisen algebran vastaaviin lakeihin. Jatkossa jätetään, kuten on tapana, operaattori x merkitsemättä, paitsi numerosymbolin vieressä.

Boolen algebran laskulait:

- $0+0 = 0,\ 0+1 = 1+0 = 1+1 = 1;\quad 0\times0 = 0\times1 = 1\times0 = 0,\ 1\times1 = 1;\quad \bar{0} = 1, \bar{1} = 0.$
- Assosiatiivisuus: $x+(y+z) = (x+y)+z,\quad x(yz) = (xy)z.$ — Vastaa tavallista algebraa.
- Kommutatiivisuus: $x+y = y+x,\quad xy = yx.$ — Vastaa tavallista algebraa.
- Distributiivisuus: $x(y+z) = xy + xz,\quad x+yz = (x+y)(x+z).$ — Distributiivisuus on voimassa molemmin päin, toisin kuin normaalialgebrassa.
- Nolla, ykkös ja vasta-alkiot: $x+0 = x,\ x+1 = 1;\quad x\times0 = 0,\ x\times1=x;\quad x + \bar{x} = 1,\ x\bar{x} = 0$
- Absorptiolait: $x+xy = x,\quad x(x+y) = x.$ — Poikkeavat tavallisesta algebrasta.
- de Morganin lait: $\overline{x + y} = \bar{x}\bar{y},\quad \overline{xy} = \bar{x} + \bar{y}.$ — Poikkeavat tavallisesta algebrasta.

Näiden tavallisten Boolen kerto- ja yhteenlaskuoperaatioiden lisäksi kaksijärjestelmässä toimittaessa usein käytetään ***modulo 2 -yhteenlaskua***, joka määritellään seuraavasti: $0\oplus0 = 1\oplus1 = 0,\ 0\oplus1 = 1\oplus0 = 1$. Modulo k -yhteenlaskuhan on juuri se operaatio, jota sovelletaan tavanomaisessa k-järjestelmän lukujen allekkain-yhteenlaskussa sarakkeittain. Esimerkiksi kymmenjärjestelmässä laskettaessa yhteen samasta sarakkeesta luvut viisi ja seitsemän merkitään viivan alle samaan sarakkeeseen luvun kaksitoista asemesta luku kaksi, koska $2 \cong 12 \pmod{10}$.

Kun 0 tulkitaan epätodeksi (false), 1 todeksi (true), × AND-operaatioksi, + OR-operaatioksi, yläviiva NOT-operaatioksi ja muuttujat x, y ja z edustavat jotain loogista lauseketta, niin kyseessä on loogisten lausekkeiden tarkastelua ja sieventämistä käyttäen Boolen algebran laskulakeja. Boolen algebran laskulaeista useimmat ovat itsestään selviä. Esimerkiksi 0+1 voidaan tulkita loogiseksi lausekkeeksi false OR true. Kahden OR-operaatiolla yhdistetyn lausekkeen arvo on true, jos toinen tai molemmat lausekkeista ovat arvoltaan true. Näin ollen 0+1=1, 0+0=0, 1+0=1 ja 1+1=1. De Morganin lait puolestaan sanovat, että NOT(X OR Y)=(NOT X) AND (NOT Y) ja NOT(X AND Y)=(NOT X) OR (NOT Y) olivatpa x ja y mitä tahansa totuusarvoisia lausekkeita. Nämäkin voidaan todeta oikeaksi käyttämällä ns. talonpoikaisjärkeä tai todistaa todeksi käyttäen totuustaulukkoa. Kaavoja voidaan käyttää esimerkiksi loogisten lausekkeiden sieventämiseen esimerkiksi ohjelmassa. Seuraavassa meidän on tarkoitus käyttää Boolen algebran laskulakeja loogisten piirien sieventämisessä, jossa neljä viimeistä laskulakia ovat keskeisessä asemassa.

Boolen kaava voidaan todistaa oikeaksi totuustaulukolla tai soveltamalla Boolen algebran laskulakeja. Totuustaulukon avulla todistaminen tarkoittaa sitä, että todetaan kaavan oikeellisuus kaikilla muuttujien arvojen (0 ja 1) kombinaatioilla.

4.2 Fysikaaliset perusteet

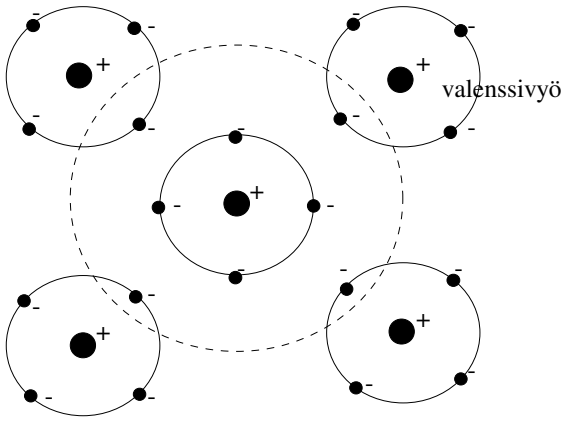
Ensimmäiset tietokoneet olivat täysin mekaanisia, myöhemmin elektromekaanisia, ja nykyiset koneet ovat täysin elektronisia. Tietokoneiden fysikaaliset perusteet ovatkin puolijohteissa ja digitaalielektroniikassa.

4.2.1 Puolijohteet

Atomeissa elektronit kiertävät ydintä erityisillä kiertoradoilla, elektronivöillä eli orbitaaleilla. Elektronin sijoittumisen orbitaaleille määräävät sen ja ytimen sähköisen vetovoiman muodostama potentiaalienergia sekä elektronin kiertoliikkeen liike-energia. Ns. ***valenssivyöllä*** sekä sitä lähempänä ydintä sijaitsevat elektronit ovat sidottuja ytimeen, kun taas ns. ***johtavuusvyöllä*** tai ulompana olevat elektronit voivat siirtyä atomista toiseen. Sidoksissa aineen valenssielektroneita voi nousta johtavuusvyölle ja siirtyä sieltä edelleen viereiseen atomiin. Tällöin aine ***johtaa sähköä***. Kyseisiä elektroneja sanotaan ***negatiivisiksi varauksenkuljettajiksi***. Tällaisia aineita ovat mm. useimmat metallit. Toisten aineiden sidoksissa taas valenssivyölle jää vajaa (alle kahdeksan elektronin) miehitys: syntyviä elektroniaukkoja, puuttuvia elektroneja, sanotaan ***positiivisiksi varauksenkuljettajiksi***. Aineet, joissa ei esiinny negatiivisia eikä positiivisia varauksenkuljettajia, siis vapaita elektroneja tai vapaita aukkoja, eivät johda sähköä. Niitä sanotaan ***eristeiksi***.

Puolijohteet ovat aineita, joilla puhtaina on yleensä hyvin niukasti niin positiivisia kuin negatiivisiakin varauksenkuljettajia. Ne siis johtavat hyvin huonosti tai eivät ollenkaan. Tilannetta muutetaan lisäämällä aineeseen sen kiderakenteeseen sopivia epäpuhtauksia. Erilaisilla yhdisteillä on erilaiset ominaisuudet. Yleisimmin käytetty puolijohde on ***pii*** (silicon), joka on maankuoren toiseksi yleisin alkuaine, tavallisen hiekan perusosa. Piillä on neljä valenssielektronia, jotka ovat ytimeen sidottuja. Kiderakenteessaan atomit lainaavat kultakin neljältä naapuriltaan yhden elektronin, joten kullakin pii-atomilla on

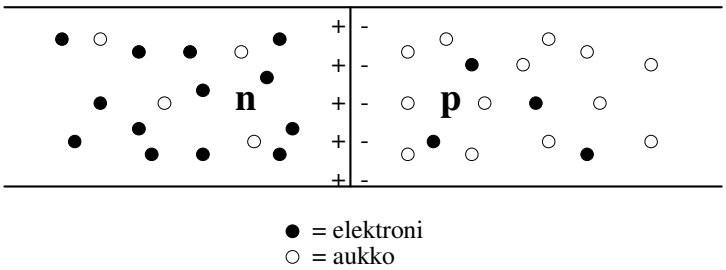
efektiivisesti kahdeksan valenssi-elektronia, joten vapaita elektroneja tai myöskään aukkoja ei ole. Pii ei siis johda sähköä.



Puolijohteet voidaan ***huumata*** (dope) vierailla kiderakenteeseen sopivilla atomeilla. Jos piihin lisätään antimonia, arseenia tai fosforia, joilla on viisi valenssielektronia, syntyy rakenteeseen vapaita elektroneja, jotka toimivat negatiivisina varauksenkuljettajina. Tällaista ainetta sanotaan ***n-aineeksi***. Jos taas piihin lisätään indiumia, booria tai galliumia, joilla on kolme valenssielektronia, syntyy rakenteeseen vapaita aukkoja, jotka toimivat positiivisina varauksenkuljettajina. Tällaista ainetta sanotaan ***p-aineeksi***. Huomaa, että niin n- kuin p-tyypinkin puolijohteet ovat kuitenkin kokonaisvarauksettomia.

4.2.2 Puolijohdekomponentit

Pelkkä n- tai p-aine ei sinänsä ole kovinkaan mielenkiintoinen. Mutta mitä tapahtuu, kun n- ja p-ainetta pannaan vierekkäin? Syntyy np-liitos:



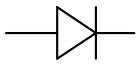
Diffuusion vuoksi vapaat elektronit pyrkivät n-aineesta, missä niitä on enemmän, p-aineeseen, missä niitä on vähemmän. Aukot pyrkivät vastaavasti p-aineesta n-aineeseen. Diffuusion seurauksena liitoksen rajapinta varautuu sähköisesti siten, että alun perin varaukseton n-aine muuttuu saamiensa aukkojen ansiosta positiiviseksi ja p-aine muuttuu vastaavasti negatiiviseksi. Varautuminen puolestaan aiheuttaa liitoksen yli vaikuttavan sähkökentän, joka pyrkii vastustamaan diffuusiota. Näin syntyy dynaaminen tasapainotila, jossa kokonaisvirta liitoksen yli on nolla.

Mitä tapahtuu, kun tasapainotilaa häiritään ulkoisella jännitteellä? Kytketään jännitelähde ensin ***myötäsuuntaan***: plusnapa p-aineeseen ja miinusnapa n-aineeseen. Silloin varaus purkautuu, sähkökenttä häviää ja diffuusio vahvistuu. Plusnapa vetää puoleensa

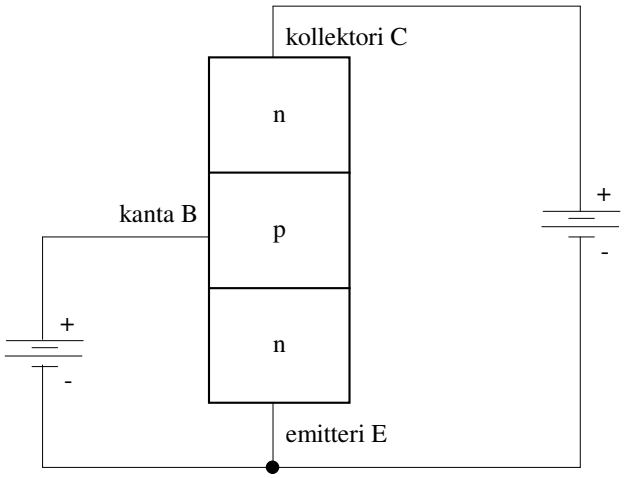
n-aineen ylimääräisiä elektroneja ja miinusnapa puolestaan imee p-aineen vapaat aukot. Vastaavasti plusnapa tuottaa p-aineeseen lisää aukkoja sieltä poistuneiden tilalle ja miinusnapa puolestaan tuottaa n-aineeseen lisää elektroneja sieltä poistuneiden tilalle. Diffuusio jatkuu niin kauan kuin ulkoinen jännite säilyy. Liitos siis johtaa.

Muutetaan kytkentä sitten ***vastasuuntaiseksi***: plusnapa n-aineeseen ja miinusnapa p-aineeseen. Silloin plusnapa vetää puoleensa n-aineen vapaat elektronit ja estää niitä tunkeutumasta diffuusion vaikutuksesta rajapinnan yli. Vastaavasti käy p-aineen vapaiden aukkojen. Liitos siis tyhjenee kokonaan niin positiivisista kuin negatiivisistakin varauksenkuljettajista, minkä seurauksena diffuusio estyy ja liitos eristää.

Np-liitos käyttäytyy siis erittäin mielenkiintoisesti: se johtaa sähköä toiseen suuntaan, mutta eristää toiseen suuntaan. Yksinkertaisesta np-liitoksesta muodostuvaa puolijohdekomponenttia sanotaan ***diodiksi***. Diodin piirrossymbolissa nuolen kärki osoittaa sähkövirran suunnan:



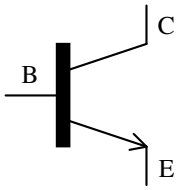
Kahdesta peräkkäisestä np-liitoksesta puolestaan muodostuu ***transistori***, jolla on kolme kytkentäkohtaa: kollektori (C), emitteri (E) ja kanta (B = base). Ns. bipolaarisen npn-transistorin rakenne on seuraavanlainen:



Transistorin toiminnan idea on, että kollektorin ja emitterin välisessä virtapiirissä kulkevaa virtaa voidaan säädellä kannan ja emitterin välisellä virralla. Tietyillä signaalitasoilla sopivasti rakennettu transistori toimii virtavahvistimena, mutta digitaalielektroniikassa ollaan enemmän kiinnostuneita transistorin *kytkentäominaisuuksista*: mitä tapahtuu, kun signaali muuttuu nopeasti korkeasta (merkitään sitä ykkösellä) matalaksi (merkitään nollalla) tai päinvastoin. Jos kannan ja emitterin välinen myötäsuuntainen jännite eli potentiaaliero on nolla (tai jos jännite on vastasuuntainen), niin kannan ja emitterin välinen np-liitos ei johda. Näin koko transistori ei johda, eikä kollektorin ja emitterin välisessä piirissä kulje virta, vaikka niiden välinen jännite olisi korkea. Mutta jos kannan ja emitterin välinen jännite on korkea (=1), niin kannan ja emitterin välinen np-liitos on myötäsuuntaan kytketty, jolloin se johtaa. Myös ylempi np-liitos johtaa, koska kantaan kytketty plusnapa estää diffuusiota vastustavan sähkökentän synnyn.

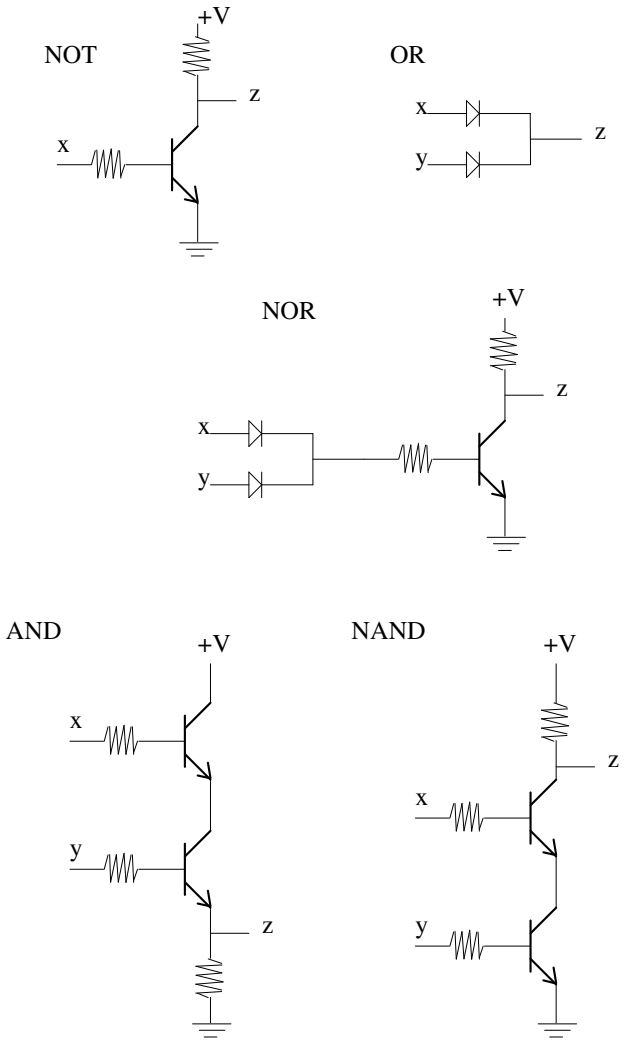
Transistorin toimintaa pyritään lisäksi edistämään erilaisilla rakenteellisilla ratkaisuilla. Esimerkiksi kanta tehdään käytännössä mahdollisimman ohueksi, jotta mahdollisimman suuri osa emitteriltä emittoituvista elektroneista päätyisi kollektorille. Näin kantavirta minimoituu. Aikaisemmin transistoria vastaava toiminto toteutettiin elektroniputkilla.

Nykyään transistoreita tehdään monilla tavoilla. Yleisiä ovat ns. kanavatransistorit (field effect transistor, FET), jotka jänniteohjattuina kuluttavat vähemmän tehoa kuin virta-ohjatut bipolaaritransistorit. Siksi ne kuumenevat vähemmän, minkä vuoksi ne voidaan pakata pienempään tilaan. Yhtä kaikki meidän tarpeisiimme riittää ymmärtää, että transistori on elektroninen kytkin, jossa kannalle tuodun signaalin avulla ohjataan kollektorin ja emitterin välistä yhteyttä. Transistorin piirrossymbolissa nuolen kärki osoittaa sähkövirran suunnan:



4.2.3 Loogiset piirit

Transistorien ja diodien avulla muodostetaan kytkentöjä, joita kutsutaan ***loogisiksi piireiksi*** (logical circuits). Nimitys tulee siitä, että jos korkea ja matala jännitetaso (potentiaali) tulkitaan totuusarvoina tosi ja epätosi, niin piirit kuvaavat päätöksentekoa, logiikkaa. Loogiset piirit muodostuvat komponenteista, joita sanotaan ***porteiksi*** tai ***veräjiksi*** (gates). Kukin portti toteuttaa jonkin totuusarvoisen funktion yhden tai useamman syöttösignaalin ja tulossignaalin välillä. Esimerkiksi jos kaksi transistoria kytketään sarjaan siten, että sähkövirta voi kulkea molempien transistorien läpi ainoastaan silloin, kun kummankin transistorin kanta on ylhäällä eli niihin johdetaan korkea jännite, saadaan konstruktio, jota nimitetään AND-portiksi: jotta tulos olisi ylhäällä, ensimmäisen syötön tulee olla ylhäällä JA toisen syötön tulee olla ylhäällä. Eräitä loogisia portteja skemaattisine toteutuksineen on esitetty seuraavassa:



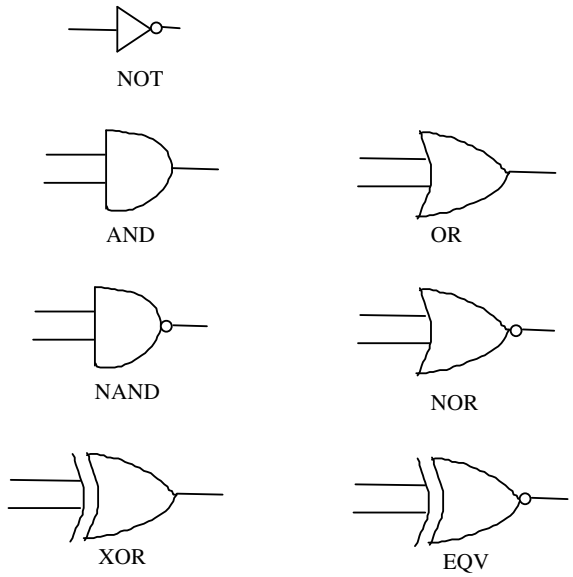
Todellisuudessa loogisten porttien toteutus on toki monimutkaisempi: niihin kuuluu mm. erilaisia puskuriasteita, joiden tarkoitus on saada portit toimimaan tarkoitetulla tavalla ympäristöstä riippumatta. Porttien toteutukseen voidaan myös käyttää erilaisia tekniikoita, logiikoita. Tässä yhteydessä tärkeintä on ymmärtää, että loogiset piirit ovat rakenteeltaan aidosti modulaarisia, ja käytännössä portteja voi yhdistellä toisiinsa vapaasti kuten Lego-palikoita.

Abstrahoimalla loogisten piirien toiminnan niiden rakenteellisesta toteutuksesta voimme siirtyä ylemmälle abstraktiotasolle ja unohtaa porttien toteutus transistoritekniikalla. Loogiset piirit voidaan nähdä Boolen lausekkeiden syntaktisena varianttina, vaihtoehtoisena esitysmuotona. Oleellista on, että lausekkeet ovat toteutettavissa puolijohdeelektroniikalla. Porttien toteuttamat totuusfunktiot voidaan määritellä ***totuustaulujen*** avulla. Merkitään totuusarvoa tosi ykkösellä ja totuusarvoa epätosi nollalla. Seuraavassa on esitetty loogisten funktioiden AND, OR, NAND (= Not AND), NOR (= Not OR), XOR (= eXclusive OR) ja EQV (= EQVivalence) totuustaulut ja niitä vastaavat Boolen lausekkeet:

x	NOT x $\overline{x}$
0	1
1	0

x	y	x AND y xy	x OR y $x+y$	x NAND y $\overline{xy}$	x NOR y $\overline{x+y}$	x XOR y $x \oplus y$	x EQV y $\overline{x \oplus y}$
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

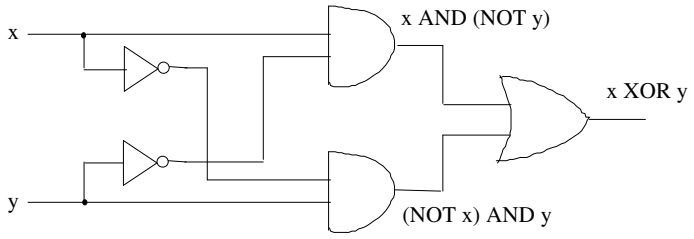
Loogisista veräjistä käytettävät piirrossymbolit ovat:



Loogisia piirejä piirrettäessä on huomattava, että signaalin haaroitus on sallittua: signaali kopioituu identtisenä jokaiseen solmukohdasta lähtevään haaraan. Sen sijaan usean signaalin yhdistämisessä tapahtuu aina jokin looginen operaatio, joka on merkittävä asianmukaisella symbolilla.

Loogisia perusportteja yhdistelemällä voidaan rakentaa loogisia piirejä moniin eri tarkoituksiin. Loogiset piirit esitetään Boolen lausekkeina, joita voidaan sieventää käyttämällä Boolen algebran laskulakeja. Sieventämisen tarkoitus on ilmeinen: matematiikkaa käyttäen voidaan tarvittavan ‘raudan’ määrää vähentää. Sieventämisestä esitetään seuraavassa esimerkkejä.

Esitettyä vähemmälläkin määrällä perusportteja tullaan toimeen. Esimerkiksi poissulkeva tai, XOR, voidaan esittää muiden porttien avulla seuraavasti:



Itse asiassa 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).

Monimutkaisissa piireissä voidaan viivojen yhtymiskohdat vahventaa, ja milloin kaksi viivaa risteävät yhtymättä, jätetään risteyskohta vahventamatta. Miten loogisia piirejä sitten muodostetaan? Piiri voidaan tietenkin muodostaa 'keksimällä'. On myös

systemaattisia tekniikoita, joita ei kuitenkaan tässä yhteydessä käydä lähemmin tarkastelemaan. Katsotaan sen sijaan esimerkkiä.

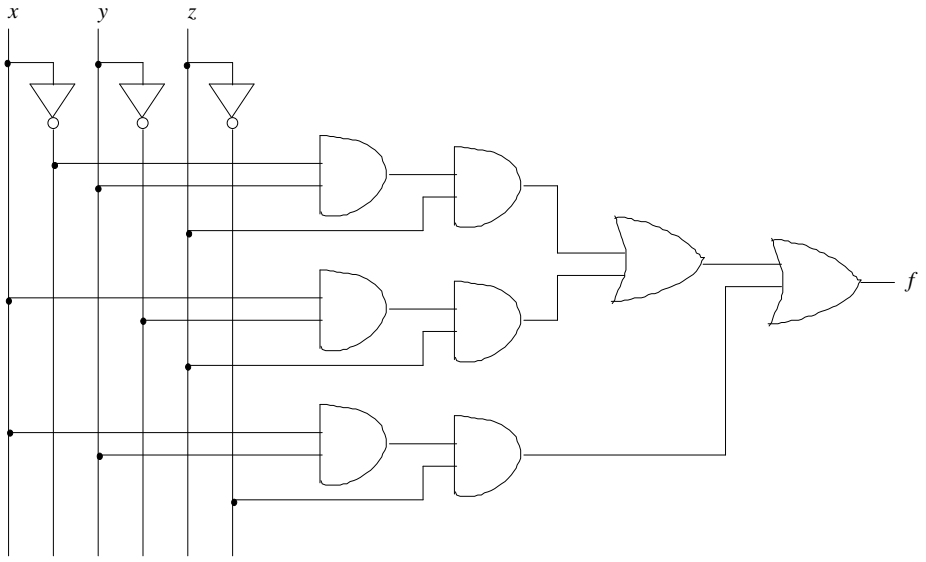
Esimerkki. Muodostettava looginen piiri, joka toteuttaa funktion *f* = *f(x,y,z)*, jonka totuustaulu on:

x	*y*	*z*	*f*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Nyt f:n lauseke voidaan lausua em. taulukon avulla valitsemalla ne x,y,f-kombinaatiot, joilla f=1. Jos syötteen arvo tässä on 1, niin syötteen symboli tulee lausekkeeseen sellaisenaan, ja jos arvo on 0, syöte komplementoidaan (miksi näin?). Lopuksi kunkin rivin lauseke yhdistetään muihin OR-operaattorilla. Saadaan:

(NOT *x* AND *y* AND *z*) OR (*x* AND NOT *y* AND *z*) OR (*x* AND *y* AND NOT *z*).

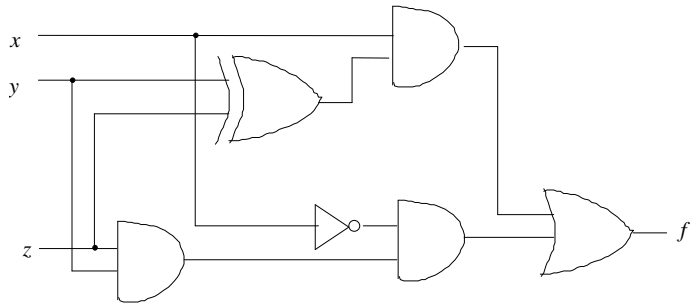
Siis $f(x,y,z) = \bar{x}yz + x\bar{y}z + xy\bar{z}$, ja voimme piirtää



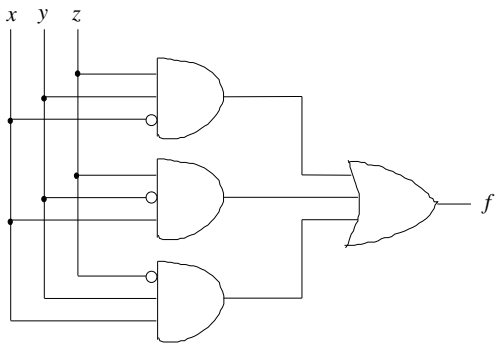
Toisaalta

$$f(x,y,z) = \bar{x}yz + x\bar{y}z + xy\bar{z} = \bar{x}yz + x(\bar{y}z + y\bar{z}) = \bar{x}yz + x(y \oplus z)\,,$$

jolloin



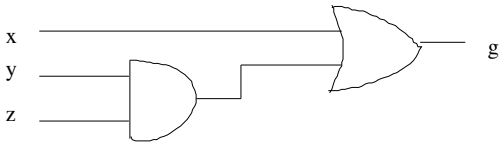
Assosiatiivisten operaatioiden osalta esitystä voidaan vielä yksinkertaistaa jättämällä turhat sulkeet pois ja sallimalla binääriselle portille useamman kuin kaksi syöttöä. Huomaa, että AND- ja OR-portille voidaan sallia useita syötteitä ja portin toiminta on ilmeinen. On myös tapana esittää käännetty syöttö erillisen NOT-veräjän asemesta pienellä ympyrällä vastaavassa syöttölinjassa. Tällaisin lyhennysmerkinnöin esimerkkipiirimme ensimmäinen versio voidaan yksinkertaistaa muotoon:



Esimerkki. Muodostettava looginen piiri, joka toteuttaa funktion $g = g(x,y,z)$, jonka totuustaulu on:

x	y	z	g
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Havaitaan, että g saa y:stä ja z:sta riippumatta arvon 1, kun $x = 1$, ja x:stä riippumatta arvon 1, kun $y = z = 1$. Voidaan siis 'keksiä', että $g(x, y, z) = x$ OR (y AND z) = $x + yz$, eli piirrettynä



Tässä tapauksessa sievä esitys oli helppo keksiä. Sama tulos toki saadaan systemaattisesti laskemallakin. Koska taulukossa on nollarivejä vähemmän kuin ykkösrivejä, kannattaa määrittää g:n komplementin lauseke ensin. Edellisen esimerkin tavoin nollariveiltä saadaan:

$$\overline{g} = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + \overline{x}\,y\,\overline{z}$$

jolloin

$$g = \overline{\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + \overline{x}\,y\,\overline{z}}$$

ja edelleen sieventämällä

$$= \overline{\overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + \overline{x}\,y\,\overline{z}}$$
$$= \overline{\overline{x}\,\overline{y}\,\overline{z}} \cdot \overline{\overline{x}\,\overline{y}\,z} \cdot \overline{\overline{x}\,y\,\overline{z}}$$
$$= (x + y + z)\,(x + y + \overline{z})\,(x + \overline{y} + z)$$
$$= x + (y + z)\,(y + \overline{z})\,(\overline{y} + z)$$
$$= x + (y + z\overline{z})\,(\overline{y} + z)$$
$$= x + y\,(\overline{y} + z)$$
$$= x + y\,\overline{y} + yz$$
$$= x + yz$$

4.3 Tietokoneen komponentteja

Tietokone on periaatteessa hyvin suuren määrän portteja sisältävä looginen piiri. Kuten kaikkien ihmisen tekemien kompleksisten systeemien, tietokoneen rakenne on erittäin

modulaarinen. Tietokone koostuu siis komponenteista. Tietokoneessa tarvitaan ainakin seuraavanlaisia loogisia komponentteja:

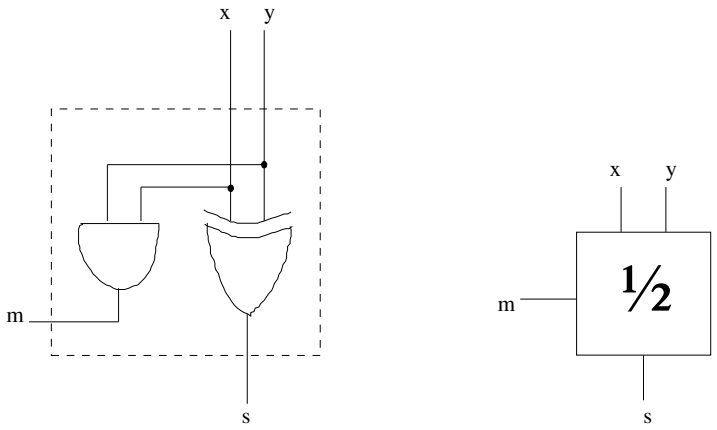
- ***Aritmetiikasta*** huolehtiva yksikkö. Monisteessa esitettävässä tietokoneessa toteutetaan laitteistollisesti kahden komplementtilukujen yhteen- ja vähennyslasku; kerto- ja jakolasku hoidetaan hyvin matalan tason ohjelmilla (firmware). Korkeamman tason laskenta toteutetaan konekielellä tai korkeamman tason ohjelmilla.
- ***Muisti*** on tietojen automaattisen käsittelyn välttämätön edellytys. Tässä kappaleessa tarkastellaan yhden bitin muistavaa laitetta, ns. kiikkua, sekä n-bittisen rekisterin konstruointia kiikkujen avulla.
- ***Väylät*** ovat edellytys tiedon siirrolle. Väylät voidaan jakaa loogisesti tieto-, ohjaus- ja osoiteväyliin. Tieto- ja osoiteväyliä tarvitaan tiedon siirtämiseen rekisterien ja keskusmuistin välillä. Ohjausväylän avulla toteutetaan koneen ohjelmoitavuus: osa käsittelyn kohteena olevasta tiedosta on käsittelyä ohjaavaa tietoa.
- ***Loogiset vertailut*** mahdollistavat algoritmeissa välttämättömien valintarakenteiden ja dynaamisten toistorakenteiden toteuttamisen hyvin matalalla tasolla. Tarkasteltavat primitiiviset operaatiot ovat kahden komplementtilukujen nolluuden ja negatiivisuuden havaitseminen.
- ***Kello*** tahdistaa eri komponenttien toiminnan. Kellotaajuus määrää koneen nopeuden. Kellotaajuutta nostamalla saadaan koneesta siis enemmän tehoa. Kellotaajuuden nostoa rajoittavat kuitenkin komponenttien fysikaaliset ominaisuudet.
- ***Syöttö ja tulostus***. Näyttö ja näppäimistö ovat loogisessa mielessä yksinkertaisia, mutta käytännössä tärkeitä komponentteja.

4.3.1 Yhteenlasku

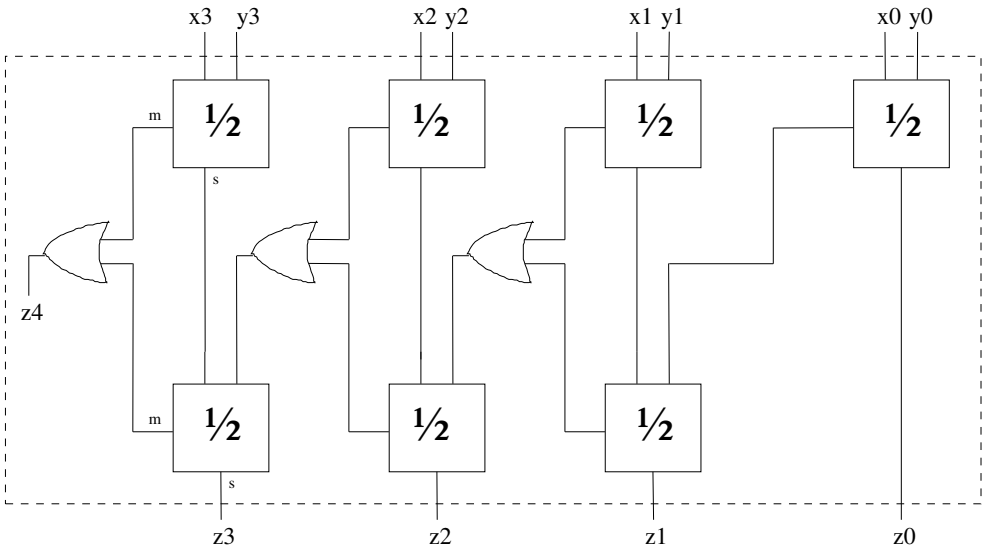
Kahden kahden komplementtiluvun yhteenlasku toteutetaan tavallisella 'kynällä ja paperilla allekkain' -algoritmilla. Kussakin sarakkeessa olevat numerot lasketaan yhteen, tulokseen lisätään mahdollinen muistinumero edellisestä sarakkeesta, ja tulos kirjoitetaan viivan alle. Samalla syntyy mahdollisesti muistinumero seuraavaan sarakkeeseen. Merkitään kahden bitin yhteenlaskun kaksibittisen tuloksen ensimmäistä bittiä *m*:llä (muistinumero) ja jälkimäistä bittiä *s*:llä (sarakesumma). Yhteenlaskua $x+y = ms$ kuvaa totuustaulu:

x	*y*	*m*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Taulukosta nähdään, että $m = xy$ ja $s = x \oplus y$. Tällainen laite on ns. ***puolisummain*** (half-adder), jonka symboli on seuraavassa kuvassa oikealla:

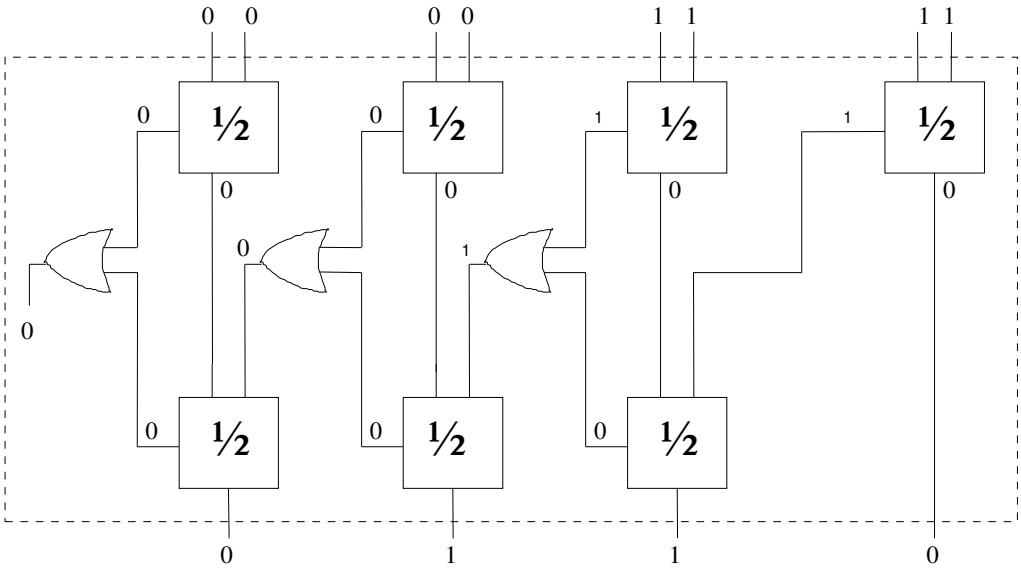


Pidempien lukujen yhteenlasku vaatii useita puolisummaimia. n-bittinen ***kokosummain*** (full adder) konstruoidaan modulaarisesti 2n–1:stä puolisummaimesta. Esimerkiksi 4-bittinen kokosummain:



Piiri laskee yhteenlaskun $x_3x_2x_1x_0 + y_3y_2y_1y_0 = z_4z_3z_2z_1z_0 \cong z_3z_2z_1z_0 \pmod{2^4}$. Nelibittisillä luvuilla laskettaessa ulostulo z_4 vuotaa yli. Kun tarkastellaan kahden komplementin lukuja, niin silloin 4 bitillä voidaan esittää luvut –8…+7 kuten edellä todettiin. Laskettaessa tällaisia lukuja yhteen, tulosta ei välttämättä voida esittää neljällä bitillä, jolloin tapahtuu ylivuoto ja tulos on väärä. Tässä on hyvä huomata, että tulos voi silti olla oikein, vaikka $z_4=1$. Yhteenlaskun tuloksen oikeellisuus voidaan todeta tarkastelemalla yhteenlaskettavien ja tuloksen ylimpiä bittejä (miten?).

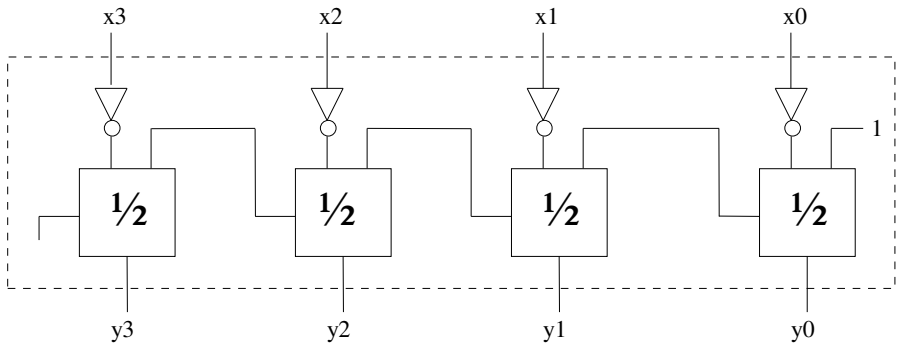
Havainnollistetaan 4-bittisen kokosummaimen toimintaa esimerkkisummalla 0011+0011, jossa ei tapahdu ylivuotoa ja summaksi tulee 0110 (eli $3_{10}+3_{10}=6_{10}$). Kirjoitetaan arvot 0 ja 1 piirin johtoihin:



Itse asiassa kokosummainta esittävä looginen piiri kuvaa kokonaislukujen yhteenlaskun 'kynällä ja paperilla allekkain' -algoritmin. Kun algoritmi on esitetty loogisena piirinä, se on helppo toteuttaa elektroniikalla. Algoritmissa kussakin sarakkeessa oikeanpuoleisinta lukuun ottamatta suoritetaan kaksi puoliyhteenlaskua: yhteenlaskettavien lukujen vastinbittien yhteenlasku ja muistinumeron lisäys. Huomaa muistinumerojen kulkeutuminen sarakkeesta toiseen. Yksinkertainen OR-portti riittää, koska (nollasta poikkeava) muistinumero voi syntyä vain korkeintaan toisessa kunkin sarakkeen puoliyhteenlaskuista: jos ylemmässä puoliyhteenlaskussa syntyy muistinumero m=1, on syntyvä tulos s=0, jolloin alemmassa puoliyhteenlaskussa toinen yhteenlaskettava on 0, joten syntyvä muistinumerokin on 0. Jotta alemmassa puoliyhteenlaskussa syntyisi muistinumero m=1, pitää yhteenlaskettavien olla 1 ja 1; tällöin ylemmässä puoliyhteenlaskussa syntynyt tulos on siis 1, jolloin syntynyt muistinumero on 0.

4.3.2 Vähennyslasku

Vähennyslasku voidaan toteuttaa kokosummaimen avulla, kunhan toinen yhteenlasket-tava muunnetaan ensin vastaluvukseen. Vastalukuoperaatio sujuu kahden komple-menttilukujen määritelmän mukaan muuttamalla bitit komplementeikseen ja lisäämällä tulokseen ykkönen:



Piiri toteuttaa laskun $y_3y_2y_1y_0 \cong -x_3x_2x_1x_0 \pmod{2^4}$. Huomaa, että lisätty ykkönen voi kulkeutua luvun eniten merkitsevään bittiin asti ja lopulta vuotaa yli. Piiri toimii myös tapauksessa x = 0.

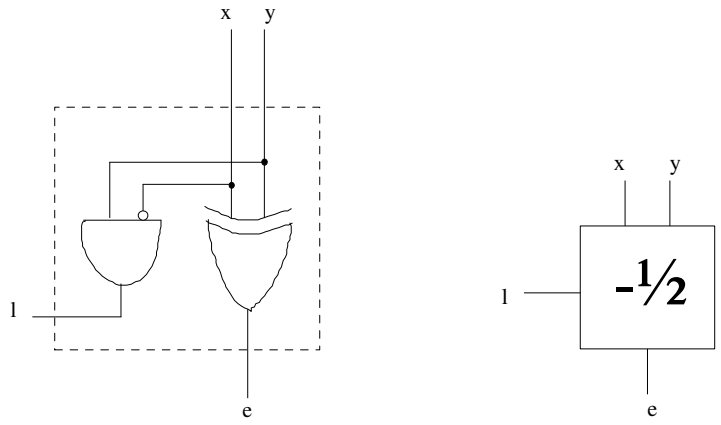
Vähennyslasku voidaan toteuttaa myös suoraan, samalla tavalla kuin yhteenlasku. Vähennyslasku suoritetaan tuttuun tapaan 'kynällä ja paperilla allekkain' -algoritmilla. Kun nollasta vähennetään ykkönen, täytyy lainata 10_2 seuraavasta sarakkeesta. Esimerkiksi:

$$\begin{array}{rr} & {\scriptstyle 10\,10} \\ & 1101 \\ - & 0111 \\ \hline & 0110 \end{array}$$

Konstruoidaan yhteenlaskun tavoin ensin puolivähennyslaskulaite eli ***puolivähennin***. Puolivähentimen toimintaa vähennyslaskussa $x–y$ kuvaa totuustaulu:

x	y	l	e
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

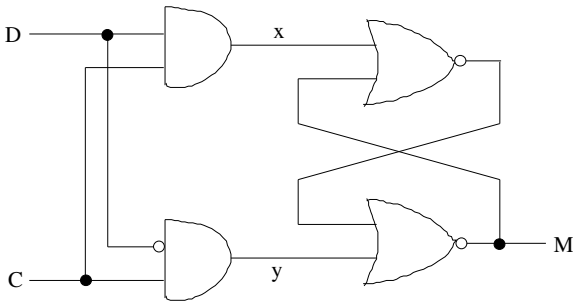
Taulukosta nähdään, että sarake-erotus $e = x \oplus y$ (samoin kuin yhteenlaskussakin!) ja lainaus $l = \bar{x}y$. Puolivähennin ja sen symboli ovat:



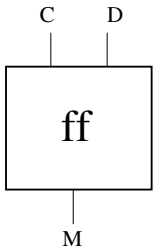
Kokovähennin n-bittisille luvuille konstruoidaan modulaarisesti puolivähentimistä, kuten yhteenlaskulaite. Tämä jätetään harjoitustehtäväksi. Huomaa lainauksen kulkeutuminen!

4.3.3 Kiikku

Muisti tarkoittaa sitä, että jokin tapahtuma riippuu paitsi syöttötiedoista myös aiemmin tapahtuneesta. Muistin toteuttamiseksi tulee siis konstruoida looginen piiri, jolla on historia: sen toiminta riippuu paitsi syötteistä myös siitä, mitä aiemmin on tapahtunut. Piirin historia kuvataan sen tilana, aiempien toimintojen lopputuloksena. Yksinkertaisin muistava piiri on yhden bitin muistava ***kiikku*** (flip-flop):



jota merkitään symbolilla



Kiikulla on kaksi syötettä, kontrollibitti C ja databitti D. Kiikun tilaa kuvaa tulosbitti M. Kiikkua käytetään seuraavasti: kun databitti D halutaan kirjoittaa muistiin M, asetetaan ohjausbitti C ykköseksi, jolloin M saa saman arvon kuin D. Kun D on kirjoitettu muistiin, ohjausbitti asetetaan nollaksi. Tällöin M ei muutu, vaikka D muuttuisikin.

Kiikku muodostuu kahdesta osasta: ohjausosasta (AND-portit) ja muistiosasta (ristiinkytketyt NOR-portit). Tarkastellaan ensin muistiosan toimintaa. Seuraavassa totuustaulussa M_t tarkoittaa M:n vanhaa arvoa ajanhetkellä t ja M_{t+1} M:n uutta arvoa ajanhetkellä t+1:

x	y	M_t	M_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1

Ohjausosa on suunniteltu niin, että jompi kumpi linjoista $x = CD$ ja $y = C\overline{D}$ on aina nolla. Yhdistelmä x=y=1 ei siten voi esiintyä. Havaitaan siis, että kun muistiosan ohjaus (x,y) = (0,0), niin ristiinkytkentä säilyttää entisen tilansa, oli se sitten 0 tai 1. Ohjauksella (x,y) = (1,0) asettuu ristiinkytkentä tilaan M=1 ja ohjauksella (x,y) = (0,1) tilaan M=0. Kiikku siis todella toimii halutulla tavalla: Kun C=0, muistiosan ohjaus (x,y) = (0,0) säilyttää ristiinkytkennän tilan ja piiri muistaa M:n. Kun C=1, muistiosan ohjaus (x,y) = $(D,\overline{D})$ pakottaa piirin tilaan M=D aiemmasta tilasta riippumatta.

Kiikun (ja muidenkin loogisten piirien) toiminnan tarkastelussa on hyvä huomata, varsinkin jos laite sisältää ristiinkytkentöjä, että piirin toiminnan tarkastelussa saattaa peräkkäinen johtoja seuraava ajattelutapa johtaa virheelliseen tulkintaan. Ideahan on se, että piirin tulee tietyillä syötteillä jäädä johonkin stabiiliin tilaan, jolloin porttien välisissä johdoissa on jännite (1) tai sitten ei (0). Esim. yllä olisi ristiinkytkennän takia periaatteessa mahdollista, että M jäisi heilahtelemaan tilojen 0 ja 1 välille, jolloin piirillä ei olisi mitään virkaa. Ohjausosa on kuitenkin rakennettu niin, että näin ei pääse käymään.

Edellä esitetty kiikun toiminta voidaan todeta myös lyhyemmin tarkastelematta yo. totuustaulua ja peräkkäisiä M:n tiloja:

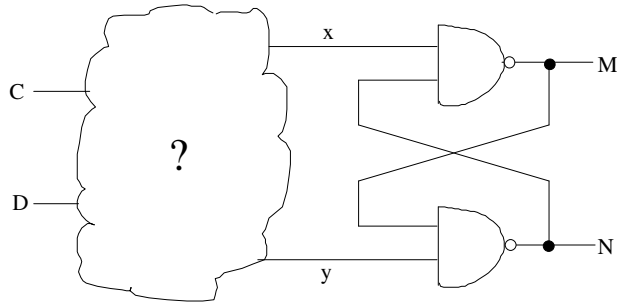
Tapaus 1: **Muista bitti M.** *Tällöin tulee asettaa C=0.* Silloin x=y=0 D:stä riippumatta. Tällöin muistiosan ylempään NOR-porttiin menee 0 ja M ja näin ollen ylemmän portin ulostulo on $\overline{M}$. Tällöin alempaan NOR-porttiin menee syötteenä $\overline{M}$ ja 0, joten ulos tulee M. Näin ollen M:n arvo ei muutu eli piiri muistaa M:n.

Tapaus 2: **Tallenna M=0**. *Tällöin tulee asettaa C=1 ja D=0.* Silloin x=0 ja y=1. Koska y=1, niin riippumatta alemman NOR-portin ylemmästä syötöstä, siitä tulee aina ulos 0. Näin ollen M saa arvon 0 riippumatta siitä mikä on M:n vanha arvo.

Tapaus 3: **Tallenna M=1.** *Tällöin tulee asettaa C=1 ja D=1.* Silloin x=1 ja y=0, jolloin alemman NOR-portin ylempi syöttö on aina 0 ja koska y on aina 0, niin alemman portin ulostulo on aina 1. Näin ollen M saa arvon 1, olipa M:n vanha arvo mikä tahansa.

Kiikku voidaan toteuttaa toisinkin. Oleellista on takaisinkytkentä, joka tuottaa muistiominaisuuden.

Esimerkki. Voiko kahden NAND-veräjän ristiinkytkentä toimia kiikkuna? Tarkastellaan hahmotelmaa:



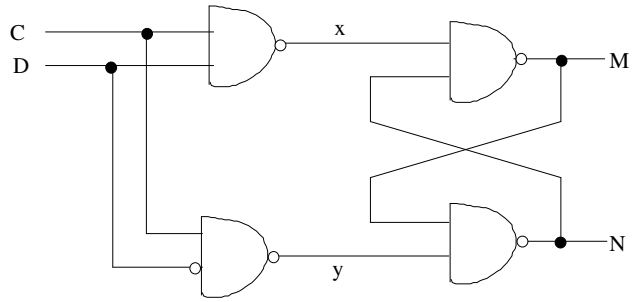
Miten ohjausosa on konstruoitava, jotta muistiosa ja koko piiri toimii halutulla tavalla? Analysoidaan piirin toiminta yksityiskohtaisesti. Ulostulon M rinnalle on merkitty apulinja N. Käydään ensin läpi kaikki mahdolliset (x,y,M,N)-yhdelmät ja katsotaan miten muistiosa käyttäytyy. Osa tapauksista on stabiileja: piiri menee johonkin tilaan ja pysyy siinä, ellei ohjausta muuteta. Kaikissa tapauksissa piiri ei käyttäydy stabiilisti, vaan ensimmäisen muutoksen jälkeen muuttuu uudelleen, mutta stabiloituu lopulta. Tällaisia tiloja kutsutaan stabiloituviksi. Joillakin ohjauksilla piirin tila ei vakiinnu lainkaan, vaan se jää heilahtelemaan kahden tilan välille. Nämä ovat labiileja tiloja.

x	y	M_t	N_t	M_{t+1}	N_{t+1}	stabiili	stabiloituva	labiili
0	0	0	0	1	1		*	
0	0	0	1	1	1		*	
0	0	1	0	1	1		*	
0	**0**	**1**	**1**	**1**	**1**	*		
0	1	0	0	1	1		*	
0	1	0	1	1	1		*	
0	**1**	**1**	**0**	**1**	**0**	*		
0	1	1	1	1	0		*	
1	0	0	0	1	1		*	
1	**0**	**0**	**1**	**0**	**1**	*		
1	0	1	0	1	1		*	
1	0	1	1	0	1		*	
1	1	0	0	1	1			*
1	**1**	**0**	**1**	**0**	**1**	*		
1	**1**	**1**	**0**	**1**	**0**	*		
1	1	1	1	0	0			*

Taulukosta nähdään, että vain yhdellä ohjauksella, ohjauksella (x,y) = (1,1), esiintyy kaksi stabiilia tilaa, (M,N) = (0,1) ja (M,N) = (1,0). Kaksi stabiilia tilaa on välttämätöntä, jos piirin halutaan kykenevän muistamaan kumman tahansa kahdesta eri tilasta, 0 ja 1. Näin ollen ohjaus (x,y) = (1,1) on ainoa mahdollinen valinta muistiosan lepo-ohjaukseksi. Lisäksi havaitaan, että ohjausta (x,y) = (0,0) lukuun ottamatta kaikissa stabiileissa tiloissa $M = \overline{N}$. Valitaan kiikun ykköstilaksi tila (M,N) = (1,0) ja nollatilaksi tila (M,N) = (0,1), jolloin kiikun muistama bitti = M. Kiikun toiminnan tulee siis olla (merkintä - tarkoittaa 'ei väliä'):

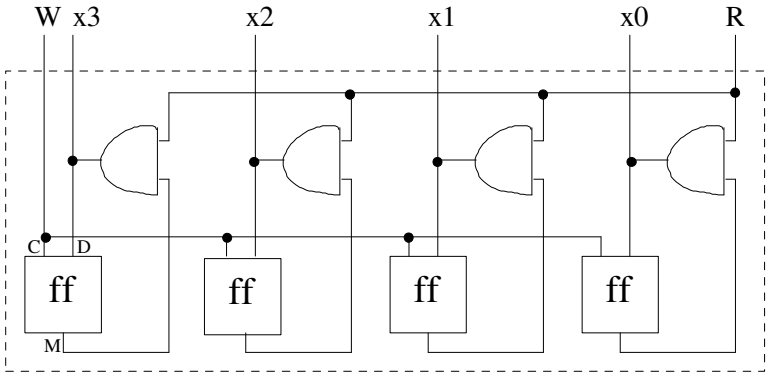
Lepotila	C = 0, D = -:	x = 1, y = 1	M_t= 0 M_{t+1} =0
Lepotila	C = 0, D = -:	x = 1, y = 1	M_t= 1 M_{t+1} =1
Asetus 0	C = 1, D = 0:	x = 1, y = 0	M_t= - M_{t+1} =0
Asetus 1	C = 1, D = 1:	x = 0, y = 1	M_t= - M_{t+1} =1

Täten x=1, kun C=0 tai C=1 ja D=0, siis $x = \overline{C} + \overline{C} + C\overline{D} = \overline{CD}$. Vastaavasti y=1 aina paitsi kun C=1 ja D=0, siis $y = \overline{C\overline{D}}$. Tällainen ohjaus estää myös epästabiilin käytöksen, joka tapahtuu, jos x ja y tulevat yhtaikaa nolliksi. NAND-veräjistä rakennettu kiikku on siis seuraavanlainen:



4.3.4 Rekisteri

Useamman bitin mittaisen sanan tallentamiseen tietokoneessa käytetään ***rekisteriä***, joka rakennetaan kiikuista. Neljäbittinen rekisteri näyttää seuraavalta:



Rekisterissä on neljä kiikkua, jotka on kytketty neljän linjan muodostamaan väylään, jota käytetään sekä lukemiseen että kirjoittamiseen. W (write) ja R (read) toimivat ohjausbitteinä. Lepotilassa R = W = 0. Kun W = 1 ja R = 0, väylällä oleva luku $x_3x_2x_1x_0$ kirjoittuu rekisteriin, kukin bitti omaan kiikkuunsa. Kiikkuun aikaisemmin tallennettu bitti ei häiritse tallennusta, sillä AND-portti R:n arvon ollessa 0 syöttää linjalle nollaa. Piirissä on hiukan oikaistu, koska siinähän on kaksi johtoa liitetty yhteen ilman porttia (AND-portin tulos ja databitti). Todellisuudessa tässä on kytkentä, joka irrottaa AND-portin ulostulon linjasta, kun R=0 ja W=1. Kun R = 1 ja W = 0, piiri antaa johtoihin x_3, x_2, x_1 ja x_0 aikaisemmin tallennetut bitit.

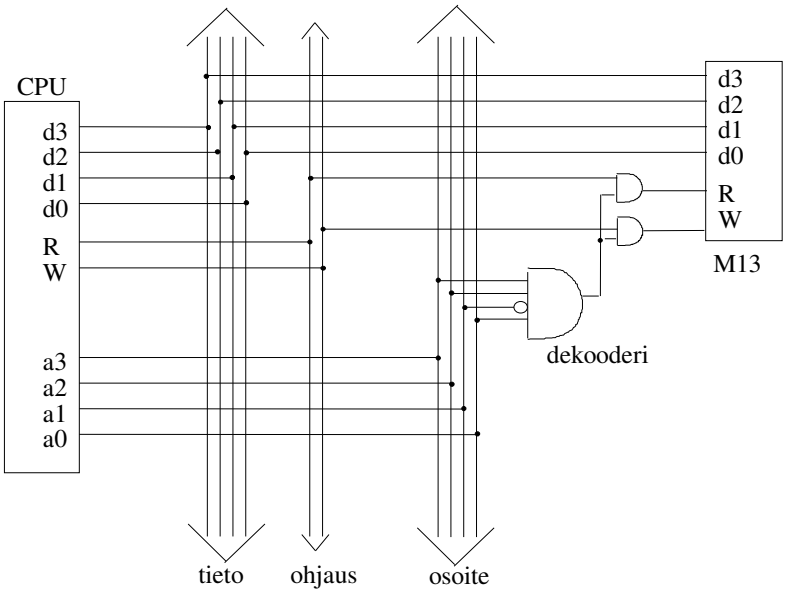
4.3.5 Väylät

Tietokoneen eri komponentteja yhdistäviä tiedonsiirtoyhteyksiä kutsutaan ***väyliksi*** (bus). Väylä on kokoelma komponentit yhteenliittäviä johtoja. Väyliä on toiminnallisesti kolmenlaisia: tietoväylä, ohjausväylä ja osoiteväylä. Sama fyysinen väylä voi toisinaan toimia useammassa tehtävässä. Tietoväylä on tarkoitettu varsinaiseen tiedon siirtoon. Tietoväylän leveys (johtojen lukumäärä) on yleensä sama kuin tietokoneen sanakoko. Ohjausväylä ohjaa tiedon siirtoa mm. käynnistäen tiedon siirron. Osoiteväylä määrää, minne tai mistä tieto siirretään. Kun tietoa halutaan tallentaa johonkin muistipaikkaan, tallennettava tieto lähetetään tietoväylään, kyseisen muistipaikan osoite lähetetään osoiteväylään ja tieto toimenpiteen laadusta lähetetään ohjausväylään.

Koska tietoväylä voi siirtää ainoastaan yhden sanan kerrallaan, on välttämätöntä määrätä, mitkä komponentit saavat käyttää väylää annettuna aikana. Tämä voidaan tehdä (periaatteessa) liittämällä jokainen komponentti väylään AND-piirillä, jonka toisena syötteenä on aktivoiva ohjaussignaali. Kun ohjaussignaali pannaan päälle (eli sen arvoksi tulee 1), komponentti aktivoituu ja sen sisältämä tieto siirtyy väylään. Todellisuudessa useamman komponentin liittäminen samaa väylään ei onnistu AND-piirillä, vaan tulee käyttää esim. kolmitilapuskureita ja multiplekseriä.

Seuraavassa kuvassa (*muuta* kuvassa CPU:n R ja W päittäin, jolloin W:n ja R:n arvojen looginen merkitys on sama sekä CPU:ssa että muistipaikoissa) on keskusyksikkö (CPU), joka on kytketty tieto-, ohjaus- ja osoiteväylään. Kuhunkin väylään on myös kytketty lukuisia rekistereitä, joilla kullakin on oma osoite. Kullakin muistipaikalla on oma dekooderinsa, joka tunnistaa ko. muistipaikan. Muistipaikan, jonka osoite on m, tunnistaminen tarkoittaa tässä sitä, että kun dekooderille annetaan syötteenä ko. muistipaikan osoite binäärisenä, niin dekooderin tuloksena tulee olla 1, ja 0 kaikilla muilla osoitteilla.

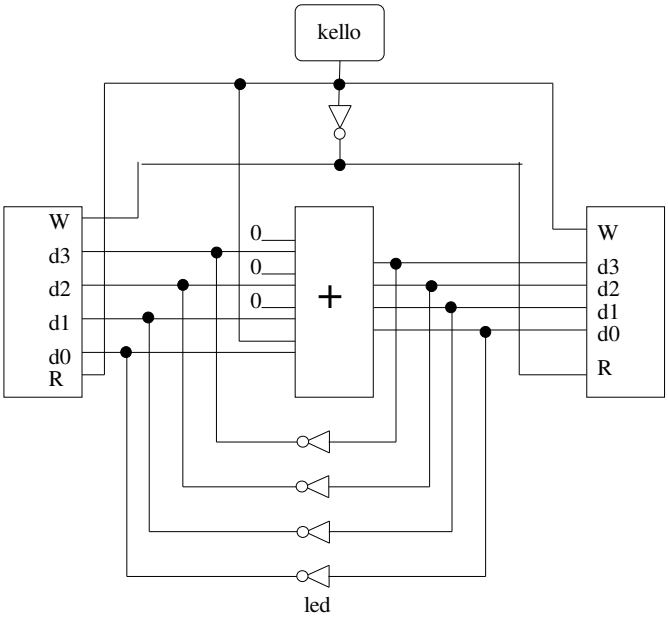
Kuvassa on muistipaikka M13 ja sille lähetetyt ohjauskäskyt tunnistava ***dekooderi***, joka aktivoituu osoitteesta $13 = 1101_2$. Tällöin a3=1, a2=1, a1=0, a0=1, joka siis aktivoi muistipaikan M13. Ohjausbiteillä R ja W ohjataan tiedon (’d niin kuin data’ eli bitit d3, d2, d2, d1, d0) siirron suuntaa CPU:n ja muistipaikan välillä. Esimerkiksi kun CPU:ssa R=1, W=0, niin M13:ssa R=0 ja W=1 eli tällöin CPU:n data kopioituu muistipaikkaan M13 (M13:n data muuttuu, mutta CPU:n data ei muutu). Jos taas CPU:ssa R=0, W=1, niin M13:ssa R=1 ja W=0 eli tällöin muistipaikan M13 data kopioituu CPU:n databitteihin (CPU:n data muuttuu, mutta M13:n data ei muutu).



4.3.6 Ohjauslogiikka ja kello

Tietokoneen yhteenliitetyt komponentit tarvitsevat ohjaussignaaleja toimintansa valvontaan. Ohjaussignaali määrää esimerkiksi, milloin kiikku voi tallentaa syötteensä ja mikä komponentti voi siirtää tietoaan väylään. Ohjaussignaalit ovat peräisin tietokoneen ohjauslogiikkakomponentista, joka vastaa oikean tiedon siirtämisestä sopivana aikana komponentilta toiselle ja varmistaa tietoihin kohdistettavien operaatioiden suorituksen. Ohjauslogiikkakomponentin tärkeä osa on kello, joka tuottaa sakara-aaltoa, vuorottelevia ykkös- ja nollasignaaleja tietyllä nopeudella. Ohjauslogiikkakomponentti varmistaa näillä signaaleilla, että tietokoneen komponentit toimivat synkronisesti ja kaikki komponentit saavat riittävästi aikaa operaatioidensa toteutukseen. Yhden signaalin aikana tietokoneessa toteutetaan aina yksi alkeistapahtuma. Niinpä tietokone toimii sitä nopeammin, mitä suurempi on sen kellotaajuus. Kellotaajuutta rajoittavat käytettyjen puolijohdekomponenttien ominaisuudet.

Esimerkki. Kuvan kytkennässä on kaksi rekisteriä ja yhteenlaskulaite. Kellon ohjaamana laite laskee 0,1,2,...,15,0,1,2,... Nämä luvut ovat luettavissa binäärisenä johtoihin kytketyistä valodiodeista (LED, light emitting diode). Kun kellosignaali on ykkönen, on vasemmanpuoleinen rekisteri aktivoitu lukemista ja oikeapuoleinen kirjoittamista varten. Tällöin vasemmanpuoleisessa rekisterissä olevaan lukuun lisätään ykkönen ja tulos tallennetaan oikeanpuoleiseen rekisteriin. Kun kellosignaali vaihtuu, vaihtuvat myös rekisterien roolit, ja oikeanpuoleisen rekisterin sisältö kopioituu vasemmanpuoleiseen. Samalla ykkösbittien kohdalla olevat ledit palavat.



4.4 Mikro-ohjelmoitava tietokone

Ohjelmoitavuus

Tietokoneen erottaa muista automaattisista koneista sen ***ohjelmoitavuus***. Tietokoneen voi ohjelmoida tekemään uusia tehtäviä muuttamatta sen rakennetta, kun taas laskukoneet, automaattipesukoneet jne. pystyvät suorittamaan vain niitä tehtäviä, joita varten ne on rakennettu. Ohjelmoitavuuden perustana on se, että osa laitteessa kulkevasta, käsittelyn kohteena olevasta tiedosta ohjaa laitteen itsensä toimintaa. Oleellista on, että tietokone kykenee muuttamaan omaa ohjelmointiaan, siis efektiivisesti muuttamaan sen loogisen piirin rakennetta, josta tietokone muodostuu. Loogisten komponenttien toimintaa ohjataan erityisistä ohjaussignaaleista koostuvilla mikrokäskyillä.

Mikro-ohjelmointi

Ensimmäiset tietokoneet suunniteltiin ja rakennettiin siten, että ne pystyivät suorittamaan konekielistä ohjelmaa suoraan. Pian huomattiin, että eri konekielten käskyt muistuttivat toisiaan. Todettiin, että on mahdollista ottaa erilleen pieni hyvin matalan tason perusoperaatioiden joukko, ***mikrokäskyt*** (microinstructions), ja ohjelmoida kukin konekielinen käsky mikrokäskyjä käyttäen. Nykyiset tietokoneet onkin rakennettu siten, että ne suorittavat mikrokäskyjä eivätkä suoraan konekielisiä käskyjä. Jotta konekielisiä ohjelmia voidaan suorittaa, tarvitaan mikrokäskyillä kirjoitettu tulkki, joka osaa tulkita (lukea, ymmärtää, suorittaa) konekielisiä käskyjä. Tulkilla varustettu tietokone pystyy näin suorittamaan konekielisiä ohjelmia. Tällä tavoin rakennettuja koneita sanotaan ***mikro-ohjelmoitaviksi*** (microprogrammed) tietokoneiksi, ja mikrokäskyillä kirjoitettuja ohjelmia kutsutaan mikro-ohjelmiksi (microprograms). Mikro-ohjelmista käytetään myös termiä laitelmisto (firmware).

Kaupallisesti saatavilla olevat tietokoneet voidaan jakaa kolmeen ryhmään: suurtietokoneet, minitietokoneet ja mikrotietokoneet (microcomputers). Ns. RISC- prosessorilla varustetusta (Alpha, MIPS, PowerPC, SPARC) suorittavat suoraan konekielisiä käskyjä.

Kuitenkin hyvin monet käytössä olevat tietokoneet ovat mikro-ohjelmoitavia koneita (mikro-ohjelmoitava tietokone ei ole sama kuin mikrotietokone!) tai välimuotoja mikro-ohjelmoitavasta ja RISC-prosessorilla varustetusta tietokoneesta. Niissä osa käskyistä suoritetaan mikrokoodin avulla ja osa tulkataan prosessorin RISC-ytimelle. Seuraavassa käsitellään kuitenkin yksinkertaistetun mikro-ohjelmoitavan koneen rakennetta ja ohjelmointia.

4.4.1 Mikro-ohjelmoitavan tietokoneen rakenne

Esitettävän mikro-ohjelmoitavan tietokoneen sanakoko on 16 bittiä, ja se muodostuu rekistereistä, muisteista, yhteenlaskulaitteesta, väylistä (dataväylät DC1, DC2 ja DC3 sekä ohjausväylä CC) sekä viisivaiheisesta kellosta. Mikro-ohjelmoitavan tietokoneemme **rakennekaavio** on samassa paikkaa, josta tämä materiaali on ladattavissa, ja sitä tulee tutkia lukiessasi seuraavaa.

Mikro-ohjelmoitavan tietokoneen rekisterit ovat 22-bittinen mikrokäskyrekisteri MIR, siihen liittyvä 8-bittinen osoiterekisteri eli mikro-ohjelmalaskuri MPC, 16-bittinen data-rekisteri MDR ja sen osoiterekisteri 12-bittinen MAR, sekä 16-bittiset erikoisrekisterit A, B, C ja D. Tietojen siirtoa rekisteristä ja rekisteriin ohjataan aktivoivilla ohjausbiteillä (kontrollisignaaleilla).

Mikrokäskyrekisteri MIR (microinstruction register)

MIR on 22-bittinen rekisteri, joka sisältää kulloinkin suoritettavana olevan mikro-käskyn. Käskyn bitit (c1 – c22) toimivat koneen muiden komponenttien ohjausbitteinä. Rekisterissä oleva käsky siis määrää, miten tietokoneen muodostava valtavan monimut-kainen looginen piiri kulloinkin toimii. MIR on kytketty ohjausväylään CC, ja se toimii kellon ohjaamana siten, että sen bitit tulevat aktiivisiksi vuorollaan kellon vaiheissa 1–4.

Mikro-ohjelmamuisti MPM (micro program memory), ROM

Mikro-ohjelmamuisti on mikro-ohjelman käskyjen säilytyspaikka. Sen hyvin nopeat muistipaikat ovat 22-bittisiä ja sen koko on 256 muistipaikkaa numeroituna 0 – 255. Mikro-ohjelmamuistia käsitellään sijoittamalla osoite MPC-rekisteriin. Vastaavan mikro-ohjelmamuistin muistipaikan sisältö siirretään MIR-rekisteriin kellon vaiheessa 5. Mikro-ohjelmamuisti on lukumuisti (ROM). Kun mikro-ohjelmaa suoritetaan, sen käskyjä tarvitsee vain lukea, ei muuttaa. Mikro-ohjelmamuistin sisältö 'poltetaan' muistiin tietokoneen rakentamisen yhteydessä.

Mikro-ohjelmalaskuri MPC (microprogram counter)

MPC on 8-bittinen rekisteri, joka ilmoittaa suoritettavana olevan käskyn osoitteen mikro-ohjelmamuistissa. MPC-rekisteriä käytetään kellon vaiheessa 4 uuden käskyn hakuosoitteen muodostamiseen, ja sen sisältö muuttuu ilman eri ohjausta aina kellon vaiheessa 5, jolloin väylän 3 sisältämä uusi osoite tallentuu MPC-rekisteriin. Koska MPC:n koko on 8 bittiä, mikro-ohjelmamuistin koko on $2^8 = 256$ muistipaikkaa (muistipaikat 0 – 255).

Datarekisteri MDR (memory data register)

MDR on 16-bittinen rekisteri, jota käytetään päämuistissa olevan tiedon siirtämiseen rekistereihin käsittelyä varten. MDR-rekisteriä voidaan käyttää kellon vaiheessa 1 (c6 =

1) tiedon siirtämiseen MDR-rekisteristä aritmeettiseen käsittelyyn, kellon vaiheessa 2 (c13 = 1) tiedon siirtämiseen aritmeettisesta käsittelystä MDR-rekisteriin sekä kellon vaiheessa 3 (c16 = 1) tiedon siirtämiseen MDR-rekisteristä päämuistiin tai (c15 = 1) tiedon siirtämiseen päämuistista MDR-rekisteriin. Lisäksi kellon vaiheessa 4 MDR-rekisterin sisällöstä voidaan siirtää 4 eniten merkitsevää bittiä väylään 1 (c21 = 1), mistä piirteestä lisää myöhemmin.

Päämuisti MM (main memory), RAM

Päämuisti on konekielisten ohjelmien ja niiden käsittelemän tiedon säilytyspaikka. Päämuistin muistipaikat ovat 16-bittisiä, ja se on hitaampi kuin mikro-ohjelmamuisti (tietoa haetaan harvemmin päämuistista kuin mikro-ohjelmamuistista). Tiedon siirto muistiin ja muistista tapahtuu MDR-rekisterin kautta. Tiedon osoite (0 – 4095) ilmoitetaan 12-bittisessä MAR-rekisterissä. Koska päämuistia voidaan käyttää kellon vaiheessa 3 sekä tiedon tallentamiseen (c16 = 1) että lukemiseen (c15 = 1), se on luku- ja kirjoitusmuisti (RAM).

Osoiterekisteri MAR (memory address register)

MAR on 12-bittinen rekisteri, jonka sisältöä käytetään osoitteena, kun päämuistista haetaan tietoa tai kun sinne tallennetaan tietoa. Rekisteriä voidaan käyttää kellon vaiheessa 2 (c14 = 1) kirjoitettaessa MAR-rekisteriin ja kellon vaiheessa 3 (c15 = 1 tai c16 = 1) luettaessa MAR-rekisteriä. Koska MAR:n koko on 12 bittiä, päämuistin koko on $2^{12} = 4096$ muistipaikkaa (muistipaikat 0 – 4095).

Rekisterit A, B, C ja D

A, B, C ja D ovat 16-bittisiä rekistereitä, jotka on tarkoitettu tietojen (laskutoimitusten operandien, välitulosten yms.) säilyttämiseen. Rekistereitä voidaan käyttää kellon vaiheessa 1 (A: c1 = 1, B: c2 = 1, C: c3 = 1, D: c4 = 1) luettaessa rekisteriä ja kellon vaiheessa 2 (A: c9 = 1, B: c10 = 1, C: c11 = 1, D: c12 = 1) kirjoitettaessa rekisteriin. Lisäksi rekisteriä A voidaan tutkia kellon vaiheessa 4: onko sisällön arvo 0 (c19 = 1) tai onko sisällön arvo negatiivinen (c20 = 1).

Aritmeettinen yksikkö (ALU)

Mikro-ohjelmoitavan tietokoneen laskutoimitukset suorittaa 16-bittinen kokosummain. Yhteenlaskun lisäksi laite osaa myös vähennyslaskun: väylällä 1 oleva luku voidaan muuttaa vastaluvukseen ennen yhteenlaskua. Lisäksi yhteenlaskun tulosta voidaan siirtää yhden bitin verran vasemmalle, jolloin vasemmanpuoleisin bitti menetetään ja oikeanpuoleisimman bitin arvoksi tulee nolla (shiftleft-operaatio). Tämä vastaa luvulla 2 kertomista (modulo 2^{16}).

Väylät

Mikro-ohjelmoitavassa tietokoneessa on 4 väylää: kolme 16-bittistä tieto/osoiteväylää (DC1, DC2, DC3) ja yksi 22-bittinen ohjausväylä (CC). Dataväyliä 1 ja 2 käytetään annettaessa syötettä yhteenlaskulaitteelle, ja väylään 3 välitetään yhteenlaskulaitteen tuloste. Väyliä käytetään myös mikro-ohjelmamuistin (MPM) ja päämuistin (MM) osoitteiden muodostamiseen ja siirtämiseen osoiterekistereihin muistien käyttöä varten. Ohjausväylään CC kytketty MIR-rekisteri ohjaa tiedon siirtoa eli koneen toimintaa.

Kello

Mikro-ohjelmoitavan tietokoneen toimintoja ohjaa viisivaiheinen kello, joka antaa pulssin syklisesti kuhunkin viiteen johtoon, jotka aktivoivat mikrokäskyrekisterin MIR bitit seuraavasti:

Ohjausbitit

- kello 1
  - c1 rekisterin A sisältö kirjoitetaan väylään 2
  - c2 rekisterin B sisältö kirjoitetaan väylään 2
  - c3 rekisterin C sisältö kirjoitetaan väylään 2
  - c4 rekisterin D sisältö kirjoitetaan väylään 2
  - c5 luku 1 kirjoitetaan väylään 1
  - c6 rekisterin MDR sisältö kirjoitetaan väylään 1
  - c7 väylän 1 sisältö muunnetaan vastaluvukseen (vähennyslaskua varten)
  - c8 yhteenlaskun tulosta väylällä 3 siirretään 1 bitti vasemmalle
- kello 2
  - c9 väylän 3 sisältö luetaan rekisteriin A
  - c10 väylän 3 sisältö luetaan rekisteriin B
  - c11 väylän 3 sisältö luetaan rekisteriin C
  - c12 väylän 3 sisältö luetaan rekisteriin D
  - c13 väylän 3 sisältö luetaan rekisteriin MDR
  - c14 väylän 3 sisältö luetaan rekisteriin MAR
- kello 3
  - c15 rekisterin MAR osoittaman päämuistin muistipaikan sisältö luetaan rekisteriin MDR
  - c16 rekisterin MDR sisältö kirjoitetaan rekisterin MAR osoittamaan päämuistin muistipaikkaan
- kello 4
  - c17 luku 1 kirjoitetaan väylään 1
  - c18 rekisterin MIR 8 eniten merkitsevää bittiä kirjoitetaan väylään 1
  - c19 luku 1 kirjoitetaan väylään 1, jos rekisterin A sisältö on nolla, muutoin kirjoitetaan luku 2 väylään 1
  - c20 väylään 1 kirjoitetaan luku 1, jos rekisterin A eniten merkitsevä bitti on 1, muutoin luku 2
  - c21 rekisterin MDR 4 eniten merkitsevää bittiä kirjoitetaan väylään 1
  - c22 rekisterin MPC sisältö kirjoitetaan väylään 2
- kello 5
  [Ei ohjausbittejä]

4.4.2 Yhteenveto

Koneessa on siis neljä väylien yhdistämää kokonaisuutta: ensiksikin MIR, MPM, MPC ja kello, toiseksi MDR, MM ja MAR, kolmanneksi aritmeettinen yksikkö ja neljänneksi rekisterit A, B, C ja D, joista erikoisasemassa on rekisteri A. Kussakin kellon vaiheessa suoritetaan jokin tälle vaiheelle tyypillinen toimenpide:

- Kello 1: Rekisterin MDR sisältö tai luku 1 kirjoitetaan väylään 1, ja rekisterin A, B, C tai D sisältö kirjoitetaan väylään 2 aritmeettista käsittelyä varten.
- Kello 2: Aritmeettisen käsittelyn tulos kirjoitetaan väylältä 3 joihinkin rekistereistä MAR, MDR, A, B, C ja D.
- Kello 3: Rekisterin MDR sisältö kirjoitetaan rekisterin MAR osoittamaan paikkaan päämuistissa, tai rekisterin MAR osoittaman päämuistipaikan sisältö kirjoitetaan rekisteriin MDR.
- Kello 4: Lasketaan rekisterin MPC uusi arvo.
- Kello 5: MPC saa uuden arvon, ja MPC:n osoittama mikro-ohjelman käsky siirretään MIR-rekisteriin.

4.4.3 Mikro-ohjelmointi

Mikro-ohjelmoitava tietokone suorittaa päämuistissa olevaa (konekielistä) ohjelmaa. Kukin konekäsky suoritetaan suorittamalla käskyä vastaava mikro-ohjelma, joka on tallennettu mikro-ohjelmamuistiin. Mikro-ohjelmia laadittaessa koneen hyvin rajalliset kyvyt asettavat algoritmeille tiukat vaatimukset. Mikro-ohjelmoitava koneemme kykenee suorittamaan seuraava toiminnat:

- Tiedon siirto rekisterien välillä sekä rekisterin ja päämuistin välillä. Tietoa ei siis voi siirtää suoraan kahden päämuistin muistipaikan välillä. Rekistereistä tieto voidaan siirtää vain sille väylälle, johon kyseinen rekisteri on kytketty.
- Yhteen- ja vähennyslasku. On huomattava, että vähennyslaskussa vähentäjä on aina väylällä 1; kone siis osaa esimerkiksi vähennyslaskun A–MDR, mutta ei laskua MDR–A.
- Kahdella kertominen eli shiftleft-operaatio. Siis vain vasemmalle siirto, ja reunimmainen bitti hukkuu. Merkitään $\times 2$.
- Loogiset vertailut $A < 0$ ja $A = 0$. Vain rekisterille A.
- Siirtyminen mikro-ohjelmassa: kontrollin toteuttamiseksi on käytössä
  - siirtyminen seuraavaan käskyyn
  - ehdollinen ylihyppy (skip)
  - ehdoton hyppy (jump) kun c18=1

Mikro-ohjelmointi on periaatteessa aivan tavallista ohjelmointia. Käytettävässä kielessä on käytössä kiinteä operaatioiden valikoima, ja kussakin mikrokäskyssä merkitään halutut operaatiot toteutettaviksi. Idea on siis hyvin samantapainen kuin mekaanisissa pianoissa tai posetiiveissa. Erona näihin on se, että mikro-ohjelmointi mahdollistaa poikkeamisen käskyjen peräkkäisestä järjestyksestä. Lisäksi on huomattava viisivaiheisen kellosyklin vaikutus: jokainen mikrokäsky sisältää viisi alikäskyä (joista viides on vakio), jotka suoritetaan peräkkäin ennen uuteen mikrokäskyyn siirtymistä. Kussakin kellon vaiheessa suoritettavat alikäskyt voivat koostua useista samanaikaisista toiminnoista. Ainut rajoitus koskee keskenään ristiriitaisia kontrollibittiyhdelmiä: yleensä kullekin väylälle siirretään vain yhden rekisterin sisältö kerrallaan kellon vaiheessa 1, paitsi silloin, kun kirjoitamme ehdottoman hypyn (kun c18=1), jolloin oletamme, että koneemme ei kuitenkaan kaadu. Toisaalta väylän 3 sisältö voidaan sijoittaa useampaankin rekisteriin yhtaikaa. Ohjelmointi on siis tavallaan kaksiulotteista.

Ohjelmointi on äärimmäisen koneenläheistä: jokaisen mikrokäskyn tarkoituksena on suorittaa joko asetuslause tai haarautuminen (joka sekin suoritetaan asettamalla mikro-ohjelmalaskurille uusi arvo). Kello 1 muodostetaan asetuslauseen oikea puoli (mitä lasketaan) ja kello 2 sen vasen puoli (minne laskun tulos sijoitetaan). Kello 3 on varattu muistin käsittelylle ja kello 4 ja 5 ohjelman kontrollille: mikä käsky suoritetaan seuraavaksi.

Tarkastellaan vielä ehdotonta hyppyä. Tällöin tulee seuraavan käskyn osoite siirtää MPC:hen. Se tapahtuu seuraavasti. Asetetaan kontrollibitit c1,...,c8 siten, että kyseinen binääriluku vastaa sitä muistiosoitetta, johon halutaan hypätä ja lisäksi asetetaan c18=1. Tämä saa aikaan myös tiettyjen rekistereiden kopioinnin väylille 1 tai 2, mutta nämä tiedon siirrot eivät kuitenkaan muuta tietokoneen tilaa (rekistereiden arvoja), koska kellon vaiheessa 2 yhteenlaskun tulosta ei viedä mihinkään rekisteriin. Asetus c18=1 saa aikaan sen, että MIR:ssä oleva binääriluku c1...c8 siirtyy väylälle 1 ja yhteenlaskulaitteen jälkeen rekisteriin MPC.

Tarkastellaan seuraavaksi mikro-ohjelmointia muutamien esimerkkien kautta. Ensin tarkastellaan tavallista asetuslausetta ja sen jälkeen paljon monimutkaisempia ohjelmia.

Esimerkki. Lisää rekisterin B sisältöön 1. Tämä saadaan aikaan käskyllä, jossa c2=c5=c10=1 ja muut bitit nollia.

Esimerkki. Lisää rekisteriin A rekisterin B sisältö. Koska väylälle 1 ei voida kopioida kuin rekisterin MDR sisältö, tarvitaan toiminnon toteuttamiseen kaksi mikrokäskyä: ensimmäisessä c2=c13=1 ja toisessa c1=c6=c9=1.

Esimerkki. Oletetaan, että rekisteri D sisältää erään päämuistin muistipaikan osoitteen. Montako 1-bittiä tässä muistipaikassa sijaitseva sana sisältää? Vastaus tulee tallentaa rekisteriin C.

Ratkaisuperiaate: koska ainoastaan sanan ensimmäistä bittiä voidaan tutkia (ykkösbitti paljastuu sanan esittämän luvun negatiivisuutena ja nolla positiivisuutena), lasketaan ykköset siirtämällä sanaa toistuvasti vasemmalle (tämä vastaa kahdella kertomista, mikä onnistuu koneessamme) ja kasvattamalla laskuria aina, kun ensimmäinen bitti on ykkönen. Koska vain rekisterin A negatiivisuutta voidaan testata, tulee päämuistin osoitteessa D oleva muistipaikan sisältö kopioida ensin A:han.

Algoritmi:

```
        C := 0
        A := D:n osoittaman päämuistin muistipaikan sisältö
        WHILE A <> 0 DO
                IF A < 0 THEN C := C+1 ENDIF
                shiftleft(A)
        ENDWHILE
```

Käytössämme ei ole toistorakenteita, joten se tulee korvata hyppykäskyillä. Merkitään osoitteessa x sijaitsevan päämuistin muistipaikan sisältöä (x):llä, ja lisätään hyppykäskyn kohteina olevien käskyjen alkuun käskystä kaksoispisteellä erotettu nimiö:

```
        C := 0
        MAR := D
        MDR := (MAR)
        A := MDR
if:     IF A = 0 THEN jump ohi ENDIF
        IF A < 0 THEN C := C+1 ENDIF
        shiftleft(A)
        jump if
ohi:
```

Tässä (MAR) tarkoittaa siis sen päämuistin muistipaikan sisältöä, jonka osoite on rekisterissä MAR. Algoritmin alkeellinen kontrolli alkaa jo muistuttaa konekieltä, mutta vieläkin se on liian korkeatasoinen. Ehdolliset jump-hypyt eivät kuulu koneemme primitiivisiin operaatioihin, joten ne pitää korvata ehdollisilla ylihypyillä (skip) ja ehdottomilla hypyillä. Tarkoittakoon käsky 'skip' mekanismia, joka ohittaa seuraavan mikrokäskyn ja jatkaa sitä seuraavasta, jos käskyssä tarkasteltava ehto ei toteudu. Tällainenhan meidän tietokoneessamme on mahdollista.

```
        C := 0
        MAR := D
        MDR := (MAR)
        A := MDR
if:     skip A = 0
        jump ohi
        skip A < 0
        C := C+1
        shiftleft(A)
        jump if
ohi:
```

Nyt algoritmi on esitetty tavalla, joka voidaan muuntaa symboliseksi mikro-ohjelmaksi. Otetaan vain huomioon kellosykli ja korvataan nimiöt todellisilla osoitteilla. Kirjoitetaan tässä symbolisessa esityksessä
1) ensimmäinen ja toinen kellonvaihe asetuslauseena (siirtona) x+y→z,
2) kolmas kellon vaihe on päämuistin käsittely ja
3) kirjoitetaan kellon vaiheet 4 ja 5 taas asetuslauseena, jossa MPC saa uuden arvon.
Erotetaan nämä kolme vaihetta puolipisteillä ja merkitään ne aina näkyviin, vaikka jotkin vaiheet olisivatkin tyhjiä. Sovitaan, että ohjelma sijoitetaan mikro-ohjelmamuistiin muistipaikasta 1 alkaen. Osoitteiltaan absoluuttinen, mutta vielä symbolinen mikrokoodi näyttää seuraavalta:

```
1:   0+0 → C; ; 1+MPC → MPC
2:   0+D → MAR; (MAR) → MDR; 1+MPC → MPC
3:   MDR+0 → A; ; 1+MPC → MPC
4:   ; ; (A=0) + MPC → MPC        (ehdollinen hyppy: c19=1)
5:   ; ; 1010₂ → MPC              (ehdoton hyppy: c18=1)
6:   ; ; (A<0) + MPC → MPC        (ehdollinen hyppy: c20=1)
7:   1+C → C; ; 1+MPC → MPC
8:   (0+A)×2 → A; ; 1+MPC → MPC
9:   ; ; 100₂ → MPC
10:
```

Merkintä x+y → z tarkoittaa seuraavaa: rekisterin x sisältö viedään väylään 1 ja rekisterin y sisältö viedään väylään 2, minkä jälkeen yhteenlaskun x+y tulos viedään väylältä 3 rekisteriin z. Merkintä (x+y)×2 tarkoittaa, että väylällä 3 olevaa tulosta siirretään bitin verran vasemmalle. Merkinnät (A=0) ja (A<0) tarkoittavat vastaavien rekisteriä A tarkastelevien laitteiden tulostusta (c19, c20). Joka kellonvaiheen jälkeen väylät nollataan eli esim. siirrossa 0+0 → C pitää asettaa vain c11=1. Lopullinen mikro-ohjelma näyttää seuraavalta (selvyyden vuoksi vain ykköset on merkitty näkyviin, tyhjät bitit ovat nollia):

osoite	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1											1						1					1
2				1										1	1		1					1
3						1			1								1					1
4																			1			1
5					1		1											1				
6																				1		1
7			1		1						1						1					1
8	1							1	1								1					1
9						1												1				

Ohjelman toiminta on seuraava:

1: Kello 1 ei viedä mitään minnekään, jolloin väylillä 1 ja 2 on pelkkää nollaa. Kello 2 väylälle 3 tulee yhteenlaskun tuloksena nolla, joka viedään rekisteriin C. Kello 3 ei tehdä mitään. Kello 4 lasketaan ykkönen väylään 1 ja MPC (= 1) väylään 2, ja yhteenlaskun tulos 1+MPC (= 2) viedään rekisteriin MPC kello 5, jolloin rekisterin MIR sisällöksi tulee mikro-ohjelmamuistin muistipaikassa 2 sijaitseva käsky.

2: Kello 1 viedään rekisteri D väylään 2 ja sieltä edelleen väylää 3 pitkin rekisteriin MAR. Kello 3 haetaan päämuistista rekisterin MAR osoittamasta muistipaikasta sana rekisteriin MDR. Kello 4 lasketaan seuraavan mikrokäskyn osoite 1+MPC, ja kello 5 kirjoitetaan rekisteriin MIR mikro-ohjelmamuistin muistipaikassa 3 sijaitseva käsky.

3: Rekisteri MDR viedään väylälle 1 ja lasketaan yhteen väylän 2 nollan kanssa. Kello 2 tulos viedään rekisteriin A. Kello 3 ei tehdä mitään. Kello neljä ja viisi päivitetään jälleen mikrokäskyrekisteriä kuten edellä: uusi mikrokäsky haetaan paikasta 4.

4: Kello 1, 2 ja 3 ei tehdä mitään. Kello 4 verrataan rekisterin A sisältöä nollaan ja vertailun tulos (= 1, jos A = 0 ja 2, jos A <> 0) viedään väylään 1 ja MPC (= 4) väylään 2. Kello 5 noudetaan käskyrekisteriin seuraava käsky joko paikasta 5 (A = 0) tai paikasta 6 (A <> 0).

5: Kello 1 viedään ykkönen väylälle 1 (c5=1), muutetaan vastaluvukseen (c7=1), lasketaan yhteen väylän 2 nollan kanssa. Kello 2 tulosta (= –1) ei viedä väylältä 3 minnekään, ja sen vuoksi symbolisessa mikrokoodissa kirjoitetaan vaihe 1 tyhjäksi (rivi alkaa ; ;), vaikka kello 1 viedään arvoja väylälle 1 (c5=c7=1). Kello 3 ei tehdä mitään. Kello 4 viedään käskyrekisterin MIR kahdeksan ylintä bittiä 00001010, siis luku 10, väylään 1 ja edelleen väylään 3, josta se asetetaan käskylaskurin MPC uudeksi arvoksi kello 5. Tuloksena on ehdoton hyppy paikkaan 10; seuraavaksi siis suoritetaan mikro-ohjelmamuistin paikassa 10 oleva käsky. Tämä on tarkasteltavan ohjelma-alueen ulkopuolella, joten tulkitsemme kirjoittamamme ohjelman suorituksen loppuvan.

6: Kello 1, 2 ja 3 ei tehdä mitään. Kello 4 verrataan rekisterin A sisältöä nollaan ja vertailun tulos (= 1, jos A < 0 ja 2, jos A ≥ 0) viedään väylään 1 ja MPC (= 6) väylään 2. Kello 5 noudetaan käskyrekisteriin seuraava käsky joko paikasta 7 (A < 0) tai paikasta 8 (A ≥ 0).

7: Kello 1 viedään ykkönen väylään 1 ja rekisteri C väylään 2 ja kello 2 näiden summa väylältä 3 takaisin rekisteriin C. Kello 3 ei tehdä mitään. Kello 4 ja 5 siirrytään jälleen seuraavaan käskyyn.

8: Kello 1 rekisterin A sisältöön lisätään nolla, ja tulosta vieritetään vasemmalle. Kello 2 tulos palautetaan rekisteriin A. Kello 3 ei tehdä mitään. Kello 4 ja 5 siirrytään jälleen seuraavaan käskyyn.

9: Kello 1 MDR viedään väylälle 1, mutta yhteenlaskun tulokselle ei tehdä mitään kello 2. Kello 3 ei tehdä mitään. Kello 4 käskyrekisterin MIR kahdeksan ylintä bittiä 00000100 saavat jälleen toisen merkityksen, kun ne viedään väylälle 1. Ohjelmalaskurin MPC arvoksi tulee siis 4, ja ohjelman suoritus jatkuu käskystä neljä.

Huomaa, että ohjausbittien c1,c2,…,c22 numerointi kulkee eri tavalla kuin rekistereiden ja väylien bittien numerointi. Rekisterien ja väylien bittien numerointi vastaa 2-järjestelmän lukujen potenssien numerointia muunnettaessa 2-järjestelmän lukua 10 järjestelmään: esim. $d_{15,}$ d_{14}, …, d_0, jolloin ylin (merkitsevin, vasemmanpuoleisin) bitti on d_{15}. Sen vuoksi esim. c1 tarkoittaa MIRin ylintä bittiä, jonka indeksi on 21. Lisäksi joskus koneessamme siirretään tietoa eripituisten rekisterien/väylien välillä. Näistä seuraa joitakin asioita, jotka tulee huomata yllä olevassa ohjausbittien c1,…,c22 toiminnan kuvauksessa, kun siirretään dataa tietokoneessamme:

- c5=1: väylän 1 (DC1) alin bitti d_0 saa arvon 1 ja muut ovat nollia.
- c14=1 (esim. käsky 2: yllä): Koska MAR on 12-bittinen ja DC3 on 16-bittinen, niin siirrettäessä DC3:n sisältö MAR:iin, DC3:n 4 ylintä bittiä menee hukkaan. Tämä voidaan kirjoittaa lyhyesti muodossa $DC3_{11\text{-}0} \rightarrow MAR$.
- c18=1 (esim. käsky 5: yllä): tällöin c1…c8 viedään väylälle 1, jolloin väylän 1 'oikeassa laidassa' on kyseinen binääriluku (ja c_8 on siis väylän 1 oikeanpuoleisin bitti eli vähiten merkitsevä bitti, jonka indeksi on nolla). Siis $c1…c8 \rightarrow DC1_{7\text{-}0}$
- c21=1: $MDR_{15\text{-}12} \rightarrow DC1_{3\text{-}0}$.

Koneemme on yksinkertaistettu (verrattuna todellisen tietokoneen prosessorin kuvaukseen) abstrakti malli, koska tarkoituksena on esittää tietokoneen tärkeimmät komponentit ja niiden synkronoitu toiminta. Esimerkiksi, meillä ei ole mitään yksinkertaista keinoa tallentaa tiettyä ykköstä suurempaa lukuvakiota suoraan rekistereihin (esim. A:=123). On tietenkin selvää, että jokaista lukuvakiota ei voi tallentaa päämuistiin. Tämä voidaan ratkaista esimerkiksi seuraavasti: 1) kirjoitetaan mikro-ohjelma, joka 'rakentaa' kyseisen luvun, tai 2) laajennetaan koneemme konfiguraatiota (laitteistoa) niin, että luvut voidaan muodostaa mikrokäskyjen avulla

(kuten esim. ehdottoman hypyn osoite). Samoin kerto- ja jakolasku puuttuvat. Näiden suorittamiseen voidaan kirjoittaa kuitenkin mikro-ohjelma (kuten seuraavassa esitetään), joka noudattaa tavallista 'koulualgoritmia', joskin käytännössä nopeuden vuoksi nämäkin toteutetaan yleensä laitteistolla.

4.4.4 Kertolasku

Kirjoitetaan kertolaskuohjelma, joka laskee rekistereissä A ja C olevien lukujen tulon rekisteriin MDR. Ohjelma perustuu pelkästään yhteenlaskuun, sillä A*C = C + C + ... + C. Ohjelma on naiivi ja hyvin hidas: yhteenlaskuja suoritetaan A–1 kpl. Kertolaskualgoritmi kirjoitetaan ensin käyttäen korkean tason kieltä, jossa on mukana korkean tason kontrollirakenne: WHILE-lause. Sen jälkeen WHILE-lause korvataan sellaisilla hyppylauseilla, jotka voimme koneellamme toteuttaa. Lopuksi tämä koodi muunnetaan symboliseksi mikro-ohjelmaksi, jossa käytetään lisäsivun mukaisia merkintöjä (esim. 0+0→MDR), jonka muuntaminen binääriseksi mikro-ohjelmaksi on suoraviivaista. Huomaa, että mikro-ohjelman lause 4 sisältää kaksi toimintoa: A:n vähennyksen yhdellä ja hypyn lauseeseen 1.

```
MDR := 0
WHILE A <>  0 DO
        MDR := MDR + C
        A := A – 1
ENDWHILE
```

Modifioidaan algoritmia hieman:

```
        MDR := 0
1:      skip A = 0
        jump 5
        MDR := MDR + C
        A := A – 1; jump 1
5:
```

Käskyssä 4 ei käskyyn 1 hyppäämiseen tarvitse käyttää tavallista hyppymekanismia (c18), koska hypyn kohde on osoite 1: pelkkä c17 riittää. Näin säästetään yksi käsky. Tämä on tietysti mitä suurinta kikkailua, mutta kuvastaa hyvin mikro-ohjelmoinnin luonnetta. Vain tehokkuus on tärkeätä, koska ohjelma kirjoitetaan vain kerran, ja sen jälkeen kukaan ei sitä näe. Mikro-ohjelma:

osoite	mikrokäskyn selvennys	selitys
0	0+0 → MDR ; ; 1+MPC → MPC	MDR:lle alkuarvo; siirry seuraavaan käskyyn.
1	; ; (A=0) + MPC → MPC	Haaraudu sen mukaan, onko A = 0 vai ei.
2	; ; 5+0 → MPC	A = 0: Poistu silmukasta (c1 ... c8 = 00000101).
3	MDR+C → MDR; ; 1+MPC → MPC	A ≠ 0: Lisää C MDR:ään.
4	–1+A → A ; ; 1+0 → MPC	Pienennä A:ta, ja hyppää silmukan alkuun.

ja binäärisenä:

osoite	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0													1				1					1
1																			1			1
2						1		1										1				
3			1			1							1				1					1
4	1				1		1		1								1					

Kahden komplementtiesityksen ansiosta tämä mikro-ohjelma toimii oikein myös kun A on negatiivinen. Arvioitaessa ohjelman nopeutta pahimmassa tapauksessa, voidaan päätellä, että koska toistossa on kolme käskyä (käskyt 1, 3 ja 4) ja koska kertojan (A) maksimiarvo on $2^{16} - 1$, suoritettavia käskyjä on $(2^{16} - 1)*3 \approx 200\ 000$. Kukin käsky käyttää 5 kellopulssia, joita on siis yhteensä $5 * 200\ 000 = 1\ 000\ 000$. Jos oletetaan, että kellotaajuus on 10 MHz (todellisuudessa kellotaajuus on tietenkin paljon isompi), kertolaskun suoritusajaksi saadaan $10^6 * 10^{-7}$ s = 0.1 s.

Kertolaskun voi suorittaa nopeamminkin. Käytetään lähtökohtana tavallista koulualgoritmia, esimerkiksi:

```
          0 0 0 1 0 1
          0 0 0 1 1 0
        ___________
          0 0 0 0 0 0
        0 0 0 1 0 1
      0 0 0 1 0 1
    0 0 0 0 0 0
  0 0 0 0 0 0
0 0 0 0 0 0
+_________________
  0 0 0 0 0 0 1 1 1 1 0
```

Algoritmiin on koneemme rajoittuneisuuden vuoksi tehtävä seuraavat muutokset:

- Koska vain sanan ensimmäistä bittiä voidaan tarkastella erikseen, on osasummat "kertojabitti*kerrottava" muodostettava päinvastaisessa järjestyksessä kuin normaalisti, eli vasemmalta oikealle. Kertojan bitit saadaan vuorollaan esiin eli ensimmäiseksi siirtämällä kertojaa vasemmalle.
- Lisätään uusi osasumma summaan heti, kun se on muodostettu.
- Koska tarkastelu etenee vasemmalta oikealle, uutta osasummaa tulisi siirtää oikealle ennen sen lisäämistä summaan. Tämä saadaan aikaan siirtämällä vanhaa summaa vasemmalle ennen uuden summan muodostamista.

Kertolasku näyttää nyt seuraavalta:

```
        0 1 0 1
        0 1 1 0
        _________
        0 0 0 0       1. osasumma
          0 1 0 1     2. osasumma
+       _________
        0 0 1 0 1
            0 1 0 1   3. osasumma
+       _________
        0 0 1 1 1 1
                0 0 0 0  4. osasumma
+       _________
        0 0 1 1 1 1 0  tulo
```

Koska sananpituus on 16, on kertolasku tehtävä 16 kertaa. Luonnollisin ratkaisu olisi käyttää laskuria, joka alustetaan 16:ksi, ja jota vähennetään joka kierroksella yhdellä, kunnes se tulee nollaksi. Vakiota 16 ei kuitenkaan ole käytettävissä suoraan, vaan se pitäisi rakentaa perusoperaatioilla (tark. myöh.). Siksi tässä menetelläänkin toisin. Jotta tiedettäisiin, milloin kaikki kertojan bitit on käsitelty, lisätään kertojan loppuun ykkönen, jolle osasummaa ei muodosteta. Tämä vale-ykkönen tunnistetaan siitä, että se on viimeinen.

Kertolaskun A * C = MDR modifioitu koulualgoritmi:

```
        MDR := 0
        IF A < 0 THEN    (* A:n vasemmanpuoleisin bitti on 1 *)
                MDR := MDR + C
        ENDIF
        shiftleft(A)
        A := A + 1   (* vale-ykkönen *)
loop:   IF A < 0 THEN  (* kertojabitti on 1 *)
                shiftleft(A)
                IF A = 0 THEN (* tällöin ennen edellistä shiftleftiä A on ollut muotoa 10...0 *)
                        lopeta
                ELSE
                        shiftleft(MDR)
                        MDR := MDR + C
                        jump loop
                ENDIF
        ELSE (* kertojabitti on 0 *)
                shiftleft(MDR)
                shiftleft(A)
                jump loop
        ENDIF
```

Mikro-ohjelma:

osoite	mikrokäsky	Selitys
0	0+0 → MDR; ; (A<0) + MPC → MPC	Onko A:n ylin bitti 1?
1	MDR + C → MDR; ; 1 + MPC → MPC	On: lisää C MDR:ään
2	(0+A)×2 → A; ; 1 + MPC → MPC	Siirrä A vasemmalle
3	1 + A→ A; ; 1 + MPC → MPC	Lisää A:n loppuun 1
4	; ; (A<0) + MPC → MPC	Onko A:n ylin bitti 1?
5	; ; 1001 + 0 → MPC	On: hyppää käskyyn 9
6	(MDR+0)×2 → MDR; ; 1 + MPC → MPC	Ei: siirrä MDR vasemmalle
7	(0+A) ×2 → A; ;1 + MPC → MPC	Siirrä A vasemmalle
8	; ; 100 + 0 → MPC	Hyppää silmukan alkuun (4)
9	(0+A) ×2 → A; ; (A=0) + MPC → MPC	Siirrä A vasemmalle; onko A=0?
10	; ; 1110 + 0 → MPC	On: lopeta (käskyyn 14)
11	(MDR+0) ×2 → MDR; ; 1 + MPC → MPC	Ei: siirrä MDR vasemmalle
12	MDR + C → MDR; ; 1 + MPC→ MPC	Lisää C MDR:ään
13	; ; 100 + 0 → MPC	Hyppää silmukan alkuun (4)

Kertolaskun mikro-ohjelman kirjoittaminen binäärisena on suoraviivasta.

Tämä kertolaskuohjelma on huomattavasti aikaisempaa nopeampi. Yhden kertolasku-silmukan pituus on 4 (käskyt 4-6-7-8) tai 6 (4-5-9-11-12-13) käskyä, joten kertolasku vaatii enintään 4 + 15 *6 + 4 = 98 käskyä, mikä 10 MHz:n kellotaajuudella merkitsee 98 *($5*10^{-7}$ s) ≈ 50 μs. Algoritmi on siten noin 2000 kertaa nopeampi kuin naiivi kertolasku!

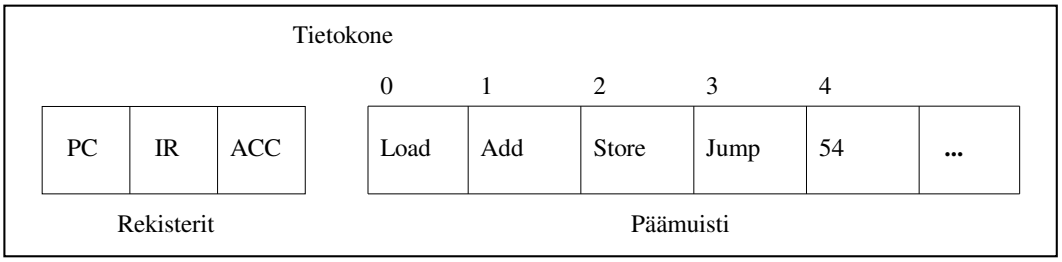
4.4.5 Jakolasku

Kokonaislukujakolaskukin (myös tulos on kokonaisluku) voi pohjautua koulualgoritmiin. Koska luvut ovat binäärisiä, on tehtävä itse asiassa helpompi kuin kymmenjärjestelmässä: sisältyyhän jakaja aina kussakin vaiheessa jaettavaan joko 0 tai 1 kertaa. Jakolaskun mikro-ohjelman esitys kuitenkin sivuutetaan.

4.5 Konekieli

Mikro-ohjelmien kirjoittaminen on ikävää, tarkkaa ja työlästä. Yksittäinen mikrokäsky on hyvin alkeellinen ja voi saada aikaan ainoastaan hyvin yksinkertaisen toiminnon. Mikrokäskyjä tarvitaan paljon yksinkertaisimpienkin toimintojen, kuten kertolaskun, toteuttamiseen. Konekielet ovat helpompia käyttää. Jotta tietokone voisi ymmärtää ja suorittaa konekielisiä käskyjä, tietokonevalmistajat tarjoavat konekielen ohella myös konekielen tulkin, joka on mikro-ohjelma, jonka tehtävä on lukea, ymmärtää ja suorittaa konekielisiä käskyjä. Tulkki sijoitetaan mikro-ohjelmamuistiin (ROM), eikä sitä voi muuttaa. Niinpä ohjelmoijan ei konekieltä (tai korkeamman tason kieltä) käyttäessään tarvitse välittää mikro-ohjelmoitavan tietokoneen ja mikro-ohjelmoinnin yksityiskohdista.

Nostetaan siis jälleen tarkastelun abstraktiotasoa. Abstrakti konekielikone näyttää seuraavalta:

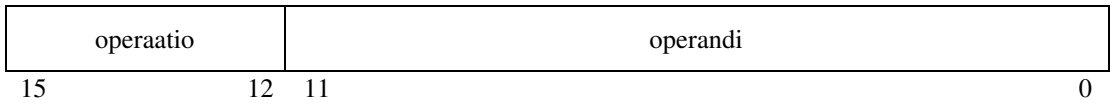


Koneessa on siis kolme erikoisrekisteriä ja päämuisti, johon on tallennettuina bittijonoina konekielen käskyistä koostuva konekielinen ohjelma ja sen käsittelemä data. Käytössä on kolme abstraktia erikoisrekisteriä:

PC rekisteri, jossa on seuraavaksi suoritettavan käskyn osoite päämuistissa
IR rekisteri, johon suoritettava konekielinen käsky tuodaan
ACC rekisteri (akku), jonka sisältöä konekieliset käskyt käsittelevät

Konekielisiä ohjelmia kirjoitettaessa meidän ei tarvitse miettiä näitä rekistereitä, mutta kun seuraavassa luvussa kirjoitamme mikro-ohjelmoidun tulkin konekielelle, meidän tulee kertoa, mitkä koneemme todelliset rekisterit vastaavat näitä abstrakteja rekistereitä. Mikro-ohjelmoinnissa PC:nä toimi todellinen rekisteri MPC ja se sisälsi MPM:n osoitteen, ja IR:nä toimi MIR, johon tuotiin 22-bittinen mikrokäsky.

Konekielten yksityiskohdat ovat konekohtaisia. Tässä esitettävä konekielikone on ns. yksiosoitekone. Jokainen käsky esitetään 16 bitillä. Neljä eniten merkitsevää bittiä yksilöivät operaation ja muodostavat käskyn operaatiokoodin. Loput 12 bittiä muodostavat käskyn operandiosan: ne ilmoittavat sen päämuistin muistipaikan osoitteen, johon operaatio kohdistuu. Akku ACC on implisiittinen operandi useimmissa käskyissä.



Jokainen konekielinen käsky siis määrittää toiminnon, joka kohdistetaan annetusta osoitteesta löytyvän päämuistin muistipaikan sisältämälle datalle (tiedolle). Konekieliset ohjelmat ja niiden käsittelemät tiedot tallennetaan suoritusvaiheessa koneen päämuistiin. Kone suorittaa käskyjä yksi kerrallaan peräkkäin, ellei suoritusjärjestystä muuteta tietyillä konekäskyillä (hyppykäskyt).

Seuraavaksi esitellään käytettävän konekielen käskyt bittimuodossa (16 bittiä: 4-bittinen operaatio + operandin 12-bittinen osoite), symbolisesti sekä käskyn aiheuttama toiminto. Bittimuotoisessa käskyssä m11...m0 tarkoittaa muistipaikan M 12-bittistä osoitetta 2-järjestelmän lukuna, (M) tarkoittaa päämuistin muistipaikan M sisältöä ja ACC akkua.

käsky bittimuodossa		Symbolinen käsky		Toiminto
0001	m11...m0	LOAD	M	ACC ← (M)
0010	m11...m0	STORE	M	(M) ← ACC
0011	m11...m0	ADD	M	ACC ← ACC + (M)
0100	m11...m0	SUBTRACT	M	ACC ← ACC – (M)
0101	m11...m0	MULTIPLY	M	ACC ← ACC * (M)
0110	m11...m0	DIVIDE	M	ACC ← ACC / (M) (kokonaislukujakolasku)
0111	m11...m0	JUMP	M	Hyppää käskyyn jonka osoite on M
1000	m11...m0	JUMPZERO	M	Hyppää M:ään, jos ACC = 0
1001	m11...m0	JUMPNEG	M	Hyppää M:ään, jos ACC < 0
1010	m11...m0	JUMPSUB	M	Hyppää M:stä alkavaan aliohjelmaan
1011	m11...m0	RETURN	M	Palaa M:stä alkaneesta aliohjelmasta

Näin ollen esimerkiksi käskyn LOAD 13, eli lataa (kopioi) akkuun päämuistin muistipaikan 13 sisältö, binäärikoodi on 0001 000000001101.

Huomaa, että edellä mikro-ohjelmoinnin yhteydessä käytettiin nuolta → ilmaisemaan sitä miten tietoa siirrettiin väyliltä rekistereihin: DC1+DC2→DC3, kun taas yllä ← tarkoittaa tavallista asetuslausetta.

Konekielinen ohjelmointi on yksinkertaista mutta työlästä. Seuraavassa esitetään konekielistä ohjelmointia koskevat tärkeimmät huomiot. Konekielinen ohjelma, sen tarvitsema data (syöttötiedot) ja myös ohjelmoinnissa tarvitut apumuistipaikat sijaitsevat kaikki päämuistissa. Lisäksi käytetään erityisasemassa oleva rekisteriä, jota kutsutaan akuksi. Ohjelmoinnissa on keskeisessä asemassa ***akku***, johon kaikki aritmeettiset operaatiot kohdistuvat. Esimerkiksi akun sisältöön voidaan lisätä päämuistin jonkin muistipaikan sisältö (esim. käsky ADD 20 lisää akun sisältöön muistipaikan 20 sisällön). Lisäksi akkuun voidaan kopioida jonkin muistipaikan sisältö (esim. käsky LOAD 20 kopioi akkuun muistipaikan 20 sisällön) tai akun sisältö voidaan kopioida päämuistiin (esim. käsky STORE 20 tallentaa akun sisällön päämuistin muistipaikkaan 20 säilyttäen akun sisällön). Ohjelman kontrollia ohjataan tarvittaessa ***hyppylauseilla***, joissa ilmoitetaan seuraavaksi suoritettavan käskyn osoite päämuistissa. Hyppylause voi olla ehdoton (JUMP) tai ehdollinen akun sisällöstä riippuva (esim. JUMPNEG). Lisäksi on hyvä huomata, että esitetyillä käskyillä ei ole mahdollista ladata akkuun suoraan jotain lukuvakiota (myöhemmin tämä korjataan ottamalla käyttöön komento LOADI), vaan ainoastaan muistipaikan sisältö. Jos esimerkiksi haluamme muuttaa akun sisällön nollaksi, tulee meillä olla nolla tallennettuna johonkin päämuistin muistipaikkaan, jonka sisältö ladataan akkuun LOAD-käskyllä.

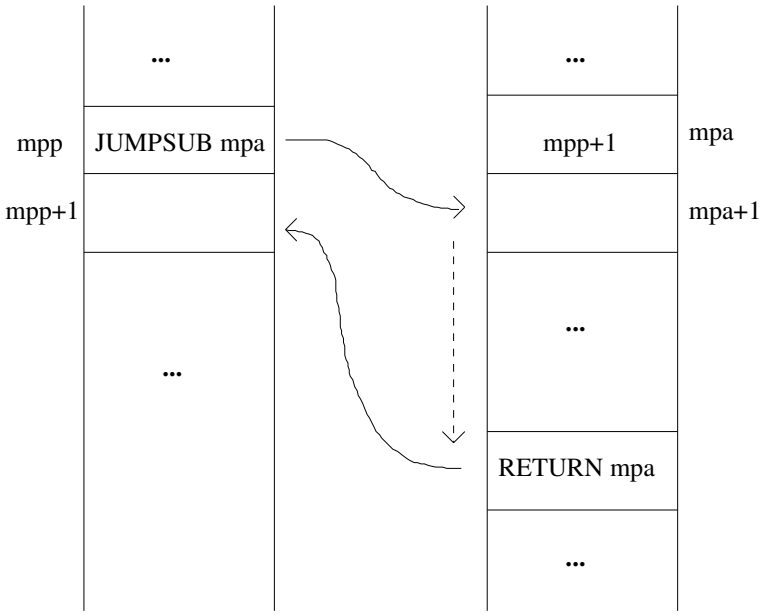
Tässä on hyvä huomata, että päämuistissa oleva data esitetään myös 16 bitillä ja se onko muistipaikan sisältö konekielinen komento vai ohjelman käsittelemää dataa (tietoa) määräytyy vasta ohjelman suoritusvaiheessa. Tarkkaavainen lukija voi tässä vaiheessa huomata myös seuraavan melko kummallisen ominaisuuden: koska konekielinen ohjelma on päämuistissa, voidaan konekielen käskyillä tallentaa uutta tietoa konekielisen ohjelman päälle (siis ohjelma voi muuttaa itse itseään!).

ADD-, SUBTRACT-, MULTIPLY- ja DIVIDE-käskyt kohdistavat nimensä mukaisen binäärisen operaation akun ja operandiosassa annetun muistipaikan M sisältöön ja jättävät operaation tuloksen akkuun. Tällainen ratkaisu juuri johtaa ns. yksiosoite-kieleen, jossa vain toinen binääristen operaatioiden operandi annetaan eksplisiittisesti itse käskyssä ja toinen on aina implisiittisesti akku. On olemassa myös kaksi- ja jopa kolmiosoitekieliä. Kaksiosoitekielissä toinenkaan operandi ei ole aina akku, vaan se

voidaan antaa käskyssä. Kolmiosoitekielissä myös operaation tulos voidaan viedä yhdellä käskyllä haluttuun muistipaikkaan. Yksiosoitekieli on kuitenkin yksinkertaisin ja helpoimmin toteutettavissa mikro-ohjelmoitavassa koneessamme.

JUMP-, JUMPZERO-, JUMPNEG-, JUMPSUB- ja RETURN-käskyt ohjaavat ohjelman suoritusjärjestystä. JUMP-käsky aiheuttaa suorituksen siirtymisen operandiosassa annettuun, muistipaikasta M löytyvään käskyyn (ehdoton hyppy). JUMPZERO-käsky aiheuttaa suorituksen siirtymisen operandiosassa annettuun, muistipaikasta M löytyvään käskyyn, jos akun sisältämä arvo on nolla. Käsky on hyödyllinen erilaisten ehtojen testauksessa, esimerkiksi toistojen päättymisen testaamisessa (ehdollinen hyppy). JUMPNEG-käsky aiheuttaa suorituksen siirtymisen operandiosassa annettuun, muistipaikasta M löytyvään käskyyn, jos akun sisältämä arvo on negatiivinen. Käsky on hyödyllinen negatiivisten lukujen tai ehtojen käsittelyssä (ehdollinen hyppy).

JUMPSUB-käsky aiheuttaa suorituksen siirtymisen operandiosassa annettuun, muistipaikasta M löytyvään käskyyn, josta alkaa aliohjelma (suoritus siirtyy aliohjelmaan, eli suoritetaan aliohjelman kutsu). RETURN-käsky aiheuttaa suorituksen siirtymisen takaisin operandiosan ilmoittamasta, muistipaikasta M alkavasta aliohjelmasta kutsuvaan ohjelman osaan (eli palataan aliohjelmasta). Aliohjelmamekanismi onkin tärkeimpiä korkean tason piirteitä konekielikoneessa. Esimerkki: Olkoon muistipaikassa mpp käsky JUMPSUB mpa. Silloin muistipaikkaan mpa kirjoitetaan paluuosoite mpp+1 ja suoritus jatkuu osoitteesta mpa+1. Kun suoritus sitten tapaa aliohjelman päättökäskyn RETURN mpa, siirtyy kontrolli takaisin paikasta mpa löytyvään paluuosoitteeseen mpp+1. Näin samaa aliohjelmaa voidaan kutsua useasta kohtaa ja kyetään palaamaan takaisin kutsuvaan kohtaan.



Esimerkki. Tarkastellaan ensin hyvin yksinkertaista asetuslausetta (13)←(13)+(10), jolloin tarkoituksena on siis lisätä muistipaikan 13 sisältöön muistipaikan 10 sisältö. Käskyt ovat päämuistin muistipaikoissa 100 jne. ja summa (13)+(10) jää akkuun.

```
100   LOAD    13
101   ADD     10
202   STORE   13
```

Esimerkki. Konekielinen ohjelma, joka laskee eksponenttifunktion 2^n ($n \geq 0$) arvon. Ohjelma on sijoitettu päämuistiin muistipaikasta 371 alkaen. Käsiteltävä data on muistipaikoissa 381-383.

```
MODULE exp(n) RETURNS 2^n
        arvo := 1
        WHILE n > 0 DO
                n := n – 1
                arvo := arvo + arvo
        ENDWHILE
        RETURN arvo
ENDMODULE
```

Konekielinen ohjelma:

```
371   LOAD       383   Hae ykkönen akkuun.
372   STORE      382   Tallenna ykkönen funktion alkuarvoksi.
373   LOAD       381   Hae n akkuun.
374   JUMPZERO   384   Jos n = 0, hyppää muistipaikkaan 384.
375   SUBTRACT   383   Vähennä n:stä yksi.
376   STORE      381   Tallenna n:n uusi arvo.
377   LOAD       382   ACC ← arvo
378   ADD        382   ACC ← arvo + arvo
379   STORE      382   arvo ← ACC
380   JUMP       373   Hyppää silmukan alkuun.
381   n
382   0                funktion arvo
383   1                laskennassa tarvittava lukuvakio
```

Esimerkki. Konekielinen ohjelma, joka laskee summan $2^x + 2^y$. Ilmeinen ratkaisu olisi kirjoittaa kaksi edellisen esimerkin ohjelmaa peräkkäin. Käytetään kuitenkin aliohjelmaa.

```
MODULE summa(x, y) RETURNS 2^x + 2^y
        RETURN exp(x) + exp(y)
ENDMODULE
```

Konekielinen ohjelma:

```
287  LOAD      314   Hae x akkuun.
288  STORE     311   Tallenna x aliohjelman syötteeksi.
289  JUMPSUB   299   Suorita aliohjelma, ts. laske 2^x.
290  LOAD      312   Hae aliohjelman tulos 2^x akkuun.
291  STORE     316   Tallenna lopputuloksen arvoksi.
292  LOAD      315   Hae y akkuun.
293  STORE     311   Tallenna y aliohjelman syötteeksi.
294  JUMPSUB   299   Suorita aliohjelma, ts. laske 2^y.
295  LOAD      312   Hae aliohjelman tulos 2^y akkuun.
296  ADD       316   Lisää 2^x akun (= 2^y) arvoon.
297  STORE     316   Tallenna lopputulos.
298  JUMP      317   Lopeta ohjelman suoritus.
299  0/290/295       Aliohjelman alku, paluuosoitteen tallennuspaikka
300  LOAD      313   Aliohjelman suoritus alkaa, vrt. edellinen esim.
301  STORE     312
302  LOAD      311
303  JUMPZERO  310
304  SUBTRACT  313
305  STORE     311
306  LOAD      312
307  ADD       312
308  STORE     312
309  JUMP      302
310  RETURN    299   Paluu pääohjelmaan
311  0               Aliohjelman syöteparametri
312  0               Aliohjelman tulosparametri
313  1
314  x               Ohjelman syöte x
315  y               Ohjelman syöte y
316  0               Lopputulos
```

4.5.1 Mikro-ohjelmoitu tulkki konekielelle

Aiemmin tietokoneet rakennettiin sellaisiksi, että ne pystyivät suorittamaan konekielisiä käskyjä suoraan. Tällaisen tietokoneen rakentaminen vaatii paljon yksinkertaisia komponentteja (esim. loogisia piirejä). Taloudellisempaa on rakentaa mikro-ohjelmoitava tietokone ja kirjoittaa tulkki, joka on mikro-ohjelma, joka ymmärtää ja suorittaa konekielisiä käskyjä. Nykyiset tietokoneet ovatkin usein ainakin osittain mikro-ohjelmoitavia tietokoneita.

Tulkin perustoimintaperiaate on seuraava: se hakee toistuvasti konekäskyn päämuistista, selvittää käskyn sisällön ja sen jälkeen suorittaa sopivat askeleet (toteuttaa käskyn). Jotta tulkki voisi suorittaa tehtävänsä, sen tulee muistaa seuraavaksi suoritettavan käskyn osoite. Rekisteriä B käytetään seuraavaksi suoritettavan käskyn osoitteen säilytyspaikkana, eli se toimii ohjelmalaskurina PC. Rekisteri A toimii akkuna ACC. Konekäsky haetaan päämuistista rekisteriin käskyrekisterinä IR toimivaan MDR:ään. Huomattakoon, että valinnat A = ACC ja MDR = IR ovat ainoita mahdollisia, koska akun arvoa pitää pystyä vertaamaan nollaan, ja käskyrekisteriin tulee voida noutaa konekieliset käskyt päämuistista. Ohjelmalaskuriksi sen sijaan kelpaisi mikä tahansa mikro-ohjelmoitavan koneen rekisteri.

```
MODULE tulkki
        REPEAT
                MDR := (B)                  (* opr = MDR15-12, opd = MDR11-0 *)
                B := B + 1
                Suorita opr(opd, A, B)   (* hyppykäskyssä myös B voi muuttua *)
        FOREVER
ENDMODULE
```

Rekisterin MDR neljä eniten merkitsevää bittiä (jotka voidaan irrottaa rekisterin muusta sisällöstä asettamalla ohjausbitti c21 päälle) kuvaavat suoritettavan operaation, ja loput 12 bittiä kertovat päämuistin osoitteen. Konekielen mikrotulkki näyttää seuraavalta:

Osoite	Mikrokäskyn toiminnot	Selitys
0	0 + B→MAR; (MAR) → MDR; 1 + MPC→MPC	Hae seuraava käsky MDR:ään.
1	1 + B→B; ; MDR_{15-12} + MPC→MPC	Kasvata ohjelmalaskuria B, ja haaraudu käskyn mukaan.
2	; ; 13→MPC	Käsky oli LOAD, hyppää suorittamaan käskyä.
3	; ; 15→MPC	STORE
4	; ; 17→MPC	ADD
5	; ; 19→MPC	SUBTRACT
6	; ; 21→MPC	MULTIPLY
7	; ; 38→MPC	DIVIDE
8	; ; 61→MPC	JUMP
9	; ; 62→MPC	JUMPZERO
10	; ; 65→MPC	JUMPNEG
11	; ; 68→MPC	JUMPSUB
12	; ; 71→MPC	RETURN
13	MDR_{11-0} + 0→MAR; (MAR)→MDR; 1 + MPC→MPC	LOAD mp alkaa: MDR←mp:n sisältö
14	MDR + 0→A; ; 0 + 0→MPC	ACC←mp; tulkkaa seuraava käsky.
15	MDR_{11-0} + 0→MAR; ; 1 + MPC→MPC	STORE mp alkaa: MAR←mp:n osoite
16	0 + A→MDR; MDR→ (MAR); 0 + 0→MPC	mp←ACC Tulkkaa seuraava käsky.
17	MDR_{11-0} + 0→MAR; (MAR)→MDR; 1 + MPC→MPC	ADD alkaa: MDR←mp:n sisältö
18	MDR + A→A; ; 0 + 0→MPC	ACC←ACC + mp, tulkkaa seuraava käsky.
19	MDR_{11-0} + 0→MAR; (MAR)→MDR; 1 + MPC→MPC	SUBTRACT alkaa: MDR←mp:n sisältö
20	(–MDR) + A→A; ; 0 + 0→MPC	ACC←ACC – mp, tulkkaa seuraava käsky.
21	MDR_{11-0} + 0→MAR; (MAR)→MDR; 1 + MPC→MPC	MULTIPLY alkaa: MDR←mp:n sisältö
22	MDR + 0→C; ; 1 + MPC→MPC Tässä kohdassa nopea kertolaskuohjelma, jonka osoitteet on muutettu vastaavasti.	Siirrä kerrottava edelleen MDR:stä rekisteriin C.
37	MDR + 0→A; ; 0 + 0→MPC	Tulo MDR:ssä: ACC←MDR; tulkkaa seur. käsky.
38	MDR_{11-0} + 0→MAR; (MAR) →MDR; 1 + MPC→MPC	DIVIDE alkaa: MDR←mp:n sisältö

Tässä kohdassa on jakolaskuohjelma. Oletetaan, että sen viimeisen käskyn osoite on 59.

60	0 + C→A; ; 0 + 0→MPC	Osamäärä C:ssä: ACC←C; tulkkaa seur. käsky.
61	MDR + 0→B; ; 0 + 0→MPC	JUMP-käsky. Tulkkaa seuraava.
62	; ; (A = 0) + MPC→MPC	JUMPZERO-käsky alkaa
63	MDR + 0→B; ; 0 + 0→MPC	ACC = 0: Hyppää annettuun muistipaikkaan.
64	; ; 0 + 0→MPC	ACC <> 0: Tulkkaa seuraava käsky.
65	; ; (A < 0) + MPC→MPC	JUMPNEG-käsky alkaa
66	MDR + 0→B; ; 0 + 0→MPC	ACC < 0: Hyppää annettuun muistipaikkaan.
67	; ; 0 + 0→MPC	ACC ≥ 0: Tulkkaa seuraava käsky.
68	MDR + 0→C, MAR; ; 1 + MPC→MPC	JUMPSUB alkaa
69	0 + B→MDR; MDR→ (MAR); 1 + MPC→MPC	Tallenna paluuosoite aliohjelman alkuun.
70	1 + C→B; ; 0 + 0→MPC	Hyppy aliohjelman toiselle riville. Tulkkaa seuraava käsky.
71	MDR_{11-0} + 0→MAR; (MAR)→MDR; 1 + MPC→MPC	RETURN alkaa: Paluuosoite MDR:ään
72	MDR + 0→B; ; 0 + 0→MPC	Paluuosoite ohjelmalaskuriin. Tulkkaa seuraava käsky.

Tulkkiohjelma on poltettu MPM:ään (ROM) muistipaikkoihin 0..72, joista kukin muistipaikka sisältää 22-bittisen ohjausbitin jonon (komennon), joka ohjaa koneemme toimintaa hyvin alhaisella tasolla. Suoritettava konekielinen ohjelma on päämuistissa (RAM) ja aluksi rekisterissä B on ensimmäisen suoritettavan konekielisen käskyn osoite päämuistissa. Tulkkiohjelmaa (joka on siis mikro-ohjelma) aletaan suorittaa asettamalla MPC:hen 0. Tulkin toiminta on yksinkertaisempaa kuin miltä näyttää. Tulkki tulkitsee ja suorittaa konekielistä ohjelmaa käsky kerrallaan. Ensin se hakee päämuistista seuraavaksi suoritettavan konekäskyn, jonka osoite on ohjelmalaskurissa B. Tämän jälkeen ohjelmalaskurin arvoa kasvatetaan yhdellä, joten B sisältää valmiiksi seuraavan (ja tavallisesti siis myös seuraavaksi suoritettavan) käskyn osoitteen. Jos suoritettava käsky onkin hyppykäsky, se ei haittaa, koska B:n arvoa muutetaan tällöin oikeaksi hyppylauseen mukaisesti. Päämuistista luettu käsky tallennetaan rekisteriin MDR, jonka neljän ensimmäisen bitin perusteella voidaan tunnistaa suoritettava operaatio. Tulkki tunnistaa suoritettavan operaation lisäämällä MDR:n neljä eniten merkitsevää bittiä mikro-ohjelmalaskurin MPC arvoon. Mikro-ohjelmalaskurin uuden arvon perusteella mikro-ohjelmassa hypätään mikrokäskyjen 2 - 12 muodostaman hyppykartan avulla suorittamaan konekäskyä vastaavaa mikrokoodia. Lopuksi palataan takaisin tulkin alkuun asettamalla MPC=0.

Esimerkki. Tulkin rivillä 0 konekielisen ohjelman 1. käsky haetaan päämuistista MDR:ään, jolloin siellä on jokin 16 bitin jono; esimerkiksi

'0001 000000001110', joka tarkoittaa konekielistä käskyä 'LOAD 14'

eli lataa akkuun (=koneen rekisteri A) päämuistin muistipaikan 14 sisältö (alussa oleva operaatiokoodi 0001 tarkoittaa komentoa LOAD, jonka jälkeen tulee sen muistipaikan osoite, johon operaatio kohdistetaan). Seuraavaksi tulkin tulee suorittaa kyseinen LOAD-käsky eli tulkkiohjelmassa tulee hypätä käskyn LOAD koodiin, joka alkaa muistipaikasta 13. Tämä hyppy saadaan aikaan lisäämällä operaatiokoodi nykyisen MPC:n arvoon (=1, koska ollaan tulkin rivillä 1), jolloin MPC saa arvon 2. Rivillä 2 on taas tavoiteltu hyppykäsky riville 13. Rivillä 1 lisätään myös ohjelmalaskuriin (=B) 1 eli oletetaan, että ohjelmassa ei ole hyppykäskyä, vaan suoritetaan konekielisen ohjelman seuraava lause. Jos MDR:ssä onkin hyppykäsky, muutetaan B:tä vastaavasti kyseisen hyppykäskyn koodissa.

Esimerkki. ADD-käskyn suorittaminen.

ADD-käskyn (eli laske yhteen akun ja operandina annetun muistipaikan sisältö ja vie tulos akkuun) suoritus: mikrokäskyt 17 ja 18 suorittavat ADD-operaation. MDR sisältää neljässä eniten merkitsevässä bitissään ADD-käskyn operaatiokoodin ja 12 vähiten merkitsevässä bitissään sen muistipaikan osoitteen, jonka sisältö lisätään akun sisältöön. Ensin sijoitetaan MAR:n arvoksi MDR:n arvo. Koska MAR on 12-bittinen, MAR saa arvokseen MDR:n 12 vähiten merkitsevää bittiä (eli muistipaikan osoitteen). Sitten luetaan päämuistista MAR:n osoittamasta osoitteesta muistipaikan sisältö MDR:ään. Sen jälkeen lasketaan yhteen akun A sisältö MDR:n kanssa ja tulos tallennetaan akkuun A. MPC asetetaan yhteenlaskun jälkeen nollaksi, mikä tarkoittaa sitä, että tulkki siirtyy tulkitsemaan ja suorittamaan seuraavaa konekäskyä.

Esimerkki. JUMPZERO-käskyn suorittaminen.

JUMPZERO-käskyn (hyppää operandina annettuun muistipaikkaan, mikäli akun sisältämä arvo on nolla) suoritus: mikrokäskyt 62-64 suorittavat JUMPZERO-operaation. Testattaessa akun A sisältöä lisätään testauksen tuloksena joko 1 (akku = 0) tai 2 (akku <> 0) mikro-ohjelmalaskurin arvoon. Ensimmäisessä tapauksessa suoritetaan seuraavaksi käsky 63, jossa kopioidaan MDR:n sisältö (vain osoiteosa, koska MAR on 12-bittinen) ohjelmalaskurin B sisällöksi, jotta tulkki tietää, mikä konekäsky seuraavaksi suoritetaan. Jälkimmäisessä tapauksessa suoritetaan käsky 64, jonka suoritus ei vaikuta ohjelmalaskuriin B, vaan tulkki siirtyy suorittamaan konekielisen ohjelman seuraavaa käskyä.

4.5.2 Monimutkaisemmat konekielet

Edellä esitelty konekieli on hyvin yksinkertainen. Esitellyssä kielessä käskyjä on vain 11. Oikeissa konekielissä käskyjä saattaa olla satoja. Lisäksi niissä voi olla muita lisäpiirteitä:

- pino-operaatiot, jotka helpottavat aritmeettisten lausekkeiden ja aliohjelmien toteuttamista
- rekisterien lukumäärä ja koko suurempi
- syöttö- ja tulostusoperaatiot
- osoitustavat (addressing)

4.5.3 Osoitustavoista

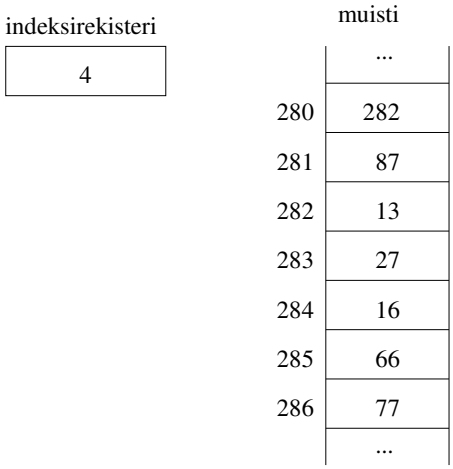
Todellisissa konekielissä on monisteen konekielen käskyissä käytetyn ns. suoran osoitustavan lisäksi käytössä myös muita osoitustapoja. Osoitustavalla tarkoitetaan sitä tapaa, millä konekielen käskyn osoiteosa määrää käskyn todellisen operandin. Tyypillisesti etenkin LOAD- ja STORE-käskyissä on käytettävissä eri osoitustapoja, mutta myös aritmeettisissa operaatioissa voidaan monissa konekielissä valita suoran osoituksen asemesta jokin muu tapa. Osoitustapa osoitetaan esimerkiksi käskyyn liitettävällä loppuliitteellä tai jollakin erikoismerkillä. Tarkoittakoon *oso* seuraavassa (12-bittistä) lukua, (*oso*) luvun *oso* osoittaman muistipaikan sisältöä, ja *X* konekielikoneen tietyn rekisterin, ns. indeksirekisterin arvoa. Tavallisimmat osoitustavat ja niiden merkitys ovat seuraavat:

- ***välitön osoitus*** (immediate addressing): käskyn operandiosa on itsessään käsiteltävä tieto
- ***suora osoitus*** (direct addressing): käsiteltävä tieto sijaitsee käskyn operandiosan ilmoittamassa osoitteessa (muistipaikassa)
- ***epäsuora osoitus*** (indirect addressing): käskyn operandiosa ilmoittaa sen muistipaikan, josta löytyy sen muistipaikan osoite, jossa käsiteltävä tieto sijaitsee

- ***indeksoitu osoitus*** (indexed addressing): käsiteltävä tieto on muistipaikassa, jonka osoite saadaan lisäämällä operandiin erään rekisterin, ns. indeksirekisterin, sisältö
- ***epäsuora indeksoitu osoitus*** (indirect indexed addressing): käsiteltävä tieto on muistipaikassa, jonka osoite saadaan laskemalla yhteen käskyn osoittaman muistipaikan sisältö ja indeksirekisterin sisältö

Osoitustapa	Latauskäsky	Merkitys
välitön (Immediate)	LOADI *oso*	ACC ← *oso*
suora	LOAD *oso*	ACC ← (*oso*)
epäsuora (InDirect)	LOADID *oso*	ACC ← ((*oso*))
indeksoitu (IndeXed)	LOADIX *oso*	ACC ← (*oso*+*X*)
epäsuora indeksoitu	LOADIDX *oso*	ACC ← ((*oso*)+*X*)

Esimerkki. Osoitustavat. Olkoon muistin ja indeksirekisterin X sisältö seuraava:



Osoitustapa	Käsky		Akun arvo
välitön	LOADI	280	280
suora	LOAD	280	282
epäsuora	LOADID	280	13
indeksoitu	LOADIX	280	16
epäsuora	LOADIDX	280	77

Suora osoitus on tavallisin osoitustapa. Siinä konekielisen käskyn osoiteosa tulkitaan viittaukseksi osoitteen osoittamaan muistipaikkaan, ja käskyn todelliseksi operandiksi tulee tämän muistipaikan sisältö. Epäsuorassa osoituksessa muistiviittaus on tavallaan kaksinkertainen: käskyn osoiteosa tulkitaan viittaukseksi muistipaikkaan, jonka sisältö tulkitaan viittaukseksi muistipaikkaan, jonka sisältö tulee käskyn todelliseksi operandiksi. Epäsuoraa osoitusta tarvitaan monissa korkean tason ohjelmointikielissä käytettävien epäsuorien muistiviittausten, kuten osoittimien, toteuttamiseen. Se on samassa tarkoituksessa hyödyllinen myös konekielisessä ohjelmoinnissa.

Indeksoitua osoitusta tarvitaan etenkin rakenteisten muuttujien, kuten vektorien ja taulukoiden käsittelyssä. Konekielen tasolla vektorin alkiot indeksoidaan nollasta alkaen, ja haluttu indeksi tallennetaan indeksirekisteriin ennen alkioon viittausta. Viittaavan käskyn osoiteosaan tallennetaan vain vektorin alkuosoite (ns. ***kantaosoite***). Jotta samalla ohjelmalla voitaisiin käsitellä eri vektoreita, tarvittaisiin vieläkin yleisempiä osoitustapoja. Käytännössä on ainakin kaksi vaihtoehtoa:

a) Käytetään ***kantarekisteriä***, johon tallennetaan kulloinkin halutun vektorin alkuosoite.

b) Käytetään epäsuoraa indeksoitua osoitusta: vektorin alkiota käsittelevä käsky sisältää osoitteen, josta löytyy vektorin alkuosoite.

Välitöntä osoitusta tarvitaan lukuvakioiden käytön mahdollistamiseksi. Tähän saakkahan olemme olettaneet, että konekielisissä ohjelmissa tarvittavat lukuvakiot on etukäteen tallennettu joihinkin muistipaikkoihin. Tämä on kuitenkin hankalaa ja tehotonta. Välitön osoitus on mikro-ohjelmoitavassa koneessamme yllä kuvatuista osoitustavoista kaikkein vaikeimmin toteutettavissa. Miten saadaan mikro-ohjelmoitavan koneen rekisteriin jokin vakio? Seuraavassa mikro-ohjelma, joka tallentaa rekisteriin A luvun 13 (=0000000000001101):

0 (1+0) ×2 → A; 1+MPC → MPC
1 (1+A) ×2 → A; 1+MPC → MPC
2 (0+A) ×2 → A; 1+MPC → MPC
3 (1+A) → A; 1+MPC → MPC

Lukuvakiot on siis rakennettava bitti kerrallaan. Jokaisen lukuvakion konstruointi vaatii oman mikro-ohjelmansa. Mutta miten toteutetaan yleinen konekielen käsky, joka vie akkuun minkä tahansa halutun lukuvakion?

Konekieltä mikrotulkattaessa suoritusvuorossa oleva käsky haetaan ensin päämuistista käskyrekisteriin IR (eli MDR). Esimerkiksi JUMP-käsky voidaan tulkata yksinkertaisesti siirtämällä koko käsky (JUMP *oso*) käskyrekisteristä ohjelmalaskuriin PC (eli rekisteriin B), koska PC:n osoittamaa seuraavaksi suoritettavaa käskyä muistista noudettaessa PC:n sisältämä muistiosoite siirretään osoiterekisteriin MAR, joka on 12-bittinen. Tällöin neljä ensimmäistä bittiä (eli käskykoodi JUMP) vuotavat yli, ja MAR saa arvon *oso*, joten seuraava käsky haetaan sieltä mistä oli tarkoituskin. Näin PC:n sisältämät ylimääräiset neljä bittiä eivät haittaa mitään.

Yllä kuvattu menettely ei käy välittömän latauksen kohdalla, koska akku (A) on 16-bittinen kuten käskyrekisterikin (MDR). Siirrettäessä käsky LOADI *oso* käskyrekisteristä akkuun, todellisen operandin *oso* lisäksi akkuun siirtyisi silloin myös käskykoodi LOADI. Ratkaisu käskykoodin eliminoimiseksi on ns. maskin käyttö. Oletetaan, että välittömän latauskäskyn LOADI operaatiokoodi on 13. Konstruoidaan ensin johonkin rekisteriin LOADI-käskyä vastaava maski, lukuvakio 0011000000000000, ja lasketaan tämä yhteen käskyn LOADI *oso* kanssa:

LOADI oso:	1101 *oso*
maski:	0011 000000000000
summa:	0000 *oso*

Tuloksena on tarkalleen haluttu lukuvakio, joka tallennetaan akkuun. LOADI-käskyn toteuttava mikro-ohjelma jätetään harjoitustehtäväksi.

4.6 Kommunikointi

Tähän saakka olemme tarkastelleet tietokoneen sisäisten komponenttien, keskusyksikön ja keskusmuistin, toimintaa. Jotta tietokonetta voidaan käyttää hyväksi, tulee ohjelmat ja niiden käsittelemät tiedot syöttää tietokoneeseen käyttäen jotain syöttölaitetta. Tietokone tulostaa käsittelynsä tulokset jollekin tulostuslaitteelle.

4.6.1 Syöttö- ja tulostuslaitteet

Tietokoneet siis kommunikoivat käyttäjän (ulkomaailman) kanssa syöttö- ja tulostuslaitteiden (input/output devices, I/O devices) eli I/O-laitteiden välityksellä. Näitä laitteita kutsutaan yhteisellä nimellä siirräntälaitteiksi tai oheislaitteiksi (peripherals).

Syöttölaitteita ovat esimerkiksi:
- näppäimistö
- optinen lukija
- hiiri
- mittalaitteet
- mikrofoni
- kello
- magneettimusteenlukija

Tulostuslaitteita ovat esimerkiksi:
- näyttöruutu
- kirjoitin
- piirturi
- kaiutin

Oheislaitteita, joita voidaan käyttää sekä syöttö- että tulostuslaitteina:
- toiset tietokoneet
- levy- ja muut muistit ja tallennusvälineet
- magneettinauhat

Levymuisteja ja magneettinauhoja kutsutaan myös toissijaiseksi muistiksi eli oheismuistiksi (secondary storage), joissa voidaan säilyttää suuria tietomääriä. Keskusmuistia käytetään tiedon käsittelyyn ja toissijaisia muisteja tiedon säilytykseen.

Konekielissä on yleensä käskyjä, joilla voidaan toteuttaa syöttö- ja tulostustoimintoja (lähettää tietoa oheislaitteille/vastaanottaa tietoa oheislaitteilta).

4.6.2 Keskeytyspohjainen siirräntä

Syöttö ja tulostus ovat hitaita toimintoja verrattuna muihin tietokoneen suorittamiin toimintoihin. Esimerkiksi näppäimistöltä ei voida syöttää sekunnissa kuin muutama merkki, kun taas prosessori pystyy toteuttamaan miljoonia käskyjä sekunnissa (million instructions per second, MIPS). Ilman erikoisjärjestelyjä keskusyksikkö joutuu odottamaan jouten hidasta siirräntätoimintoa. ***Keskeytyspohjaisessa siirrännässä*** keskusyksikkö suorittaa syöttö- ja tulostustoiminnot oheistoimintoina. Tällöin sille annetaan syöttö- ja tulostustoimintojen kanssa samanaikaisesti suoritettavaksi jokin muu tehtävä, jonka suoritus keskeytetään ajoittain syöttö- ja tulostustoimintojen käsittelyä varten.

Jokaisella I/O-laitteella on oma tilarekisterinsä ja puskurirekisterinsä. Tilarekisteri ilmoittaa, missä tilassa laite on, ja puskurirekisteriin kerätään I/O-laitteeseen lähetettävää tai sieltä tulevaa tietoa. Kun jonkin I/O-laitteen tilarekisterin arvo muuttuu, prosessori keskeyttää sillä hetkellä suorittamansa tehtävän ja siirtyy suorittamaan erityistä prosessia, ns. ***keskeytyksen käsittelijää***, jossa käsitellään keskeytyksen aiheuttanut I/O-tapahtuma. Keskeytyksen käsittelijän suorituksen jälkeen prosessori jatkaa keskeytynyttä tehtäväänsä. Prosessori tarkastaa jokaisen kellosyklin jälkeen, onko

minkään I/O-laitteen tilarekisterin arvo muuttunut. Jos arvo on muuttunut, prosessori siirtyy suorittamaan keskeytyksen käsittelijää, muutoin se jatkaa suoritusvuorossa olevaa tehtäväänsä.

Kehittyneimmissä tietokoneissa on erityinen I/O-prosessori I/O-toimintojen suoritusta varten. Keskusyksikkö kertoo suoritettavan I/O-operaation I/O-prosessorille, joka suorittaa operaation ja keskeyttää keskusyksikön vasta sitten, kun koko siirräntäoperaatio on suoritettu. I/O-prosessoreita nimitetään myös DMA-laitteiksi (Direct Memory Access devices) tai kanaviksi (channels).

4.6.3 Tietokoneverkot

Tietokoneiden valmistuskustannusten aletessa tietokoneiden määrä on kasvamistaan kasvanut. Yhä lisääntyvässä määrin on tullut tarpeelliseksi yhdistää koneet toisiinsa esimerkiksi saman organisaation sisällä. Näin yhdistetyt koneet muodostavat yhdessä tietokoneverkon (computer network), joka voi olla paikallinen, alueellinen tai jopa maailmanlaajuinen. Verkossa on tiettyjä koneita (ns. palvelimia), jotka tarjoavat palveluita (tietoa ja ohjelmia) verkossa oleville koneille. Palveluita käytetään ns. asiakasohjelmien avulla.

Tietokoneverkkojen palveluja:

- Sähköposti (e-mail): verkkoon kuuluvien tietokoneiden käyttäjät voivat lähettää toisilleen viestejä verkon kautta
  - viestien automaattinen tallennus
  - viesti-ilmoitusten näyttö
  - vastaaminen ja vastauksen lähettäminen on helppoa ja edullista
  - viestin perillemeno on nopeaa
- Sähköuutiset (news): verkkoon kuuluvien tietokoneiden käyttäjät voivat lähettää ja vastaanottaa verkon välityksellä uutisia ja käydä julkista keskustelua
  - vastaanottajaa ei välttämättä tarvitse nimetä (uutisen lähetys esim. tiettyyn uutisryhmään)
  - uutisryhmät aihepiireittäin: kukin uutisten lukija voi rajoittua seuraamaan vain haluamiensa aihepiirien uutisia
- Tietokantojen yhteiskäyttö: verkon (esim. Internetin) kautta on mahdollista käyttää yhteen tai useampaan verkon koneeseen sijoitettuja tietokantoja
  - kirjaston tietokanta
  - tietokantaan sijoitettu monen käyttäjän yhteisesti valmistelema tekstidokumentti
- Tiedostojen siirto: asiakasohjelmalla (esim. FTP eli File Transfer Protocol) tai sähköpostin liitetiedostona (Attachment).
- Yhteiset syöttö- ja tulostuslaitteet: verkon olemassaolo tekee mahdolliseksi syöttö- ja tulostuslaitteiden yhteiskäytön. Esimerkiksi verkon koneilla voi olla yhteinen kirjoitin.
- Tiedostopalvelijat eli yhteiskäyttöiset levyasemat: sen sijaan, että kukin verkon kone varustetaan suurella ja kalliilla levyasemalla, voidaan verkon yhteen

koneeseen sijoittaa suurikapasiteettisia levyasemia koko verkon yhteiskäyttöön (tiedostopalvelijat eli "serverit")

Tiedosto lähetetään verkossa *paketteina* (frames). Pienet tiedostot lähetetään kokonaisina ja isommat pilkotaan pienempiin paketteihin tiedonsiirtovirheiden välttämiseksi. Paketti muodostuu tunnisteosasta (header) ja lähetettävästä tiedosta (data). Tunnisteosassa on mm. vastaanottajan osoite ja viimeisessä paketissa on lisäksi tieto tiedoston päättymisestä. Yhteen tiedostoon liittyvät paketit voivat saapua määränpäähänsä eri reittejä (reititysalgoritmit), ja pakettien koonti alkuperäiseksi tiedostoksi tapahtuu perillä. Tällaisia tietokoneverkkoja nimitetään pakettiverkoiksi (packet-switching networks).

Tietokoneverkot ovat yleensä hierarkkisia ja poikkeavat toisistaan verkkoon kytkettyjen koneiden lukumäärän ja maantieteellisten etäisyyksien suhteen. Ne voidaan jakaa karkeasti kahteen luokkaan: *laajat verkot* (WAN, Wide Area Network) ja *lähiverkot* (LAN, Local Area Network). Laajat verkot yhdistävät lähiverkot sekä kaupunkiverkot yhdeksi suureksi verkoksi, ääritapauksena Internet.

Laajat verkot:

- useita koneita ja verkon koneet fyysisesti etäällä toisistaan
- yhteydet koneiden välillä toteutettu kaapeleilla, puhelinlinjoilla ja satelliittiyhteyksillä
- viestien lähetys solmukoneiden kautta eli koneet lähettävät viestit toisilleen koneelta koneelle -periaatteella: kukin solmukone vastaanottaa viestin, selvittää sen lähetysreitin ja lähettää viestin eteenpäin, kunnes viestin vastaanottaja on tavoitettu

Lähiverkot:

- verkon koneet fyysisesti lähellä toisiaan (yksi rakennus tai rakennuskompleksi), vähemmän koneita
- neljä perustopologiaa (”kaapelointiarkkitehtuuria”): *väylä* (bus), *rengas* (ring), *tähti* (star) ja *puu* (tree)
- ongelma: yksittäiseen verkkoon voidaan kytkeä vain rajallinen määrä koneita
- lähiverkkoja voidaan liittää yhteen esim. toistinten (repeater), siltojen (bridge), reitittimien (router) avulla, ja arkkitehtuureiltaan erilaisia lähiverkkoja tai laajoja verkkoja voidaan yhdistää ns. yhdyskäytävien (gateway) avulla
- esim. kilpavarausväylä ETHERNET (yleisin), vuororengas ja vuoroväylä
- **WLAN** (Wireless Local Area Network) on langaton lähiverkkotekniikka, jolla erilaiset verkkolaitteet voidaan yhdistää ilman kaapeleita.

  ETHERNET:

  - tietokoneet liitetty yhteen nopeaan väylään, joihin kaikilla koneilla on luku- ja kirjoitusoikeus
  - paketin vastaanotto: kone valvoo väylää jatkuvasti ja vastaanottaa itselleen tarkoitetun paketin osoitteen perusteella

- paketin lähetys: kone käynnistää paketin lähetyksen. Jos jokin toinen kone yrittää lähettää pakettia samanaikaisesti, tapahtuu yhteentörmäys (collision), jolloin molemmat koneet katkaisevat lähetyksen ja yrittävät satunnaisen ajanjakson kuluttua lähetystä uudelleen. Kyseessä on siis ns. *kilpavarausväylä*: lähetysoikeuden saa kone, joka ensimmäisenä ehtii.

Vuororengas (token ring):

- tietokoneet on kytketty renkaaseen, jossa kukin kone voi lähettää vuorollaan paketin (ei siis törmäyksiä)
- renkaassa kiertää vuoromerkki (token) koneelta koneelle. Lähettääkseen paketin tulee koneen saada ensin haltuunsa vuoromerkki, jonka jälkeen se lähettää pakettinsa renkaaseen. Kukin renkaassa oleva kone tarkastaa onko paketti tarkoitettu kyseiselle koneelle. Jos ei, niin se lähettää paketin edelleen renkaan seuraavalle koneelle. Jos on, niin kone kopioi paketin itselleen, liittää tiedon paketin perille tulosta paketin loppuun ja lähettää paketin eteenpäin, kunnes se saapuu takaisin lähettäjälleen, joka poistaa paketin renkaasta ja vapauttaa vuoromerkin.
- ongelma: epäkuntoinen kone verkossa
- esim. IBM Token Ring

Vuoroväylä (token bus):

- väylä- ja vuororengasarkkitehtuurin hybridi
- arkkitehtuuriltaan samanlainen kuin väyläverkko, mutta mahdolliset lähetysvaiheen yhteentörmäykset vältetään vuoromerkkien avulla
- esim. MAP (Manufacturing Automation Protocol, tuotannonohjausprotokolla)

5 Systeemiohjelmisto

Neljännessä luvussa kuvattiin, miten tyypillinen tietokone rakennetaan ja miten tällainen tietokone suorittaa konekielistä ohjelmaa. Esitelty tietokone ei sellaisenaan ole erityisen hyödyllinen. Jotta tietokoneesta saadaan käyttökelpoinen, tulee ratkaista mm. seuraavat ongelmat:

- Miten ohjelma tallennetaan tietokoneen muistiin?
- Miten ohjelma saa tarvitsemansa syöttötiedot ja miten se antaa tulokset?
- Miten ohjelmointi voitaisiin tehdä helpommaksi?
- On tuhlaavaista varata koko kone yhden ohjelman suoritusta varten.
- Miten koneen toiminta hallitaan, jos suoritettavina on useampia ohjelmia?

Laitteisto ei siis yksin tee käyttökelpoista tietokonejärjestelmää. Jotta tietokoneen mahdollisuudet voitaisiin käyttää hyväksi, tarvitaan ns. systeemiohjelmia (system programs, system software), jotka helpottavat laitteiden käyttöä. Koneen 'räätälöinti' käyttöympäristön mukaan taas edellyttää sovellusohjelmia.

1) Kääntäjät ja tulkit

Kielen kääntäjät (translators, compilers)/tulkit (interpreters) kääntävät/tulkitsevat korkean tason kielellä kirjoitettuja ohjelmia käytettävän tietokoneen konekielelle. Tulkkia käytetään myös tulkittaessa konekieltä koneen ymmärtämään muotoon eli mikrokoodiksi.

2) Käyttöjärjestelmät

Käyttöjärjestelmän (operating system) tärkeimpiä tehtäviä ovat:

- tiedonsiirron hallinta tietokonelaitteiston eri osien (syöttö- ja tulostuslaitteet, ulkoiset muistilaitteet) välillä (ts. monen laitteen samanaikainen hallinta)
- ohjelmien suorituksen ohjaus (ts. monen ohjelman samanaikainen hallinta)
- tietokoneen resurssien jako käyttäjille (ts. monen käyttäjän samanaikainen hallinta)
- tietojen pitkäaikaissäilytyksestä huolehtiminen (toissijaisen muistin ylläpito)

Muita systeemiohjelmia, joita ei käsitellä tässä yhteydessä tarkemmin, ovat:

3) Linkittäjät

Linkittäjien (linkers) tehtävänä on koota erillään käännetyt ohjelmamoduulit ja niiden käyttämät systeemin apuohjelmat yhdeksi suorituskelpoiseksi kokonaisuudeksi (ns. latausmoduuliksi), joka yleensä tallennetaan oheismuistiin.

4) Lataajat

Lataajia (loaders) käytetään siirtämään ohjelmia toissijaisesta muistista (esim. levyltä) keskusmuistiin.

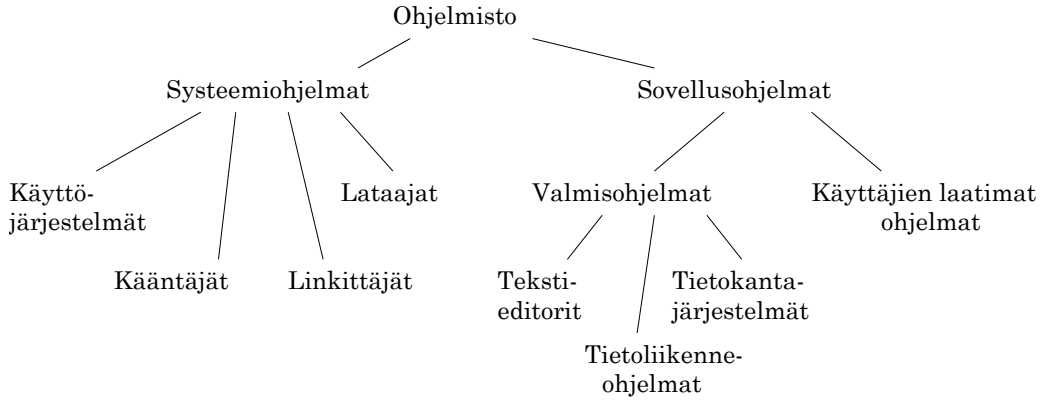
5) Editorit

Editoreita (editors) käytetään ohjelmien (tai muiden tekstien) kirjoittamiseen ja korjailemiseen, ja niitä ovat mm. muotoiluohjelmat ja tekstinkäsittelyohjelmat.

6) Tietoliikenneohjelmisto

Tietoliikenneohjelmiston (communications software) tehtävänä on mm. toisilta tietokoneilta saapuvan tiedon vastaanottaminen ja lähettäminen, virheiden havaitseminen tiedonsiirrossa ja reititys.

Seuraavassa kuvassa on esitetty ohjelmiston karkea luokittelu. On huomattava, että ero systeemi- ja valmisohjelmien välillä ei ole kovin selvä.



5.1 Ohjelmointikielten kääntäminen

Konekielen käyttö mahdollistaa ohjelmoinnin abstraktilla koneella, tietokoneen fyysistä arkkitehtuuria suuremmin ajattelematta. Ohjelmointi konekäskyilläkin (koneenläheisillä kielillä) on kuitenkin hidasta ja virhealtista. Ohjelmoinnin helpottamiseksi on kehitetty satoja korkean tason ohjelmointikieliä (high level programming languages).

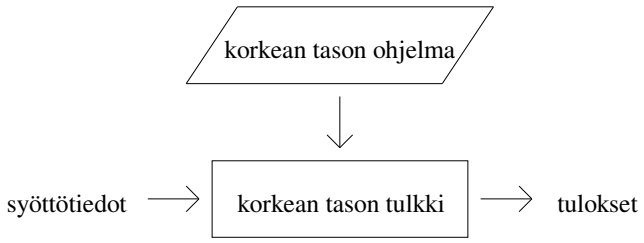
Korkean tason ohjelmointikielissä on luonnolliset ohjaus- ja tietorakenteet. Konekielissä ohjausrakenteet toteutetaan primitiivisillä operaatioilla (esim. JUMP-käsky) ja tietorakenteet primitiivisten, osoitteisiin kohdistettavien, operaatioiden avulla. Niinpä algoritmien suunnittelu on helpompaa ja ohjelmakoodi luettavampaa/ymmärrettävämpää korkean tason kielellä ohjelmoitaessa. Korkean tason ohjelmointikielellä kirjoitetut ohjelmat ovat luotettavampia (virheet löytyvät helposti). Konekielellä ohjelmointi on virhealtista, ja virheet ovat vaikeasti löydettävissä. Korkean tason ohjelmointikielellä kirjoitetut ohjelmat ovat myös siirrettävämpiä (portable) koneesta toiseen, koska ne ovat koneesta riippumattomia. Konekielet ovat puolestaan konekohtaisia.

Kone ymmärtää mikrokäskyjä ja mikrotulkittuja konekäskyjä suoraan, mutta korkean tason kielellä kirjoitetut ohjelmat on käännettävä koneen ymmärtämään muotoon. Koneenläheinen ohjelma on nopeampi suorittaa ja vaatii vähemmän muistia kuin vastaava korkean tason kielellä kirjoitettu ohjelma, joka on käännetty koneen ymmärtämään muotoon. Koneenläheisellä kielellä ohjelmoitaessa konetta (rekistereitä, muistia yms.) pystytään käyttämään paremmin hyväksi kuin korkean tason kielillä ohjelmoitaessa (etua esim. käyttöjärjestelmän ohjelmoinnissa). Vaikka koneenläheisellä ohjelmoinnilla on omat etunsa, aina kun on mahdollista, tulisi käyttää korkean tason ohjelmointia.

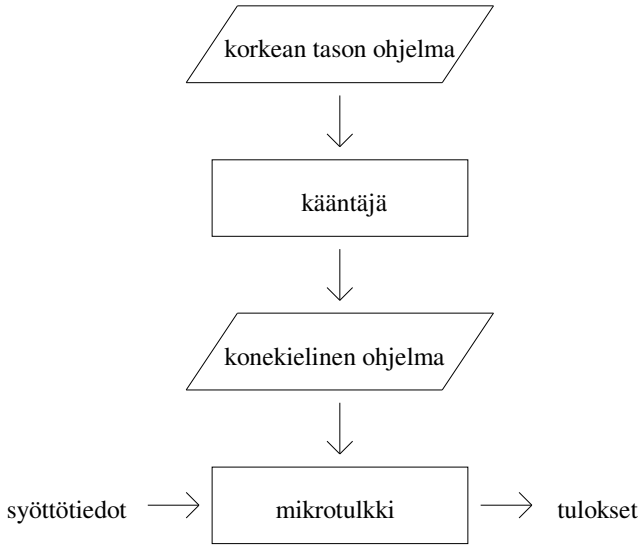
Ohjelmointikielten kääntämisellä tarkoitetaan korkean tason ohjelmointikielellä kirjoitetun ohjelman kääntämistä koneen ymmärtämään muotoon, konekieliseksi ohjelmaksi, joka voidaan suorittaa tietokoneessa suoraan (mikro-ohjelmoidulla tulkilla). Korkean

tason ohjelmointikielellä kirjoitetuista ohjelmista käännetään käskyt ja tietorakenteet. Jokainen korkean tason kielellä kirjoitettu käsky käännetään yhdeksi tai useammaksi konekieliseksi käskyksi. Tietorakenteet esitetään konekielissä bitteinä, numeroina ja osoitteina. Esimerkiksi peräkkäinen vektori esitetään muistissa tallentamalla sen alkiot peräkkäisiin muistipaikkoihin, ja vektorin alkioon viittaus käännetään viittaukseksi muistipaikkaan.

Korkean tason ohjelmointikielellä kirjoitetun ohjelman muuntamiseksi koneen ymmärtämään muotoon on kaksi tapaa: tulkitseminen (interpretation), jonka suorittaa tulkki (interpreter) ja kääntäminen (compilation), jonka suorittaa kääntäjä (compiler). Tapojen erona on oikeastaan vain taso, jolla tulkinta tapahtuu:



Tulkitseva suoritus



Kääntävä suoritus

5.1.1 Tulkitseminen

Tulkitsemisessa korkean tason kielellä kirjoitettua ohjelmaa suoritetaan lause kerrallaan. Lauseen suoritus ei edellytä sen kääntämistä, vaan selvitettyään lauseen tyypin, oikeellisuuden ja operandit, tulkki suorittaa lauseen vaatimat toimenpiteet. On muistettava, että tulkki on itsessään konekielinen ohjelma, joka sisältää ikäänkuin valmiina kaikkien lähdekielen lausetyyppien toteutuksen. Tulkin tulee vain osata haarautua kulloisenkin lähdekielen lauseen mukaisesti oikeaan kohtaan omassa konekielisessä koodissaan. Samaa ideaa käytettiin edellä alemmalla tasolla, laadittaessa konekielen mikrotulkkia.

Voidaan myös rakentaa tulkkeja, jotka ennen lauseen suoritusta tekevät sille jonkinasteisen muunnoksen/käännöksen, mutta tämä askel on yleensä ohjelmoijalta piilossa.

Miksi ei suoraan tulkita korkean tason ohjelmointikielellä kirjoitettua ohjelmaa mikro-ohjelmoidulla tulkilla? Sellaista on vaikea kirjoittaa korkean tason kielelle, koska tällaisen kielen syntaksi ja semantiikka on monimutkaisempi kuin konekielen. Niinpä tulkitseminen kannattaa tehdä vaiheittain yksinkertaistaen tulkittavia käskyjä. Kaksivaiheisessa tulkinnassa eri kielille riittää kirjoittaa vain konekielinen tulkki (1. vaihe), koska tulkki konekielestä mikrokoodiin (2. vaihe) säilyy aina samana. Mikro-ohjelmoitu tulkki korkean tason kielelle vaatisi myös paljon kallista tilaa mikro-ohjelmamuistista.

5.1.2 Kääntäminen

Kääntämisessä korkean tason kielellä kirjoitettu ohjelma käännetään ensin kokonaan konekielelle. Kääntämisen suorittaa kääntäjäohjelma, ja sen tulos on konekielinen ohjelma, joka voidaan antaa mikrotulkille suoritettavaksi. Alkuperäistä korkean tason kielellä kirjoitettua ohjelmaa nimitetään lähdekieliseksi ohjelmaksi (source program) ja käännöksen tuloksena syntynyttä konekielistä ohjelmaa objektiohjelmaksi (object program). Objektiohjelman linkitys muiden tarvittavien ohjelmien kanssa suoritetaan usein käännöksen yhteydessä. Linkityksen tuloksena olevaa latausmoduulia säilytetään toissijaisessa muistissa, josta se voidaan haluttaessa suorittaa. Tällöin lataajaohjelma hakee moduulin keskusmuistiin suoritettavaksi.

Käännetty ohjelma toimii nopeammin kuin tulkittava ohjelma. Kerran käännetty ohjelma saadaan nopeasti suoritukseen useammilla ajokerroilla, tulkittava ohjelma vaatii aina saman ajan (tulkinta + suoritus). Jos virheetöntä ohjelmaa ei muuteta, se kannattaakin useimmiten kääntää suorituskelpoiseksi.

Virheiden löytäminen tulkitsemalla ohjelmaa on helpompaa, koska tulkki käsittelee lähdekielistä ohjelmaa. Tulkitseminen soveltuukin erityisesti ohjelman kehitystyövaiheeseen ja testaukseen, koska työskentely on vuorovaikutteisempaa. Tulkki on helpompi kirjoittaa ja vaatii vähemmän muistia kuin kääntäjä.

5.2 Syntaksin määrittely

Ohjelma on johonkin merkkivalikoimaan kuuluvien merkkien jono. Kaikki merkkijonot eivät tietenkään ole ohjelmia. Sääntöjä, jotka kertovat, minkälaiset merkkijonot ovat määrätyllä ohjelmointikielellä kirjoitettuja ohjelmia, sanotaan syntaktisiksi säännöiksi ja kaikkien tarvittavien syntaktisten sääntöjen kokoelmaa kyseisen kielen ***syntaksiksi*** (syntax).

5.2.1 Kieliopit

Jotta korkean tason ohjelmointikieltä voidaan tulkita/kääntää, on ensin määriteltävä kielen syntaksi. Syntaksin määrittelyyn käytetään ***kielioppia*** (grammar). Kieliopilla tarkoitetaan kielioppisääntöjä, jotka määräävät, miten kielen symboleja saa laillisesti käyttää. Ohjelmointikielten kieliopeilla on sama tarkoitus kuin luonnollisen kielen kieliopilla, mutta ohjelmointikielten kieliopit ovat täsmällisempiä ja yksinkertaisempia.

Kieliopin tarkoituksena on määritellä, mitkä kirjoitelmat ovat käytettävällä ohjelmointikielellä oikein muodostettuja ohjelmia eli kyseisen ohjelmointikielen kieliopin generoimia. Generointi tarkoittaa sitä, että ohjelma muodostetaan eli johdetaan käyttäen kieliopin sääntöjä eli produktioita.

Ohjelmointikielten syntaksi kuvataan yleensä ns. ***kontekstittomalla kieliopilla*** (context-free grammar), jota kutsutaan myös BNF-kieliopiksi (Backus-Naur Form grammar). Kontekstiton (eli yhteysvapaa) -termi tulee siitä kieliopin ominaisuudesta, että kielen komponenttien rakennevaihtoehdot eivät riipu ympäristöstään. Kontekstiton kielioppi koostuu seuraavista osista:

- ***Päätesymbolit*** eli ***terminaalit***
- ***Välisymbolit*** eli ***nonterminaalit***
- ***Alkusymboli***: eräs nonterminaali
- ***Produktiosäännöt***

Päätesymbolit ovat kielen perussymboleja, ohjelmatekstissä esiintyviä symboleja (esim. IF, THEN, =, +, 1, 2, 3, ...), jotka voivat muodostaa ehtoja, lauseita, moduuleja jne. Välisymbolit määräävät syntaksi- eli symboliluokat, jotka muodostuvat pääte- ja välisymboleista. Kontekstittoman kieliopin produktiot ovat muotoa A w, missä A on välisymboli ja w on nollasta tai useammasta väli- ja/tai päätesymbolista muodostettu symbolijono. Päätesymbolit eivät koskaan ole kontekstittoman kieliopin säännöissä vasemmalla puolella. Säännöt määräävät, miten syntaksiluokka (jokin välisymboli) voidaan muodostaa muista väli- tai päätesymboleista. Alkusymboli on eräs nonterminaali, josta jokainen generointi alkaa.

Ohjelma (tai luonnollisen kielen lause) on oikein muodostettu, jos se voidaan johtaa alkusymbolista S. Alkusymboli on kielen jonkin säännön vasempana puolena, eli kieliopista löytyy sääntö S $\rightarrow$ w, josta kaikki kieliopin mukaiset ohjelmat (luonnollisen kielen lauseet) ovat johdettavissa. Johtaminen tapahtuu siten, että välisymboli S lavennetaan oikean puolen symbolijonoksi, jossa esiintyvät välisymbolit lavennetaan sopivilla säännöillä uusiksi symbolijonoiksi, kunnes symbolijono muodostuu pelkistä päätesymboleista. Kaikkien välisymbolien tulee laventua jossakin vaiheessa pääte-symboleiksi, eli kieliopissa on kullekin välisymbolille määriteltävä sellainen sääntö, että välisymbolit voivat laventua päätesymboleiksi.

Jokainen syntaktisesti kelvollinen (kieliopin mukainen) päätesymbolijono voidaan esittää ***johto- eli jäsennyspuuna*** (parse tree). Päätesymbolit ovat lehtisolmuissa, ja haarautumasolmut esittävät syntaksiluokkia (välisymboleja), joista päätesymbolit johdetaan. Puun juuressa on syntaksiluokka (välisymboli), johon koko päätesymbolijono kuuluu (eli josta koko päätesymbolijono on johdettavissa). Puussa kuvataan pääte-symbolijonon muodostuminen (johtuminen) syntaksiluokistaan. Puu syntyy, kun vedetään viiva vasemman puolen välisymbolista jokaiseen oikean puolen symboliin (voivat olla sekä pääte- että välisymboleja). Jotta välisymbolit erottuisivat selvästi päätesymboleista, kirjoitetaan välisymbolit kulmasulkeisiin. Ellei alkusymbolia ole muuten merkitty, se on kieliopin ensimmäisen produktiosäännön vasemman puolen välisymboli.

Esimerkki. Kontekstiton kielioppi.

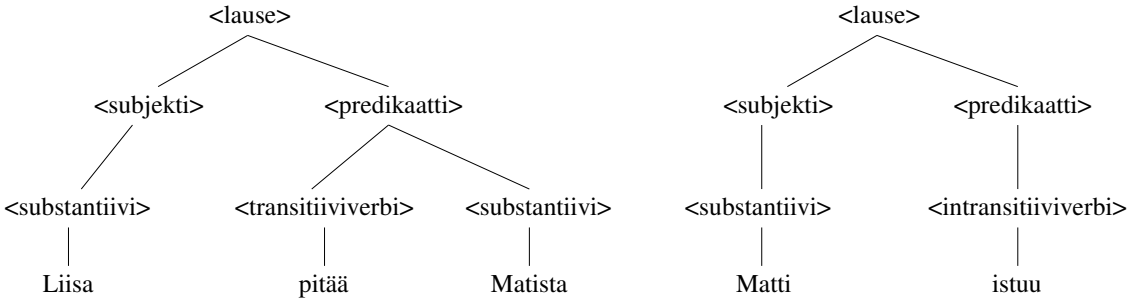
Nonterminaalit: <lause>, <subjekti>, <predikaatti>, <substantiivi>, <intransitiiviverbi>, <transitiiviverbi>

Terminaalit: Matti, Liisa, Matista, istuu, pitää

Produktiot:

<lause> → <subjekti><predikaatti>
<subjekti> → <substantiivi>
<predikaatti> → <intransitiiviverbi>
<predikaatti> → <transitiiviverbi> <substantiivi>
<substantiivi> → Matti
<substantiivi> → Liisa
<substantiivi> → Matista
<intransitiiviverbi> → istuu
<transitiiviverbi> → pitää

Johto- eli jäsennyspuita:



Esimerkin kielioppi ei ole oikein hyvä, sillä sen mukaan voitaisiin johtaa myös lause "Matista istuu", joka ei ole suomea. Luonnolliselle kielelle onkin vaikea kirjoittaa hyvä kielioppi – eikä tarkkaan ottaen kontekstiton kielioppi riitäkään luonnollisen kielen lauseiden kuvaamiseen.

Produktiosäännön vasemmalla puolella on aina nonterminaali, ja oikealla puolella voi olla nonterminaaleja ja/tai terminaaleja. Ohjelmointikieliä kuvattaessa on produktiosäännöissä tapana käyttää seuraavia lyhennysmerkintöjä:

$N \rightarrow w_1 \mid w_2 \mid ... \mid w_k$ tarkoittaa itse asiassa sääntöjä $N \rightarrow w_1$, $N \rightarrow w_2$, ... ja $N \rightarrow w_k$ eli N voidaan korvata millä tahansa pystyviivojen erottamista jonoista.

$N \rightarrow \{w_1, ..., w_k\}$ tarkoittaa samaa kuin $N \rightarrow \varepsilon$, $N \rightarrow w_1$ N, ..., $N \rightarrow w_k$ N, missä ε tarkoittaa tyhjää jonoa. N voidaan siis korvata rekursiivisesti mielivaltaisen monella w_i:llä tai tyhjällä jonolla.

$N \rightarrow [w_1, ..., w_k]$ tarkoittaa samaa kuin $N \rightarrow \varepsilon$, $N \rightarrow w_1$, ..., $N \rightarrow w_k$, missä ε tarkoittaa tyhjää jonoa. N voidaan siis korvata jollakin w_i:llä tai tyhjällä jonolla.

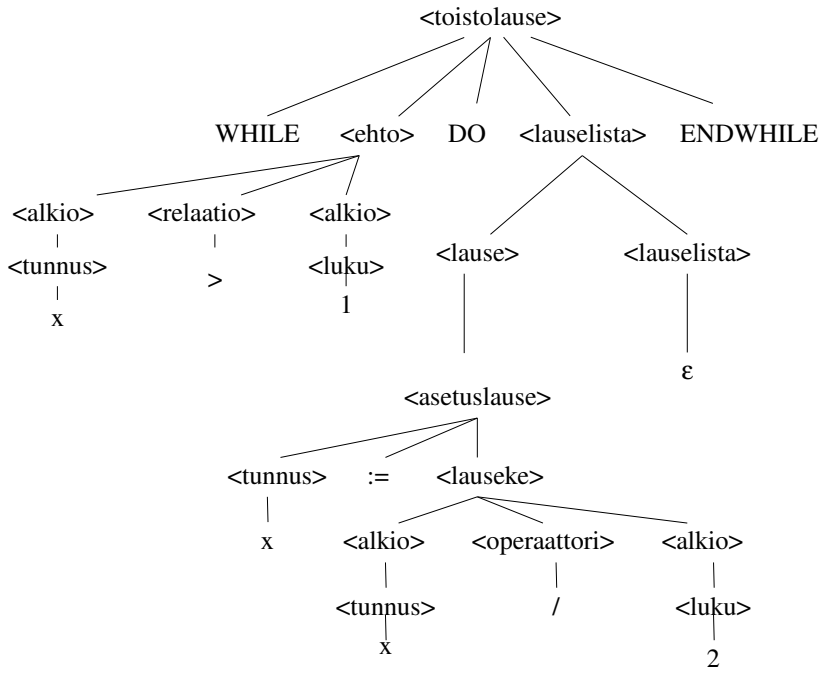
Esimerkki. Seuraavassa on kielioppi luvun 2 pseudokielelle (epätäydellinen ja yksinkertaistettu).

<ohjelma>	<moduuli> {<moduuli>}
<moduuli>	MODULE <tunnus> (<parametrilista>) [RETURNS <tyyppi>] {<lauselista>}ENDMODULE
<tyyppi>	INTEGER | REAL | BOOLEAN | CHAR | LIST
<parametrilista>	<tyyppi> <tunnus> <parametrilista> | ε
<lauselista>	<lause> <lauselista> | ε
<lause>	<ehtolause> | <valintalause> | <toistolause> | <asetuslause> | <palautuslause>
<ehtolause>	IF <ehto> THEN <lauselista> [ELSE <lauselista>] ENDIF
<valintalause>	CASE <tunnus> OF <alkio> : <lauselista>{<alkio> : <lauselista>} [OTHERS: <lauselista>] ENDCASE

<toistolause>	WHILE <ehto> DO <lauselista> ENDWHILE
	| REPEAT <lauselista> UNTIL <ehto>
	| FOR <joukko> DO <lauselista> ENDFOR
<asetuslause>	<tunnus> := <lauseke>
<palautuslause>	RETURN <lauseke>
<ehto>	<alkio> <relaatio> <alkio>
<relaatio>	= | >= | > | < | <= | <>
<lauseke>	<alkio> <operaattori> <alkio> | <alkio>
<operaattori>	+ | – | * | /
<joukko>	<tunnus> := <alkio>, ..., <alkio>
<alkio>	<tunnus> | <luku>

Välisymbolien <tunnus> ja <luku> määrittely jää harjoitustehtäväksi.

Esimerkki. Lauseen WHILE x>1 DO x := x/2 ENDWHILE jäsennyspuu**:**



5.3 Kääntäjän toiminta

Kääntäjän tehtävänä on muuntaa korkean tason kielellä kirjoitettu ohjelma tietokoneen ymmärtämään muotoon eli vastaavaksi konekieliseksi ohjelmaksi. Kääntäjä selaa ja jäsentää lähdekielisen ohjelman ja jäsennyksestä saatua tietoa hyväksi käyttäen muodostaa konekielisen ohjelman, joka voidaan suorittaa tietokoneessa. Kääntäjän toiminta voidaan jakaa kolmeen vaiheeseen:

1) Leksikaalinen analyysi (lexical analysis) eli *selaaminen*

Selaamisessa lähdekielisen ohjelman muodostavat merkkijonot jaetaan yhden tai useamman merkin muodostamiksi loogisesti yhteenkuuluviksi, erillisiksi symboleiksi, tekstialkioiksi (tokens), kuten tunnuksiksi (esimerkiksi x, y, syt), operaattoreiksi (+, –, *, /, :=) ja kieleen sisältyviksi varatuiksi sanoiksi (IF, THEN, WHILE, DO). Nämä symbolit kuvaavat lähdekielisen ohjelman päätesymboleja. Luonnollisessa kielessä selaamisessa erotellaan päätesymboleina esimerkiksi sanat, välimerkit ja numerot.

Selaaja (scanner, lexical analyzer) on siis ohjelmointikielen alkiorakenteen tunnistaja, ja selauksen tulos on selattujen tekstialkioiden lista.

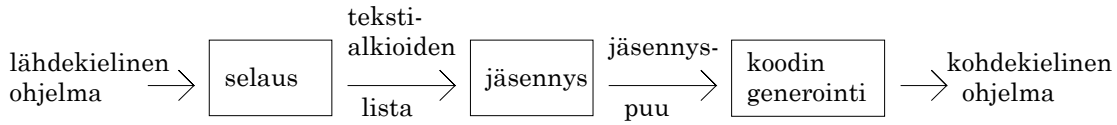
2) Syntaktinen analyysi (syntax analysis) eli *jäsentäminen*

Jäsentämisessä selvitetään, miten lähdekielinen ohjelma on johdettu kielioppisääntöjen mukaan, eli määritetään ohjelman syntaktinen rakenne. Tämä vaihe vastaa luonnollisen kielen lauseenjäsentämistä. Jäsennyksen tulos voidaan esittää johto- eli jäsennyspuuna.

3) *Koodin generointi*

'Koodin' generoinnissa muodostetaan ohjelman toteuttavat konekieliset käskyt eli objektikoodi ja varataan tilaa ohjelmalle ja siinä esiintyville muuttujille. Koodin generoinnin yhteydessä tai sen jälkeen voidaan suorittaa optimointia, jonka tarkoituksena on pienentää (tilan säästö) ja nopeuttaa (ajan säästö) objektikoodia.

Kääntäminen voidaan suorittaa monivaiheisesti siten, että edellisen vaiheen tulokset annetaan syötteenä seuraavalle vaiheelle, tai yksivaiheisena, jolloin välituloksia ei tarvitse muodostaa (koko ohjelmalle) erikseen. Käännöksen vaiheita ei tarvitse suorittaa täysin peräkkäisesti, vaan ne voidaan suorittaa osittain samanaikaisestikin.



5.3.1 Selaaminen

Selaamisen tarkoituksena on tunnistaa, mitkä merkit kuuluvat yhteen symboliin eli tekstialkioon (tokeniin) ja selvittää, minkätyyppinen kyseinen tekstialkio on. Jokainen tekstialkio muodostaa kieliopin päätesymbolin.

Tekstialkion tyyppejä:

- varattu sana: MODULE, IF, THEN, ELSE, CASE, OF, FOR, DO, REPEAT, UNTIL, WHILE, RETURN, RETURNS, ENDIF, ENDFOR, ENDWHILE, ENDCASE, ENDMODULE
- operaattori: =: , = , <> , < , > , >=, <= + , – , * , / (pilkku on tässä vain erotin)
- välimerkki: sulkeet, välilyönti, rivinvaihto, pilkku, puolipiste, kaksoispiste, piste
- nimi eli tunnus: muuttuja, nimetty vakio, moduulin tunnus
- luku- tai merkkivakio: –354.786, 3, "A"

Selaaminen on yleensä helppo tehtävä. Yleensä peräkkäiset kirjaimet kuuluvat yhteen symboliin, ja peräkkäisiä symboleja erottaa välimerkki. Tarpeettomat välimerkit (välilyönnit, rivinvaihdot, tabuloinnit yms.) poistetaan. Jos ohjelmatekstistä löytyy ohjelmointikielen merkkivalikoimaan kuulumattomia merkkejä, annetaan virheilmoitus.

Selaamisen tuloksena käännettävän ohjelman symbolit luokitellaan eri luokkiin tai tyyppeihin. Ohjelmassa esiintyviin tunnuksiin liittyy jokin arvo. Jo selausvaiheessa tunnus viedään luetteloon, jota nimitetään ***symbolitauluksi***. Siihen tallennetaan kunkin tunnuksen nimi ja tyyppi.

Esimerkki. Selaus. Tarkastellaan ohjelman osaa IF x < 5 THEN x := x + 1

Symbolit:	IF	x	<	5	THEN	x	:=	x	+	1
Tyypit:	varattu	tunnus	oper.	lukuvakio	varattu	tunnus	oper.	tunnus	oper.	lukuvakio

5.3.2 Jäsentäminen

Selaamisen jälkeen ohjelma on tekstialkioiden muodostama lista. Jäsentämisessä selvitetään, millä kielioppisäännöillä ohjelma on tuotettu, eli selvitetään ohjelman syntaktinen rakenne. Minkä tahansa kelvollisen päätesymbolijonon syntaktinen rakenne voidaan esittää jäsennyspuussa, jossa päätesymbolit ovat lehtisolmuissa, ja syntaksiluokka, johon päätesymbolijono kuuluu, puun juuressa. Jäsennyspuun muodostamiseen on olemassa kaksi päästrategiaa: osittava (top-down) jäsennys eli juuresta lehtiin -menetelmä ja kokoava (bottom-up) jäsennys eli lehdistä juureen -menetelmä.

Osittavassa jäsennyksessä ohjelman jäsennyspuu muodostetaan juuresta lähtien edeten lehtiin päin (ylhäältä alas). Osittava jäsentäjä on helppo johtaa kieliopin säännöistä. Jäsentäjä lähtee liikkeelle lähtösymbolista (juuresta) osittaen tarkasteltavan päätesymbolijonon (lähdekielisen ohjelman tekstialkiot) yksinkertaisempiin syntaksiluokkiin, kunnes tarkasteltava päätesymbolijono voidaan esittää puun lehdissä.

Kokoavassa jäsennyksessä ohjelman jäsennyspuu muodostetaan lehdistä lähtien edeten juureen päin (alhaalta ylös). Jäsentäjä yrittää yhdistellä lähdekielisen ohjelman vierekkäisiä tekstialkioita yksinkertaisten syntaksiluokkien elementeiksi, jotka sitten yhdistellään monimutkaisempien syntaksiluokkien elementeiksi jne., kunnes kaikki syntaksiluokat on saatu koottua yhdeksi syntaksiluokaksi. Jokaisessa yhdistelyvaiheessa saattaa olla useampia yhdistelymahdollisuuksia, joten jäsentäjä joutuu yrittämään eri vaihtoehtoja yhdistelyssä ja joutuu mahdollisesti peruuttamaan epäonnistuessaan jatkossa.

Jäsennyksen ongelmana on kieliopin epädeterminstisyys: kirjoitelman johtamisessa ja siis myös jäsennyksessä on kussakin tilanteessa usein mahdollista soveltaa useampaa kuin yhtä sääntöä. Jäsennysvirheen havaittuaan jäsentäjä antaa virheilmoituksen.

Osittava jäsentäminen

Jos ohjelma P (jono tekstialkioita) on oikein muodostettu, se voidaan johtaa alkusymbolista S. Arvataan, että

$S \rightarrow s_1 \ldots s_m$ (s_i:t ovat nonterminaaleja tai terminaaleja)

on johdon ensimmäinen produktio. Silloin ohjelmateksti P pitää voida jakaa m:ään osajonoon $p_1 \ldots p_m$, jotka vastaavat järjestyksessä symboleja $s_1 \ldots s_m$. Nyt on kaksi mahdollisuutta: s_i on terminaali tai nonterminaali. Tarkastellaan ensin tapausta: s_i on terminaali. Jos $p_i = s_i$, on jäsennys sen osalta valmis. Jos taas $p_i \neq s_i$, niin voidaan yrittää uutta P:n jakoa. Jos mikään P:n jako ei onnistu, peräännytään kokeilemaan seuraavaa produktiota $S \rightarrow \ldots$. Jos taas s_i on nonterminaali, niin jäsennystä sovelletaan rekursiivisesti osajonoon p_i kokeilemalla s_i:lle määriteltyjä produktioita. Jos kaikki rekursiiviset alijäsennykset onnistuvat, arvaus oli oikea ja koko jäsennys onnistuu. Jos jossain vaiheessa kokeiltu produktio havaitaan vääräksi, se hylätään ja kokeillaan seuraavaa vaihtoehtoa. Jos mikään vaihtoehto ei johda tulokseen, peräännytään rekursiossa ylemmälle tasolle. Jos perääntyminen ei enää anna uusia vaihtoehtoja,

ohjelman jäsennys ei onnistu, ts. ohjelma ei ole oikein muodostettu. Alla oleva jäsennyksen suorittava algoritmi on siinä mielessä epädeterministinen, että produktioiden kokeilujärjestystä ei ole kiinnitetty. Yleensä produktioita kokeillaan niiden esiintymisjärjestyksessä tarkasteltavassa kieliopissa.

```
MODULE jäsennys(nonterminaali N, terminaalijono w, johtopuu P) RETURNS onnistuu / ei
    REPEAT
        Kokeile produktiota N → u
        IF u = w THEN   (* u oikea terminaali, jäsennys onnistuu suoraan *)
            P := N(w)
            RETURN onnistuu
        ELSE                    (* u:ssa yksi tai useampia nonteminaaleja *)
            Olkoon u muotoa u0 N1 u1 ... Nk uk
                    (* Ni:t ovat nonterminaaleja ja ui:t terminaaleja, ehkä tyhjiä *)
            WHILE jokin w:n jako terminaaleilla u0, u1, ..., uk on kokeilematta DO
                Olkoon w:n uusi jako muotoa: w = u0 w1 u1 ... wk uk
                IF jokainen jäsennys(Ni, wi, Pi) onnistuu  (* rekursiiviinen kutsu ! *) THEN
                    P := N(u0, P1, u1, ..., Pk, uk)
                    RETURN onnistuu
                ENDIF
            ENDWHILE
        ENDIF
    UNTIL kaikki produktiot N → ... on kokeiltu
    RETURN ei onnistu
ENDMODULE
```

Esimerkki. Olkoon määritelty kielioppi:

(1) <lauseke> → <lauseke> + <lauseke>
(2) <lauseke> → <lauseke> – <lauseke>
(3) <lauseke> → luku

Onko lauseke

luku – luku + luku

kieliopin mukainen? On siis ratkaistava

(4) jäsennys(<lauseke>, luku – luku + luku, P)

Kokeillaan ensin produktiota (1): $N \rightarrow u = u_0\ N_1\ u_1\ N_2\ u_2$ eli $u_0 \rightarrow \varepsilon$, $N_1 \rightarrow w_1$, $u_1 \rightarrow +$, $N_2 \rightarrow w_2$ ja $u_2 \rightarrow \varepsilon$. Vastaava lausekkeen jako on $w = w_1 + w_2$, missä w_1 = luku – luku, w_2 = luku. Tämä johtaa rekursiivisiin kutsuihin:

(5) jäsennys(<lauseke>, luku – luku, P_1) ja
(6) jäsennys(<lauseke>, luku, P_2)

Nyt (6) ratkeaa helposti, sillä (1) sen paremmin kuin (2) ei onnistu, mutta (3) onnistuu suoraan, jolloin

P_2 := <lauseke>(luku)

Jäsennyksessä (5) oikea produktio on (2) ja oikea jako $w = w_1 – w_2$, missä $w_1 = w_2$ = luku. Se johtaa kutsuihin:

(7) jäsennys(<lauseke>, luku, P_3) ja
(8) jäsennys(<lauseke>, luku, P_4)

Ilmeisesti (7) ja (8) ratkeavat suoraan:

P_3 := <lauseke>(luku) ja
P_4 := <lauseke>(luku)

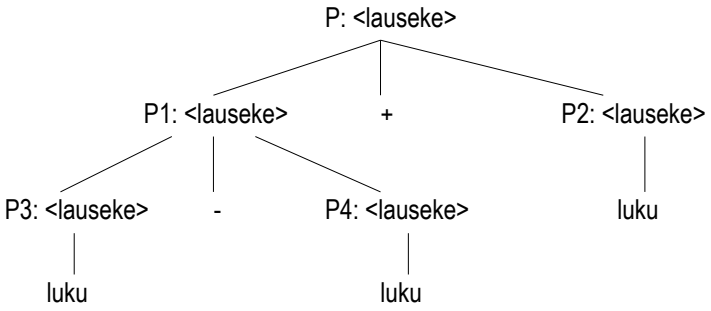
Mutta tällöin myös (5) onnistuu:

P_1 := <lauseke> (P_3, –, P_4)
= <lauseke> (<lauseke> (luku),
–,
<lauseke> (luku))

Näin myös (4) ratkeaa, ja lopullinen tulos on:

P := <lauseke>(P_1, +, P_2)
= <lauseke> (<lauseke> (<lauseke> (luku),
–,
<lauseke> (luku))
+,
<lauseke>(luku))

eli



5.3.3 Koodin generointi

Leksikaalisen ja syntaktisen analyysin jälkeen tuotetaan käännetty ohjelma, eli generoidaan lähdekielistä ohjelmaa vastaava konekielinen ohjelma.

Koodin generointi jakautuu kolmeen osatehtävään:

- muistin varaaminen (memory allocation) ohjelman käyttämälle tiedolle
- lähdekielistä ohjelmaa vastaavien konekäskyjen muodostaminen
- koodin optimointi

1) Muistin varaaminen

Symbolitaulun perusteella lasketaan, montako muistipaikkaa tunnuksiin liittyvät arvot vaativat, ja sen mukaan varataan koneen muistista riittävän suuri alue tunnuksille. Erityyppiset tunnukset tarvitsevat eri määrän muistia arvojensa tallennusta varten (esim. kokonaisluvut, merkkijonot, merkit). Tässä yhteydessä kuhunkin tunnukseen liitetään muistiosoite.

Esimerkki. Ohjelmasta

```
MODULE Pythagoras(INTEGER x INTEGER y INTEGER z) RETURNS BOOLEAN
        RETURN x^2 + y^2 = z^2
ENDMODULE
```

löytyvät muuttujanimet x, y ja z, jotka viedään symbolitauluun. Symbolitaulu voisi näyttää seuraavalta:

tunnus	tyyppi	osoite
Pythagoras	BOOLEAN	9
x	INTEGER	10
y	INTEGER	14
z	INTEGER	18

Jos käytettävän koneen luvut ovat nelitavuisia (eli 32-bittisiä), totuusarvot yksitavuisia, ja muistialue osoitteesta 10 eteenpäin on käytettävissä muuttujien tallennukseen, tallennetaan x, y ja z muistiosoitteista 10, 14 ja 18 lähtien.

2) Konekäskyjen muodostaminen (varsinainen koodin generointi)

Kun ohjelma on jäsennetty, sen rakenne tunnetaan, joten vastaava konekielinen koodi voidaan generoida. Yleensä generoidaan ns. symbolista konekieltä, jossa muisti-

paikkoihin viitataan todellisten osoitteiden asemesta tunnuksin. Nämä tunnukset korvataan ohjelman latausvaiheessa todellisilla osoitteilla. Koodin generointi noudattaa jäsennyspuun rakennetta:

```
MODULE Generoi(jäsennyspuu P)
    Olkoon P = N(P1, ..., Pk)
    CASE N OF
        <moduuli> :          generoimoduuli(P1, ..., Pk)
        <asetuslause> :      generoiasetuslause(P1, ..., Pk)
        <joslause> :         generoijoslause(P1, ..., Pk)
        <josmuutoin> :       generoijosmuutoin(P1, ..., Pk)
        <whilelause> :       generoiwhilelause(P1, ..., Pk)
        .....
    ENDCASE
ENDMODULE
```

Kieliopin tärkeimmille välisymboleille kirjoitetaan koodin generointiin omat moduulinsa. Kirjoitetaan koodin generoiva moduuli asetuslauseelle x := y o z, missä o on jokin operaattori. Kieliopissa on säännöt:

```
<asetuslause>    →    <tunnus> := <lauseke>
<lauseke>        →    <tunnus> <operaattori> <tunnus>
<operaattori>    →    + | – | * | /
<tunnus>         →    <kirjain> | <kirjain> <tunnus>
<kirjain>        →    a | b | ... | z
```

Kun asetuslause x := y o z on jäsennetty tämän kieliopin mukaisesti, tulos on muotoa

<asetuslause> (<tunnus> (x), :=, <lauseke> (<tunnus> (y), <operaattori> (o), <tunnus> (z)))

Kääntämisen kannalta tässä on tarpeetontakin informaatiota. Sama tieto on puussa

<asetuslause>(x, y, o, z)

Kirjoitetaan käännösmoduuli tämän mukaisesti.

```
MODULE generoiasetuslause(x, y, o, z)
    Tulosta LOAD y
    CASE o OF
        '+'  :  Tulosta ADD z
        '–'  :  Tulosta SUBTRACT z
        '*'  :  Tulosta MULTIPLY z
        '/'  :  Tulosta DIVIDE z
    ENDCASE
    Tulosta STORE x
ENDMODULE
```

Itse asiassa x, y ja z ovat symbolitaulusta saatavia osoitteita. Vastaavasti generoidaan koodi muita rakenteita varten. Jos esimerkiksi ehtolausekkeen syntaksi on

```
<oper>    > | < | = | ≠
<ehto>    <tunnus> <oper> <tunnus>
```

niin IF…THEN…ELSE-rakenteen jäsennyspuu

<josmuutoin> (IF, <ehto> (...), THEN, <lause> (...), ELSE, <lause> (...))

kääntyy seuraavasti:

```
MODULE generoijosmuutoin (ehto-osa E, niin-osa P, muutoin-osa Q)
        Olkoon E = <ehto> (<tunnus> (x), <oper> (o), <tunnus> (y))
        Määrätään yksikäsitteiset nimiöt (eli tunnukset) a ja b.
        generoiehto (x, o, y, a)
        (* saa aikaan hypyn kohtaan a konekielisessä ohjelmassa, jos ehto x o y on epätosi *)
        generoi(P)
        tulosta                 JUMP b
                        a:      NOP
        generoi(Q)
        tulosta         b:      NOP
ENDMODULE

MODULE generoiehto(x, o, y, a)              (* a on epätosi-ehtoa vastaava hyppynimiö *)
        CASE o OF
                '>' , '>=' :    tulosta     LOAD   x
                                            SUBTRACT    y
                                            JUMPNEG     a
                                IF o = '>' THEN tulosta   JUMPZERO   a   ENDIF
                '<' , '<=' :    tulosta     LOAD   y
                                            SUBTRACT    x
                                            JUMPNEG     a
                                IF o = '<' THEN tulosta   JUMPZERO   a   ENDIF
                '=' :           tulosta     LOAD   x
                                            SUBTRACT    y
                                            JUMPNEG     a
                                            LOAD   y
                                            SUBTRACT    x
                                            JUMPNEG     a
                '≠' :           tulosta     LOAD   x
                                            SUBTRACT    y
                                            JUMPZERO    a
        ENDCASE
ENDMODULE
```

Konekielen käsky NOP (No Operation) on tyhjä käsky, jota tarvitaan koodin generoinnissa erityisesti nimiöiden yhteydessä. WHILE-rakenne

<while-lause> (WHILE, <ehto> (...), DO, <lausejono> (<lause> (...), ..., <lause> (...)), ENDWHILE)

käännetään seuraavasti:

```
MODULE generoiwhilelause(ehto-osa E, lause-osa P)
        Olkoon E = <ehto> (<tunnus> (x), <oper> (o), <tunnus> (y))
        Määrätään yksikäsitteiset nimiöt a ja b.
        tulosta         b:      NOP         (* silmukan alku *)
        generoiehto(x, o, y, a)
        generoilausejono(P)
        tulosta                 JUMP b
                          a:    NOP
ENDMODULE

MODULE generoilausejono(P1, ..., Pk)
        FOR i := 1, 2, ..., k DO
                generoi(Pi)
        ENDFOR
ENDMODULE
```

3) Optimointi

Mekaanisesti generoitua koodia voidaan yleensä parantaa. Eri kääntäjät eroavat tehokkuudeltaan siinä, miten hyvin ne kykenevät optimoimaan tuottamansa koodin käytettävälle tietokoneelle. Koodin optimointi ei ole helppoa. Lähinnä pyritään havaitsemaan ja poistamaan turhat ohjelman palaset sekä käyttämään niitä operaatioita, jotka ovat käytettävän koneen kannalta tehokkaimpia. Esimerkiksi lauseesta

```
IF x > y THEN x := x – y ELSE x := 0 ENDIF
```

generoitua koodia

```
        LOAD      x
        SUBTRACT  y
        JUMPNEG   a
        JUMPZERO  a
        LOAD      x          *
        SUBTRACT  y          *
        STORE     x
        JUMP      b
a:      NOP
        LOADI     0
        STORE     x
b:      NOP
```

voitaisiin nopeuttaa jättämällä tähdellä (*) merkityt käskyt turhina pois. Kääntäjän ei myöskään kannata varata muistia lukuvakiolle nolla, siksi ELSE-osan asetuslauseessa käytettiin välitöntä latausta LOADI.

Esimerkki. Tarkastellaan lopuksi yhteenvedonomaisesti, mitä kokonaiselle ohjelmalle tapahtuu käännettäessä. Olkoon määritelty seuraava kielioppi (mieti miten tämä eroaa s. 161 kieliopista ja kumpi on yleisempi ?):

```
<ohjelma>      →  <moduuli> { <moduuli> }
<moduuli>      →  MODULE <tunnus> ( { <tyyppi> <tunnuslista> } ) RETURNS <tyyppi>
                  <lauselista> ENDMODULE
<tyyppi>       →  INTEGER | REAL | BOOLEAN
<tunnuslista>  →  <tunnus> | <tunnuslista>, <tunnus>
<lauselista>   →  <lause> | <lause> <lauselista>
<lause>        →  <ehtolause> | <toistolause> | <asetuslause> | <returnlause>
<ehtolause>    →  IF <ehto> THEN <lauselista> [ ELSE <lauselista>] ENDIF
<toistolause>  →  WHILE <ehto> DO <lauselista> ENDWHILE
<asetuslause>  →  <tunnus> := <lauseke>
<returnlause>  →  RETURN <tunnus>
<ehto>         →  <tunnus> <relaatio> <luku> | <tunnus> <relaatio> <tunnus>
<relaatio>     →  = | >= | > | < | <= | <>
<lauseke>      →  <tunnus><operaattori><tunnus> | <tunnus> | <tunnus><operaattori><luku> | <luku>
<operaattori>  →  + | – | * | /
```

Käännetään seuraava kieliopin mukainen ohjelma konekielelle:

```
MODULE syt(INTEGER x, y) RETURNS INTEGER
    WHILE x <> y DO
        IF x > y THEN
            x := x – y
        ELSE
            y := y – x
        ENDIF
    ENDWHILE
    RETURN x
ENDMODULE
```

1) Selaus

Selauksen tulos on seuraava:

MODULE varattu	syt tunnus	(välim.	INTEGER varattu	x tunnus	, välim.	y tunnus	) välim.			
RETURNS varattu	INTEGER varattu	WHILE varattu	x tunnus	<> rel.	y tunnus	DO varattu				
IF varattu	x tunnus	> rel.	y tunnus	THEN varattu	x tunnus	:= as. op.	x tunnus	– oper.	y tunnus	
ELSE varattu	y tunnus	:= as. op.	y tunnus	– oper.	x tunnus	ENDIF varattu	RETURN varattu	x tunnus	ENDWHILE varattu	ENDMODULE varattu

2) Jäsennys

Yksinkertaistettuna jäsennyspuun sulkumerkkiesitys on seuraavanlainen (kielen avainsanat jätetty pois). Piirrä harjoituksena myös vastaava täydellinen jäsennyspuu.:

<ohjelma> (
    <moduuli> (
    <tunnus> (syt),
    <tyyppi> (INTEGER),
    <tunnuslista> (
        <tunnus> (x), <tunnus> (y)),
    <tyyppi>(INTEGER),
    <lauselista> (
        <toistolause> (
            <ehto> (<tunnus> (x), <rel> (<>), <tunnus> (y)),
            <lauselista> (
                <ehtolause> (
                    <ehto> (<tunnus> (x), <rel> (>), <tunnus> (y)),
                    <asetuslause> (
                        <tunnus> (x),
                        <lauseke> (<tunnus>(x), <oper>(–), <tunnus>(y))),
                    <asetuslause> (
                        <tunnus> (y),
                        <lauseke> (<tunnus>(y), <oper>(–), <tunnus>(x))))))),
        <returnlause> (<tunnus> (x)))))

3) Koodin generointi (symbolinen konekieli)

Muistinvaraus (todelliset osoitteet täydennetään latausvaiheessa):

tunnus	tyyppi	osoite
syt	INTEGER	-
x	INTEGER	-
y	INTEGER	-

Seuraava koodi saadaan mekaanisesti esitettyjä generointimoduuleja käyttäen. Koodia voi jälkeenpäin optimoida.

```
paluu:  paluuosoite
        JUMP        b0
x:
y:
syt:    moduulin arvo
b0:     NOP
b1:     NOP                                                          *
        LOAD        x                                                *
        SUBTRACT    y       <ehto>(x, <> , y)                        *
        JUMPZERO    a1                                               *
        LOAD        x                       *                        *
        SUBTRACT    y       <ehto>(x, > ,y) *                        *
        JUMPNEG     a2                      *                        *
        JUMPZERO    a2                      *                        *
        LOAD        x                       * <ehtolause>            * <toistolause>
        SUBTRACT    y       <asetuslause>(x, x, –, y)  *             *
        STORE       x                       *                        *
        JUMP        b2                      *                        *
a2:     NOP                                 *                        *
        LOAD        y                       *                        *
        SUBTRACT    x       <asetuslause>(y, y, –, x)  *             *
        STORE       y                       *                        *
b2:     NOP                                 *                        *
        JUMP        b1                                               *
a1:     NOP                                                          *
        LOAD        x
        STORE       syt     <returnlause>
        RETURN      paluu
```

5.3.4 Symbolinen konekieli

Esimerkeissämme olemme käyttäneet itse asiassa ns. ***symbolista konekieltä***. Symbolinen konekieli on hieman konekieltä korkeatasoisempi kieli, jonka jokainen käsky vastaa yksittäistä konekäskyä. Symbolinen konekieli on konekielten tavoin konekohtainen. Se poikkeaa konekielestä lähinnä muistiviittausten kohdalla. Konekielessä muistipaikkaan viitataan muistipaikan osoitteella, kun taas symbolisessa konekielessä muistipaikkaan voidaan viitata ohjelmoijan nimeämällä tunnuksella. Näin vältytään työläältä osoitteiden laskemiselta. Tunnusten ja osoitteiden välinen vastaavuus täytyy tietenkin määritellä ohjelmassa.

Ns. ***assemblerit*** ovat symbolisten konekielten kääntäjiä, jotka kääntävät symbolisella konekielellä (assembly languages) kirjoitettuja ohjelmia konekielelle. Koska symbolisella konekielellä on yksinkertainen syntaksi, assembler-kääntäjän tekeminen on helppoa. Selaus ja jäsennys ovat suoraviivaista. Jäsennyksen aikana symbolitauluun kerätään nimien ja muistiosoitteiden välinen vastaavuus. Myös koodin generointi on suoraviivaista. Jokaisesta symbolisen konekielen käskystä generoidaan yksi konekielinen käsky. Assemblerin tehtävänä on siis laskea symbolisia osoitteita vastaavat todelliset osoitteet ja korvata operaattoreiden nimet operaatiokoodeilla.

Assembler-kääntäjäkään ei yleensä tuota lopullista objektikoodia: käännetyssä ohjelmassa osoitteet eivät ole absoluuttisia, vaan suhteellisia. Suhteellisessa konekoodissa

ohjelman käyttämien muistipaikkojen osoitteet lasketaan suhteessa ohjelmalle varatun muistialueen alkuun. Tällaisen koodin etuna on sen uudelleensijoitettavuus (relocatable code): ohjelmaa ei tarvitse joka kerran suorittaa täsmälleen samalla muistialueella. Tietenkin ohjelman suhteelliset osoitteet täytyy ennen suoritusta (lataajan toimesta) tai suorituksen aikana korvata absoluuttisilla osoitteilla. Koska todellisten osoitteiden ylläpito on muistinhallintaa, se liittyy myös käyttöjärjestelmän tehtäviin.

5.4 Käyttöjärjestelmät

Käyttöjärjestelmä (operating system) on tärkein tietokoneen systeemiohjelmistoista ja samalla monimutkaisin. Ilman käyttöjärjestelmää tietokoneella ei oikein voi tehdä mitään. Sen tarkoituksena on helpottaa ja tehostaa tietokoneen käyttöä. Kaikissa tietokoneissa on käyttöjärjestelmä (esim. DOS, Windows, Unix), joka toimitetaan yleensä koneen mukana.

Käyttöjärjestelmän tärkeimpiä tehtäviä ovat

- keskusmuistin ja oheismuistin hallinta
- tiedonsiirron hallinta tietokonelaitteiston eri osien (syöttö- ja tulostuslaitteet, ulkoiset muistilaitteet) välillä (ts. monen laitteen samanaikainen hallinta)
- ohjelmien suorituksen ohjaus (ts. monen ohjelman samanaikainen hallinta)
- tietokoneen resurssien jako käyttäjille (ts. monen käyttäjän samanaikainen hallinta)
- tietojen pitkäaikaissäilytyksestä huolehtiminen (toissijaisen muistin ylläpito)

Käyttöjärjestelmä joutuu suorittaakseen jonkin työn (työ = toimenpidejono tehtävän suorittamiseksi) tekemään useita tehtäviä. Tarkastellaan esimerkiksi ohjelman suorittamista tietokoneessa, jolloin suoritetaan lähdekielisen ohjelman kääntäminen, objektiohjelman lataaminen keskusmuistiin ja sen suorittaminen. Työn suoritus vaatii seuraavat vaiheet:

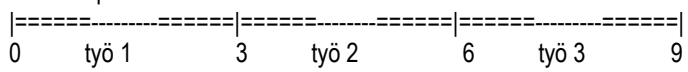
- siirrä kääntäjä oheismuistista keskusmuistiin
- käynnistä kääntäjä, ja syötä sille käännettävä ohjelma joltakin syöttölaitteelta
- käännä ohjelma, ja sijoita käännöksen tulos, objektiohjelma, oheismuistiin
- linkitä käännetty ohjelma ja sen apuohjelmat latausmoduuliksi
- siirrä latausmoduuli oheismuistista keskusmuistiin
- käynnistä suoritettava ohjelma, ja anna sille syöttötiedot joltakin syöttölaitteelta
- vastaanota ohjelman tulosteet jollekin tulostuslaitteelle

Ilman erikoisjärjestelyjä prosessori joutuu odottamaan jouten hidasta siirräntätoimintoa. Keskusyksikkö voi suorittaa syöttö- ja tulostustoiminnot oheistoimintoina, jolloin sille annetaan syöttö- ja tulostustoimintojen kanssa samanaikaisesti suoritettavaksi jokin muu työ, jonka suoritus keskeytetään ajoittain syöttö- ja tulostustoimintojen käsittelyä varten.

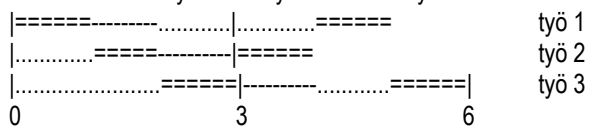
Esimerkki. Töiden suorittaminen peräkkäin ja rinnakkain. Suoritetaan kolme samanlaista työtä. Merkitään:

====== suoritettavana -------- odottaa syöttö-/tulostustoimintoa ..... odottaa suoritusvuoroa

Työt suoritetaan peräkkäin:



Kokonaissuoritusaika on yhdeksän yksikköä. Jos työt suoritetaan rinnakkain, päästään vähemmällä:



Jotta usean työn samanaikainen suorittaminen olisi mahdollista, käyttöjärjestelmän tulee tarjota seuraavat toiminnot:

- resurssien varaaminen (resource allocation): tietokoneen resurssien (muistin, siirräntälaitteiden ja prosessorin) varaaminen töille
- vuoronvaihto (dispatching): työn vaihto suoritusvuoroon eli prosessoriresurssin varaaminen työlle
- ajoitus (scheduling): päätös siitä, mikä oheismuistissa oleva työ tulee seuraavaksi suoritusvuoroon; kriteerejä: kiireellisyysaste, resurssien saatavuus, resurssien tarve, odotusaika
- resurssien suojaus (resource protection): varmistus siitä, että työ ei pääse käsiksi resurssiin, jota se ei ole varannut
- keskeytysten käsittely (interrupt handling): siirräntäpyyntöjen käsittely

5.4.1 Vuoronvaihto

Työn suoritus saattaa sisältää monen ohjelman suorituksen, esim. kääntäjän, lataajan ja käännetyn ohjelman suorituksen. Käyttöjärjestelmän tarjoamien toimintojen toteutus saattaa myös sisältää useiden erilaisten käyttöjärjestelmään kuuluvien ohjelmien suorituksen. Ohjelman suorittamista nimitetään prosessiksi (process). Käyttöjärjestelmä on vastuussa eri prosessien käynnistämisestä ja koordinoinnista. Osa näistä prosesseista suorittaa käyttöjärjestelmän toimintoja (ajoitus ja siirrännän käsittely), kun taas osa prosesseista suorittaa käyttäjän käynnistämiä toimintoja (esim. kääntäminen).

Koska yhden työn suoritus edellyttää usein monen prosessin suorittamista, keskusyksikkö (prosessori) tulee jakaa prosessien kesken. Tämä tehdään vaihtamalla suoritettavaa prosessia toistuvasti. Vaihto tehdään yleensä niin nopeasti, että kaikki prosessit näyttävät tulevan suoritetuiksi samanaikaisesti, vaikkakin käytännössä yksiprosessorisessa koneessa vain yksi prosessi voi olla kerrallaan suoritusvuorossa.

Jos koneessa on useampia prosessoreita (rinnakkaiskoneet), suoritettavina voi samanaikaisesti olla useampia prosesseja (yksi kutakin prosessoria kohden). Tällaisessa tapauksessa suoritettavia prosesseja on yleensä enemmän kuin prosessoreita, joten silloinkin joudutaan vaihtamaan ajoittain suoritusvuorossa olevaa prosessia.

Kun prosessi käynnistetään, se voi joutua mihin tahansa seuraavista tiloista:

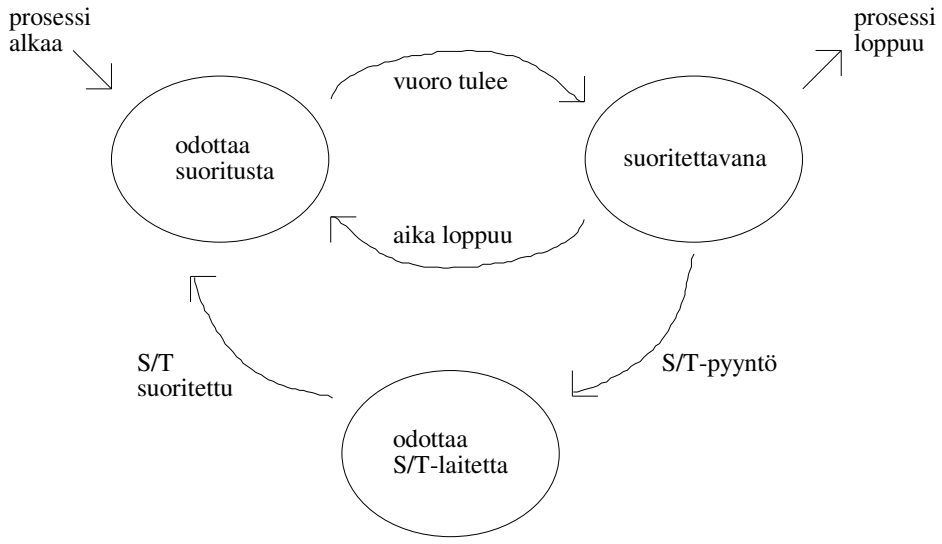
- suoritustila (running)
- suoritusvalmiustila (ready)
- siirräntälaitteen (tai muun resurssin) odotustila (blocked)

Prosessi on käynnistämisen jälkeen suoritusvalmiustilassa eli odottaa suoritusvuoroaan. Suoritusvalmiustilasta se voi siirtyä suoritustilaan. Suoritustilassa oleva prosessi siirtyy siirräntälaitteen odotustilaan silloin, kun se suorittaa siirräntäpyynnön, ja se jää siihen tilaan, kunnes siirräntä on valmis. Tällöin prosessi siirtyy suoritusvalmiustilaan ja myöhemmin vuorollaan suoritustilaan. Näin sama prosessi, jossa suoritetaan laskentaa ja siirräntää, kiertää tilasta toiseen.

Joissakin käyttöjärjestelmissä ainoastaan siirräntäpyyntö aiheuttaa prosessin siirtymisen pois suoritustilasta. Tällöin saattaisi käydä niin, että sellainen prosessi, joka ei suorita lainkaan (tai suorittaa vähän) siirräntää, olisi suoritettavana lähes koko ajan. Jotta keskusyksikkö saadaan tasaveroiseen käyttöön, on välttämätöntä rajoittaa sitä aikaa, jonka prosessi voi olla yhtäjaksoisesti suoritettavana ja vapauttaa keskusyksikkö toiselle prosessille suoritustilassa olevan prosessin saavuttaessa aikarajan. Aikaa, jonka prosessi voi olla yhtäjaksoisesti suoritettavana (suoritustilassa), nimitetään aikaviipaleeksi (time-slice). Kun prosessi on ollut suoritettavana aikaviipaleen ilmoittaman ajan, se siirretään suoritusvalmiustilaan, ja uusi prosessi siirretään suoritettavaksi. Prosessille annetaan uusi aikaviipale, kun se seuraavan kerran otetaan suoritettavaksi. Aikaviipaleen kokoon vaikuttaa mm. prosessin jo käyttämä aika.

Käyttöjärjestelmään kuuluva ohjelma vuoronantaja (dispatcher) eli prosessin vaihtaja huolehtii prosessin vaihdosta suoritustilaan, ja se herätetään seuraavissa tilanteissa:

- Suoritustilassa oleva prosessi suorittaa siirräntäpyynnön: prosessi siirtyy odottamaan siirräntälaitetta (odotustilaan) ja uusi, suoritusvuoroaan odottava (suoritusvalmiustilassa oleva) prosessi siirtyy suoritettavaksi (suoritustilaan).
- Suoritustilassa olevan prosessin aika loppuu: prosessi siirtyy odottamaan suoritus-vuoroaan (suoritusvalmiustilaan) ja uusi, suoritusvuoroaan odottava prosessi siirtyy suoritettavaksi (suoritustilaan).
- Suoritusvuorossa oleva prosessi päättyy: uusi, suoritusvuoroaan odottava prosessi siirtyy suoritettavaksi.



Vuoronantaja pyrkii prosessien väliseen tasapuolisuuteen ajassa ja vuorojen määrässä. Tärkeillä prosesseilla voi kuitenkin olla joitakin etuoikeuksia.

5.4.2 Ajoitus ja resurssien varaaminen

Suorituskelpoiset työt säilytetään keskusmuistissa, jos sinne mahtuu, muuten oheismuistissa. Yksittäisen työn suoritus vaatii usein monen prosessin käynnistämisen ja koordinoinnin joko samanaikaisesti tai peräkkäin. Käyttöjärjestelmään kuuluvan ohjelman, ajoittajan (scheduler), tehtävänä on saattaa prosessit suoritusvalmiiksi. Ajoittaja ottaa huomioon seuraavat asiat:

- prosessin tarvitsemien resurssien määrä
- vapaina olevien resurssien määrä
- prosessiin liittyvän työn prioriteetti (tärkeys- eli kiireellisyysaste)
- työn odotusaika (jonka se on jo odottanut)

Ajoittaja ylläpitää tietoja vapaina olevien resurssien määristä päivittäen tietoja resurssia varattaessa ja vapautettaessa. Resurssin varaus voi olla staattinen, jolloin kaikki prosessin tarvitsemat resurssit varataan, kun prosessi käynnistetään ja vapautetaan, kun prosessi on suoritettu. Resurssin varaus voi olla myös dynaaminen, jolloin resurssit varataan prosessille tarvittaessa ja vapautetaan, kun prosessi ei niitä enää tarvitse. Staattisessa varauksessa prosessi käynnistetään vasta sitten, kun kaikki sen tarvitsemat resurssit ovat saatavilla. Dynaamisessa varauksessa prosessi voidaan käynnistää milloin tahansa, mutta se voi joutua odotustilaan, jollei resurssia ole saatavilla tarvittaessa. Dynaaminen varaus voi johtaa parempaan resurssien hyväksikäyttöön, mutta on vaikeampi hallita. Erityisesti saattaa syntyä ns. ***lukkiuma*** (deadlock): kaksi prosessia pitää kumpikin hallussaan resurssia, jota toinen tarvitsee. Jos kumpikaan prosessi ei voi luopua resurssistaan, kummankaan prosessin suoritus ei voi jatkua, ja kumpikin prosessi jää resurssin odotustilaan "ikuisiksi ajoiksi". Lukkiumien a) havaitseminen, b) korjaaminen ja c) ennalta ehkäiseminen ovatkin melko vaikeita ongelmia, jotka käyttöjärjestelmän tulee kyetä ratkaisemaan.

5.4.3 Muistinhallinta

Eräs käyttöjärjestelmän tärkeimmistä tehtävistä on tietokoneen muistinhallinta (memory management), johon liittyy kolme asiaa:

1) ***Varaus*** (allocation): Suoritettavalle prosessille tulee varata riittävästi muistia.
2) ***Suojaus*** (protection): Suoritettava prosessi ei saa päästä muistialueelle, jota sille ei ole varattu.
3) ***Hyväksikäyttö*** (utilization): Muisti on tärkeä resurssi, koska se on ainoa paikka, missä suoritettavien prosessien ohjelmat ja tiedot voivat sijaita. Muistia tulee käyttää tehokkaasti hyväksi: sitä tulee varata riittävästi ja tulee varmistua siitä, että muistissa on aina suoritusvalmiustilassa olevia prosesseja.

Käyttöjärjestelmän tulee voida varata mikä tahansa riittävän suuri vapaa alue muistista. Sen tulee myös pystyä siirtämään prosessiin liittyviä tietoja muistialueelta toiselle yhdistettäessä pieniä varaamattomia alueita yhdeksi, käyttökelpoisemmaksi alueeksi. Tämä

tarkoittaa sitä, että kun ohjelma kirjoitetaan ja käännetään, ei ohjelmoija eikä kääntäjä voi etukäteen tietää, missä kohden muistia ohjelma tulee suoritusaikanaan olemaan.

Kääntämisvaiheen eräs tehtävä on generoida objektiohjelman muistiosoitteet. Mutta miten kääntäjä voi generoida oikeat osoitteet, kun se ei tiedä, missä kohden muistia ohjelma tulee suoritusaikana olemaan? Kääntäjän generoimien ohjelmaosoitteiden (program addresses) ja todellisten muistiosoitteiden (memory addresses) välillä on selvä ero. Kääntäjä generoi ohjelmaosoitteet suhteellisina nollasta eteenpäin: nämä osoitteet tulee myöhemmin muuntaa vastaamaan sen muistialueen osoitteita, jotka käyttöjärjestelmä on varannut muistista ohjelmalle. Ohjelmaosoitteiden muuntamisessa todellisiksi muistiosoitteiksi on kaksi perusvaihtoehtoa: (a) staattinen muunto, jonka lataaja tekee ennen suorituksen alkua, (b) dynaaminen muunto, jonka suorittaa keskusyksikkö objektiohjelman suorituksen aikana. Jälkimmäiseen on olemassa useita menetelmiä, joista tässä esitellään alkuosoitteiden käyttö ja sivutus. Lisäksi esitellään niihin liittyviä muistinhallintatekniikoita.

1) Alkuosoitteiden käyttö

Pidetään yllä suoritettavana olevan prosessin alkuosoitetta (ensimmäinen muistipaikka, joka on varattu suoritettavalle prosessille). CPU muuntaa prosessin jokaisen ohjelmaosoitteen sopivaksi muistiosoitteeksi yksinkertaisesti lisäämällä kuhunkin ohjelmaosoitteeseen alkuosoitteen arvon. Aktiivisen prosessin alkuosoitetta säilytetään erityisessä kantarekisterissä. Alkuosoitteen käyttö mahdollistaa kunkin prosessin tietojen sijoittelun mihin tahansa vapaaseen paikkaan muistissa ja antaa käyttöjärjestelmälle mahdollisuuden siirrellä prosessien tietoja tarvittaessa. Alkuosoitteen käyttö ei kuitenkaan vielä riitä yksin suojaamaan toisen prosessin varaamaa muistiresurssia vierailta viittauksilta. Nämä voidaan estää pitämällä yllä tietoa suoritettavan prosessin pituudesta (= prosessille varattujen muistipaikkojen lukumäärä).

Heittovaihto

Käyttöjärjestelmän tavoitteena on pitää muisti mahdollisimman täynnä suoritusvalmiita prosesseja. Jos prosessi siirtyy siirräntälaitteen tai muun resurssin odotustilaan, sille varattu muistialue on hyödytön, eli muisti ei ole tehokkaassa käytössä. Jos muistia on vähän käytettävissä, on olemassa vaara, että muisti täyttyy ennen pitkää odotustilassa olevien prosessien tiedoilla. Muistinkäyttöä voidaan tehostaa tällaisessa tilanteessa suorittamalla ns. ***heittovaihto*** (swapping): poistetaan odotustilassa olevan prosessin tiedot muistista oheismuistiin ja vapautetaan näin muistialue muuhun käyttöön. Levylle heitetyn prosessin tiedot haetaan myöhemmin takaisin muistiin prosessin siirtyessä odotustilasta suoritusvalmiustilaan.

Heittovaihdon suorituksen kriteerejä:

- prosessin prioriteetti
- prosessin varaama muistin määrä
- siirrännän odotusaika (odotustilan kesto)
- muistissaoloaika tähän saakka

Pirstoutuminen

Suoritetun prosessin muistialue voidaan varata uudelle prosessille. Uuden prosessin muistin tarve ei välttämättä ole sama kuin päättyneen prosessin muistin tarve. Jos se on pienempi, osa vapautuneesta muistialueesta jää käyttämättä. Pitkän ajan kuluessa muistitilan vapauttamiset ja uudelleen varaamiset voivat aiheuttaa ***pirstoutumista***

(fragmentation): varattujen alueiden välissä on hyödyttömän pieniä yhtenäisiä vapaita alueita. Tämä johtaa muistin tehottomaan hyväksikäyttöön. Käyttöjärjestelmän tehtävänä on estää muistin pirstoutuminen yhdistämällä varattujen alueiden välissä olevat hyödyttömän pienet vapaat alueet yhtenäiseksi, käyttökelpoiseksi alueeksi ja tarvittaessa järjestämällä varatut alueet uudelleen.

Käytettäessä alkuosoitetta suoritettavana olevan prosessin kaikkien tietojen tulee olla samanaikaisesti muistissa, joten muistista tulee löytyä riittävästi tilaa prosessin suorittamiseksi. Näin meneteltäessä muistikapasiteetti rajoittaa suoritettavien ohjelmien kokoa.

2) Sivutus

Sivutus (paging) on muistinhallintatekniikka, joka mahdollistaa prosessin suorittamisen, vaikka vain osa sen tiedoista on muistissa. Siksi se mahdollistaa prosessin suorittamisen, vaikka kaikki sen tiedot eivät mahtuisikaan muistiin samanaikaisesti. Sivutuksessa jokaisen prosessin tiedot jaetaan samankokoisiin osiin, sivuihin. Tietokoneen muisti jaetaan myös sivunkokoisiin alueisiin, kehyksiin. Tietokoneen sivukoko ilmoittaa, montako sanaa (muistipaikkaa) mahtuu yhdelle sivulle. Käyttöjärjestelmä huolehtii muistisivujen varaamisesta. Kullakin prosessilla voi olla sivuja sekä muistissa että oheismuistissa.

Prosessin tietojen jakaminen sivuihin on yksinkertaista. Jos sivukoko on 1000, sivu 0 sisältää tiedot, jotka löytyvät ohjelmaosoitteista 0-999, sivu 1 sisältää tiedot, jotka löytyvät ohjelmaosoitteista 1000-1999 jne. Prosessin jakaminen sivuihin on näkymätöntä ohjelmoijille ja kääntäjille, se on täysin käyttöjärjestelmän ja tietokonelaitteiston ominaisuus. Keskusyksikkö muuntaa ohjelmaosoitteen muistiosoitteeksi a) laskemalla ohjelmaosoitteesta, mille sivulle ja mihin muistipaikkaan sivulla viitataan sekä b) käyttämällä hyväkseen käyttöjärjestelmän ylläpitämää tietoa määrittäessään, minkä muistisivun ohjelmasivu varaa.

Ensimmäinen vaihe on suoraviivainen: jaetaan ohjelmaosoite sivukoolla. Jos esimerkiksi ohjelmaosoite on 2764 ja sivukoko on 1000, niin silloin viitataan sivulle kaksi muistipaikkaan 764. Käytännössä sivukooksi valitaan kakkosen potenssi (tavallisesti 512, 1024, 2048), mikä yksinkertaistaa binäärilukujen jakolaskua.

Toinen vaihe osoitteen muuntamisessa tehdään käyttäen ns. sivutaulua (page table), joka ilmaisee, mikä muistisivu (jos mikään) on varattu kullekin prosessin sivulle. Käyttöjärjestelmä ylläpitää omaa sivutaulua kullekin prosessille (varmistaen samalla, että ei käytetä muuhun tarkoitukseen varattua aluetta) ja päivittää sitä tarvittaessa. Kunkin prosessin sivutaulu sisältää kaikista sivuista seuraavat tiedot:

- onko sivu muistissa vai ei
- jos sivu on muistissa, muistisivun osoite, muutoin sivun sijainti oheismuistissa

Jos viitattu sivu on muistissa, ohjelmaosoitteen muuntaminen muistiosoitteeksi onnistuu helposti. Jos prosessissa viitataan sivulle, joka ei ole muistissa, viitattu sivu tulee hakea muistiin, ennen kuin osoitteen muuntaminen onnistuu lopullisesti. Tällöin prosessi siirtyy odotustilaan, ja jokin toinen prosessi pääsee suoritettavaksi. Kun sivu on siirretty muistiin, prosessi siirtyy suoritusvalmiustilaan, josta se voi myöhemmin siirtyä suoritettavaksi.

Sivutus mahdollistaa sen, että ohjelman koko voi olla suurempi kuin tietokoneen muisti, koska ohjelman ei tarvitse olla kokonaan muistissa voidakseen tulla suoritetuksi.

Sivutus on ns. ***virtuaalimuistitekniikka*** (virtual memory technique), joka laajentaa muistin fyysistä muistia suuremmaksi.

Kun johonkin ohjelmasivuun viitataan, se pitää hakea muistiin käsiteltäväksi. Jos muistissa ei ole tilaa, sieltä pitää palauttaa jokin sivu oheismuistiin. Sivun palauttamiseksi muistista oheismuistiin on erilaisia strategioita:

- muistissa pisimpään (longest resident) ollut sivu palautetaan
- kauimmin käyttämättä ollut (least recently used) sivu palautetaan
- harvimmin käytetty (least frequently used) sivu palautetaan

Koska kaikki ohjelmasivut ja muistisivut ovat samankokoisia, muistin pirstoutumista ei voi tapahtua, ja muistin hyväksikäyttö on siten tehokasta. Sivutaulujen ylläpitämiseksi tarvitaan kuitenkin lisätilaa.

5.4.4 Siirräntä

Eräs käyttöjärjestelmän tehtävistä on käsitellä siirräntätoiminnot. Tämä on annettu käyttöjärjestelmän tehtäväksi useista syistä: ensinnäkin siirräntälaitteet ovat hyvin erilaisia, ja toiseksi, jotta prosessorin varaus olisi tehokasta, käyttöjärjestelmän tulee tietää, milloin siirräntätoimintoja ryhdytään suorittamaan ja milloin ne on saatu valmiiksi: tämä on helpointa järjestää, jos käyttöjärjestelmä käsittelee kaiken siirrännän itse.

Jotkut siirräntälaitteista ovat ***jaettavissa*** (shareable): ne voivat käsitellä peräkkäisiä siirräntätoimintoja, jotka ovat peräisin eri prosesseilta. Esimerkiksi levy (tai keskusmuisti) on jaettavissa, koska erilaiset toiminnot voidaan kohdistaa eri alueille levyä (tai keskusmuistia) ilman sekaantumismahdollisuutta. Muut laitteet ovat ***jakamattomia*** (unshareable). Esimerkiksi rivikirjoitin on jakamaton laite. Jos usea prosessi käyttää sitä samanaikaisesti, tulosteet eri prosesseilta tulisivat samalle paperille sekaisin. Jakamaton laite voidaan varata ainoastaan yhdelle prosessille kerrallaan, ja käyttöjärjestelmän täytyy varmistua siitä, että prosessi käyttää ainoastaan niitä laitteita, jotka sille on varattu. Tämäkin on helpointa, jos käyttöjärjestelmä huolehtii kaikesta laitteiden hallinnasta.

Prosessi suorittaa siirräntäpyynnön kutsumalla sopivaa käyttöjärjestelmäohjelmaa, joka tarkastaa pyynnön laillisuuden (siirräntälaite on varattu siirräntää pyytävälle prosessille), käynnistää siirrännän, siirtää prosessin siirrännän odotustilaan ja pyytää vuoronantajaa valitsemaan uuden prosessin suoritettavaksi. Siirrännän suorituksen päättymismerkkinä on keskeytys, jolloin keskusyksikkö siirtyy suorittamaan käyttöjärjestelmän ohjelmaa, keskeytyskäsittelijää (interrupt handler), joka käsittelee keskeytyksen. Keskeytyskäsittelijä tunnistaa keskeytyksen aiheuttaneen laitteen ja prosessin, jonka siirräntätoiminnosta on kyse, muuttaa ko. prosessin tilan siirrännän odotustilasta suoritusvalmiustilaan, ja antaa prosessorin jatkaa prosessia, jonka suoritus keskeytyi.

Prosessi voi käyttää jakamatonta laitetta vain, jos se on sille varattu. Joinakin aikoina monet prosessit odottavat paljon käytettyjä laitteita, välillä samat laitteet saattavat olla jouten. Paljon käytettyjen jakamattomien laitteiden kuormituksen tasaamiseksi on mahdollista käyttää ***sivuajoteknikkaa*** (spooling; Simultaneous Peripheral Outputting On-Line): kaikki siirräntä jakamattomalle laitteelle suunnataan jollekin välissä olevalle jaettavissa olevalle laitteelle, esim. levylle, josta se sitten myöhemmin edelleen siirretään jakamattomalle laitteelle.

Tarkastellaan esimerkiksi prosessia, joka haluaa tulostaa kirjoittimelle. Prosessi kutsuu käyttöjärjestelmän siirräntäohjelmaa, joka lähettääkin tulosteen levylle kirjoittimen sijasta. Muiden prosessien kirjoittimelle suunnatut tulosteet lähetetään samalla tavoin levylle ns. kirjoitinjonoon. Sivuajosta huolehtiva käyttöjärjestelmäprosessi (spooler), jolle kirjoitin on pysyvästi varattu, siirtää tulosteet levyltä kirjoittimelle, kun se vapautuu. Näin prosessi ei joudu odottamaan tulostuksen valmistumista, vaan voi jatkaa suoritustaan samaan aikaan, kun spooleri huolehtii tulostuksesta.

5.5 Tiedostojärjestelmä

Useimmissa tietokonejärjestelmissä on tarve tallentaa tietoa helposti haettavaan muotoon pitkiksi ajoiksi. Tallennettavan tiedon luonne vaihtelee sen mukaan, mihin järjestelmää käytetään, mutta yleisesti ottaen tieto on sellaista tietoa, jota tullaan toistuvasti käyttämään, esimerkiksi: systeemiohjelmat, datatiedot, käyttäjien kirjoittamat ohjelmat sekä tietokannat.

Jotta tieto olisi helposti saatavilla, se on tallennettava tietokonejärjestelmään (on epäkäytännöllistä syöttää tietoja joka kerta sisään joltakin syöttölaitteelta). Tietojen pitkäaikaissäilytyspaikka on oheismuisti eli toissijainen muistilaite, esim. magneettinauha tai levy. Käyttöjärjestelmän tehtävänä on hallita oheismuistia siten, että tieto olisi helposti paikallistettavissa ja haettavissa.

Toissijaisessa muistissa tieto säilytetään erikokoisissa tiedostoissa (files). Jokainen tiedosto on kokoelma yhteenkuuluvia tietoja. Esimerkiksi ohjelmat säilytetään tiedostoissa. Tiedosto on siis käyttöjärjestelmän ylläpitämä looginen yksikkö. Tiedostojärjestelmä (file system) on se osa käyttöjärjestelmää, joka huolehtii tiedostoista.

Tiedostojärjestelmän tyypillisiä toimintoja ovat:

- tiedostojen luonti, käsittely ja tuhoaminen
- tiedostojen ylläpito (oheismuistin ylläpito)
- oheismuistivälineiden muistitilan hallinta
- tiedostojen suojaaminen luvattomilta käyttäjiltä ja laitevioilta.

Käyttäjän ei tarvitse tietää tiedoston fyysisestä sijainnista mitään, vaan hän voi antaa tiedostolle nimen, jolla hän sitten pääsee käsiksi tiedoston sisältämiin tietoihin. Tiedostojärjestelmä ylläpitää tiedostojen nimien hakemistoa (directory) ja nimiä vastaavien fyysisten tiedostojen sijaintitietoja muistivälineellä. Oheismuistista tila varataan kiinteänkokoisina lohkoina (blocks) eli ***sivuina*** (page; esimerkiksi 512 tai 1024 sanaa). Jokainen tiedosto varaa tarvittavan määrän sivuja, jotka voivat sijaita missä tahansa toissijaisella muistivälineellä (sivujen ei tarvitse olla vierekkäisiä). Tiedostojärjestelmä ylläpitää tietoa vapaista sivuista ja tiedostoille varatuista sivuista.

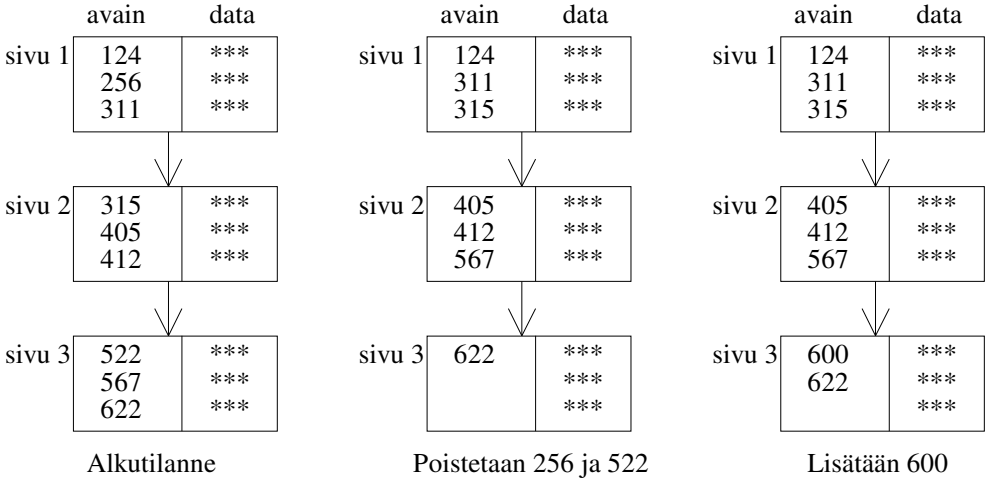
Oheismuistin luku ja kirjoitus tapahtuvat sivu kerrallaan. Oheismuistin käsittelynopeudessa on ratkaisevaa suoritettujen luku- ja kirjoitusoperaatioiden lukumäärä, ei niinkään siirretyn datan volyymi. Siksi kannattaa käyttää melko suurta sivukokoa, ja pyrkiä sijoittamaan peräkkäin käsiteltävät tietueet samalle sivulle. Jatkossa oletetaan yksinkertaisuuden vuoksi, että tietueet ovat enintään sivun kokoisia, ja että yhtä tietuetta ei jaeta kahdelle eri sivulle. Sivujen ei tarvitse olla täynnä – usein niihin jätetään tarkoituksella kasvuvaraa.

Tiedostot voidaan organisoida kolmella tavalla: ne voivat olla ***peräkkäistiedostoja*** (sequential files), jolloin sivut on käsiteltävä (luettava tai kirjoitettava) peräkkäisessä järjestyksessä, ***suorasaantitiedostoja*** (random access files), joiden sivuja voidaan käsitellä missä järjestyksessä tahansa, tai ***indeksoituja tiedostoja***, jotka erillisin hakemisto-tiedostoin tarjoavat kaikkein monipuolisimmat saantimenetelmät.

5.5.1 Peräkkäisorganisaatio

Tietueet sijaitsevat tiedostoon kuuluvilla sivuilla peräkkäin. Sivujen sisällä ne ovat fyysisesti peräkkäin, mutta sivut voivat keskenään olla mielivaltaisesti sijoitettuina. Looginen peräkkäisyys saavutetaan esimerkiksi asettamalla kullekin sivulle aina seuraavan sivun osoite. Toinen mahdollisuus on, että levyjärjestelmä ylläpitää käyttäjälle näkymätöntä hakemistoa, jossa sivujen järjestys on määritelty.

Peräkkäistiedostoa ei voi yleensä vapaasti muuttaa, vaan päivitetty versio joudutaan kopioimaan kokonaisuudessaan uudeksi peräkkäistiedostoksi. Joskus tiedoston loppuun voidaan kuitenkin tehdä lisäyksiä (append). Usein on hyödyllistä tallentaa tietueet avainkentän mukaiseen järjestykseen. Jos päivitykset kerätään pitemmän ajan (esim. päivän) kuluessa ja lajitellaan saman kentän mukaan, voidaan päivitykset toteuttaa yhdellä läpikäynnillä, ns. ***eräajona***. Seuraavassa esimerkissä sivulle mahtuu kolme tietuetta.



Erikoistapauksena peräkkäistiedostosta tulkoon mainittua ***tekstitiedosto***, jossa tietueena voidaan pitää yhtä merkkiä, ja tietueissa ei ole mitään erityistä avainta.

5.5.2 Suora organisaatio

Suora organisaatio tarjoaa suorasaannin; nopean ja (lähes) vakioaikaisen tavan mielivaltaisen tietueen hakemiseen. Tietueen osoite saadaan tietyllä kaavalla tietueessa olevan avainkentän x perusteella. Yksinkertaisimmassa muodossaan x on kokonaisluku (1, 2, ...), joka sellaisenaan määrää tietueen suhteellisen osoitteen eli järjestysnumeron tiedoston sisällä. Oikean sivun järjestysnumero saadaan kaavalla $\lceil x/s \rceil$ (pyöristys ylöspäin; s on sivun kapasiteetti), kun tietueet ovat kiinteän mittaisia. Sivun todellisen osoitteen määräämiseksi on kaksi vaihtoehtoa: (1) Jos on mahdollista varata tiedostolle fyysisesti peräkkäinen joukko sivuja, saadaan osoite helposti selville alkuosoitteen ja sivun järjestysnumeron avulla. (2) Jos tiedoston sivut sijaitsevat hajallaan, tarvitaan

hakemisto (vektori), josta tiedoston sivujen osoitteet löytyvät. Sivun järjestysnumeroa käytetään indeksinä tähän vektoriin.

Yksinkertainen suora organisaatio on erittäin tehokas, mutta jos kaikki mahdolliset avaimen arvot eivät ole käytössä, tilaa tuhlaantuu, koska jokaiselle arvolle varataan tietuepaikka. ***Hash-organisaatiossa*** (***hajautettu***-organisaatio) osoitteen laskemisessa käytetään monimutkaisempaa kaavaa, joka ei säilytä avainten alkuperäistä järjestystä, eikä määrää tietueen järjestysnumeroa yksikäsitteisesti. Kyseessä on ns. kutistava kuvaus: suuri avainavaruus kuvataan hash-funktiolla pienemmäksi osoiteavaruudeksi, jolloin säästyy tilaa. Hash-funktio (h) tuottaa sivun järjestysnumeron h(x), jota kutsutaan tietueen ***kotiosoitteeksi***. Samaan osoitteeseen voi osua useita tietueita, ja niitä kutsutaan ***törmäyksiksi***. Jos sivulle on pyrkimässä useampia tietueita kuin sinne mahtuu, tapahtuu ***ylivuoto***, joka pitää hoitaa jotenkin. Tähän on kaksi päävaihtoehtoa:

1. ***Suljettu hash***: Ylivuototietueet sijoitetaan sellaisille sivuille tiedostoalueen sisällä, joilla vielä on tilaa. Yksinkertaisinta on etsiä lähin seuraava vajaa sivu (ns. lineaarinen ylivuotomenettely).
2. ***Avoin hash***: Ylivuototietueet sijoitetaan erilliselle ylivuotoalueelle, johon kotiosoitteesta asetetaan linkki.

Esimerkki. Olkoon hash-funktio h(x) = x mod 7 + 1. Oletetaan yksinkertaisuuden vuoksi, että käytettävissä on peräkkäinen joukko oheismuistin sivuja. Suljettu hash lineaarisella ylivuotomenettelyllä toimii seuraavasti:

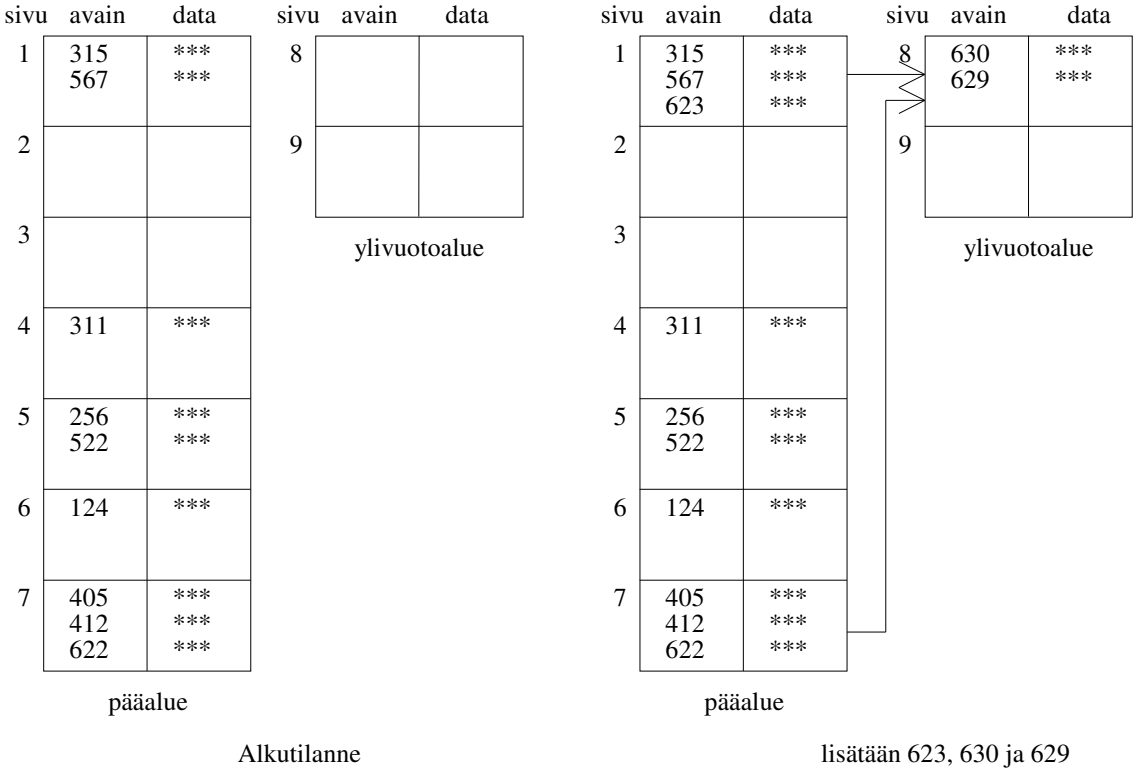
Alkutilanne

sivu	avain	data
1	315 567	*** ***
2		
3		
4	311	***
5	256 522	*** ***
6	124	***
7	405 412 622	*** *** ***

lisätään 623 ja 630

sivu	avain	data
1	315 567 623	*** *** ***
2	630	***
3		
4	311	***
5	256 522	*** ***
6	124	***
7	405 412 622	*** *** ***

Lisäysten suoritus käyttäen avointa hashia ylivuotoalueella:

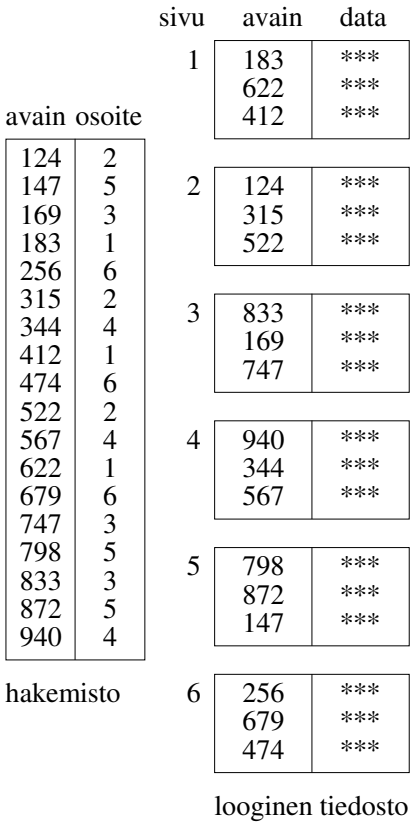


5.5.3 Indeksoitu organisaatio

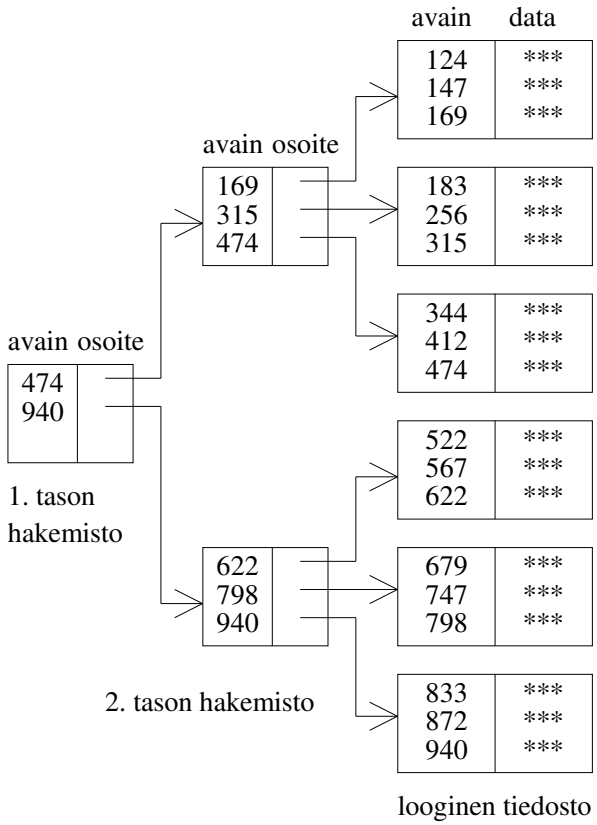
Indeksoidut rakenteet jakautuvat kahteen tiedostoon: varsinaiseen tiedostoon (data file) ja hakemistotiedostoon (index file), jotka organisoidaan erikseen. Varsinainen tiedosto sisältää loogiset tietueet, ja hakemisto muodostuu ns. indekseistä eli avain-osoite-pareista. Hakemisto voi olla kahta päätyyppiä:

1. ***Tiheä*** hakemisto sisältää tiedoston jokaisen tietueen avaimen ja osoitteen. Varsinainen tiedosto voi tällöin olla organisoitu mielivaltaisella tavalla.
2. ***Harva*** hakemisto edellyttää, että varsinainen tiedosto on avaimen mukaan järjestyksessä (peräkkäisorganisaatio). Hakemisto sisältää kunkin sivun suurimman avaimen ja osoittimen ko. sivulle.

Hakemisto voidaan edelleen organisoida vaikkapa indeksoidusti, jolloin muodostuu useampitasoinen hakemisto: Ylemmän tason hakemisto on aina harva ja sisältää alemman tason hakemiston kunkin sivun suurimman avaimen ja osoittimen sivulle. Alimman tason hakemisto voi olla harva tai tiheä. Monitasoinen hakemisto on itse asiassa puurakenne, joka yleensä on leveä ja siksi matala; datatietueet ovat normaalisti saavutettavissa 2-4 sivun haulla.

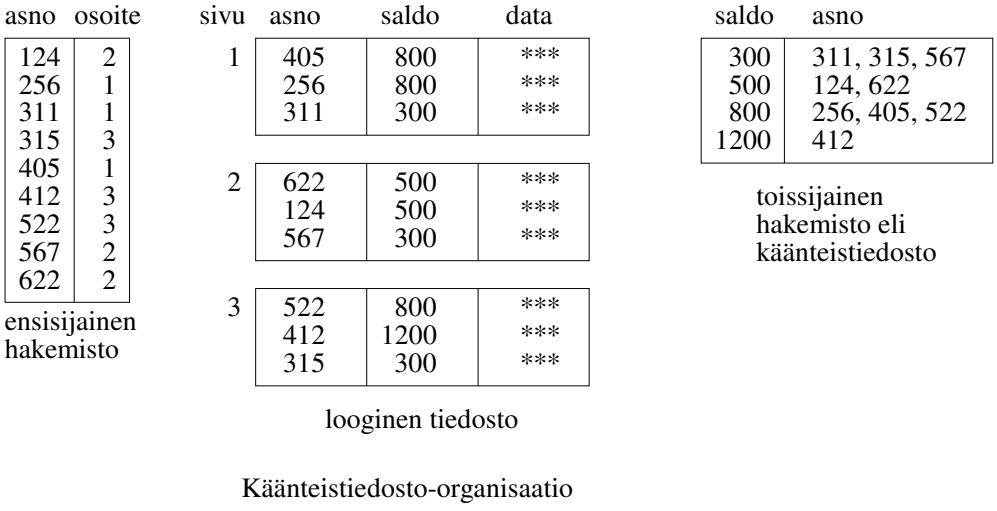


Yksitasoinen tiheä hakemisto



Kaksitasoinen harva hakemisto

Yhdellä loogisella tiedostolla voi olla useampiakin hakemistoja. Pääavaimen lisäksi voidaan tiedosto indeksoida toissijaisten avainten suhteen. Ideana on varmistaa tietueiden nopea saanti jokaisen avaimena käytettävän kentän (tai kenttäyhdelmän) arvon perusteella. Tällaista toissijaisten avainten ja indeksien avulla organisoitua rakennetta nimitetään ***käänteistiedosto-organisaatioksi***. Nimitys tulee siitä, että kenttien roolit ovat kääntyneet: kun normaalisti tietyn kentän arvo löydetään tietueen osoitteen avulla, käänteistiedostosta tietueen osoite on löydettävissä kentän tietyn arvon perusteella. Jokaista toissijaista avainta kohden tarvitaan oma tiheä hakemisto eli käänteistiedosto. Erona tavallisiin hakemistoihin on se, että toissijaiset avaimet eivät pääavaimien tavoin ole yksikäsitteisiä. Niinpä kutakin avainarvoa kohti tarvitaan lista osoittimia datatietueisiin. Suorien osoittimien haittana on, että tietueen siirto aiheuttaa kaikkien hakemistojen päivityksen. Siksi käytetäänkin usein pääavainarvoja loogisina osoittimina. Haku tapahtuu tällöin kolmessa vaiheessa: 1) toissijaishakemisto, 2) ensisijainen hakemisto, 3) varsinainen tiedosto. Tarkastellaan esimerkkinä tiedostoa, joka sisältää asiakkaiden tilitietoja. Tiedosto on normaalisti indeksoitu pääavaimena käytettävän asiakasnumeron mukaan. Toissijaisena avaimena halutaan käyttää tilin saldoa, jolloin tätä varten on perustettava käänteistiedosto.



Käänteistiedosto-organisaatio

6 Epädeterminismi, rinnakkaisuus ja vapaajärjesteisyys

Koko tähänastista tarkastelua on leimannut vahva sitoutuminen perinteiseen algoritmiseen tietojenkäsittelyyn tavanomaisella von Neumann -arkkitehtuurin mukaisella tietokoneella. Kantavia periaatteita ovat olleet determinismi, peräkkäisyys ja imperatiivinen ohjelmointiajattelu. Tässä luvussa tehdään suppea katsaus vaihtoehtoisiin malleihin. Ensin katsotaan, miten ratkotaan luonteeltaan epädeterministisiä ongelmia pohjimmiltaan deterministisin algoritmein. Sitten tarkastellaan sitä, mitä annettavaa useamman prosessorin sisältävillä rinnakkaiskoneilla on algoritmien suorittamiseen. Lopuksi esitellään ns. vapaajärjesteiseen ohjelmointiin perustuvaa ei-proseduraalista ongelmanratkaisua, joka ei ole sitoutunut peräkkäisyyteen ja determinismiin. Rinnakkaisuutta lukuun ottamatta tässä luvussa käsiteltävät ongelmat ja niiden ratkaisut liittyvät ns. ***tekoälyyn*** (artificial intelligence).

6.1 Epädeterministiset ongelmat

Kaikki edellä esitetyt algoritmit ovat olleet deterministisiä: suorituksen jokaisessa vaiheessa on tarkkaan tiedetty, millä yksikäsitteisellä tavalla suoritusta jatketaan. Eri tilanteissa on toki voitu toimia eri tavoin, mutta valinta vaihtoehtojen välillä on puhtaasti deterministinen. Monet tehtävät ovat kuitenkin luonteeltaan epädeterministisiä. Erilaisissa hakuongelmissa haun suorittamiseen käytettävissä olevien toimenpiteiden joukko on etukäteen tiedossa, mutta sen sijaan on epäselvää, mihin haku tulisi suunnata. Saatetaan tietää, että ratkaisu kyllä löytyy käytettävissä olevin menetelmin, mutta ei tiedetä mistä. Esimerkiksi erilaisten pelien pelaamisessa koko idea on juuri tehtävän epädeterministisyydessä: mikä mahdollisista siirroista kannattaa kulloinkin tehdä, tai miten päästään labyrintista ulos mahdollisimman suoraa reittiä.

6.1.1 Hakuprobleemat

Epädeterministiset ongelmat voidaan formuloida hakuongelmina. ***Hakuprobleema*** muodostuu ongelman ratkaisun tilasta ja tilasta toiseen vievistä siirroista. Ongelman tilaesitykseen kuuluvat:

- tilojen joukko S
- alkutila $s \in S$
- lopputilojen joukko $F \subseteq S$
- siirrot t: $S \rightarrow S$, $s_i \rightarrow s_j$

Ongelman ratkaisu alkaa ***alkutilasta***. Kussakin tilassa on joukko sallittuja ***siirtoja***, jotka vievät uuteen tilaan. Kaikki mahdolliset tilat ja niitä yhdistävät siirrot muodostavat graafin, jota kutsutaan ***hakuavaruudeksi***. Avaruudessa suoritettava haku muodostaa ***hakupuun***. Tehtävänä on löytää alkutilasta (puun juuri) johonkin ***lopputilaan*** (lehteen) johtava ***ratkaisupolku***. Joissakin tapauksissa voidaan lopputilojen joukosta vielä määritellä paras.

Hakuongelmat voidaan jakaa kahteen luokkaan. Jos hakuavaruus on äärellinen, eli graafi on annettu tai se voidaan kokonaisuudessaan muodostaa haun aikana, on hakuprobleemakin äärellinen, ja ratkaisu löytyy varmasti. Kyse onkin siitä, miten ratkaisu löydetään nopeimmin, tai miten löydetään paras ratkaisu. Jos taas graafia ei ole annettu, vaan tunnetaan ainoastaan ongelman tilaesitys, saattaa hausta muodostua ääretön. Tällöin ratkaisua ei välttämättä löydetä lainkaan, tai sen löytyminen voi kestää liian kauan.

6.1.2 Syvyys- ja leveyshaku

Epädeterministen hakuongelmien ratkaisemisen perusstrategiat ovat syvyyshaku ja leveyshaku. ***Leveyshaku*** (breadth first search) on luonteeltaan varovainen ja pessimistinen menetelmä. Siinä haku etenee leveänä rintamana joka suuntaan yhtä pitkälle. Leveyshaku käyttää tietorakenteena FIFO-periaatteella toimivaa tilojen jonoa. Leveyshaku on

- täydellinen: se käy tarvittaessa läpi koko hakugraafin, mistä syystä se on hidas;
- varma: täydellisyyden ansiosta ratkaisu (jos sellainen on olemassa) aina löydetään ennemmin tai myöhemmin äärettömässäkin haussa; ja
- optimaalinen: samasta syystä äärellisessä haussa löydetään myös paras ratkaisu, jos tehtävä on sen luonteinen, että paras ratkaisu osataan tunnistaa, kun se tulee vastaan.

Leveyshakualgoritmi:

```
MODULE leveyshaku
        Pane alkutila jonoon
        WHILE jono ei ole tyhjä AND lopputilaa ei ole saavutettu DO
                Ota jonosta tila s
                Olkoot s1, ...,sk ne vielä tutkimattomat tilat, joihin s:stä pääsee yhdellä siirrolla
                Aseta osoittimet tiloista s1, ...,sk tilaan s
                Lisää tilat s1, ...,sk jonoon
        ENDWHILE
        IF lopputila saavutettu THEN
                osoitinketju lopputilasta alkutilaan antaa ratkaisun (käänteisessä järjestyksessä)
        ELSE
                ratkaisua ei löydy
        ENDIF
ENDMODULE
```

Syvyyshaku (depth first search) puolestaan on luonteeltaan rohkea ja optimistinen menetelmä. Siinä haku lähtee etukäteen parhaana pidettyyn suuntaan, ja suuntaa muutetaan ainoastaan, jos joudutaan perääntymään. Syvyyshaku käyttää tietorakenteena LIFO-periaatteella toimivaa tilojen pinoa. Syvyyshaku on

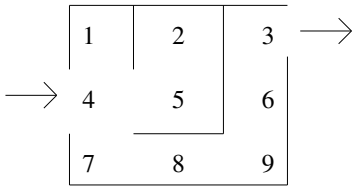
- epätäydellinen: se ei käy läpi koko hakugraafia, mistä syystä se on onnistuessaan nopea;
- epävarma: epätäydellisyyden vuoksi äärettömässä haussa ratkaisu voi jäädä kokonaan löytymättä, vaikka sellainen olisi lähelläkin; ja
- suboptimaalinen: äärellisessäkään haussa ei välttämättä löydetä parasta ratkaisua.

Syvyyshakualgoritmi:

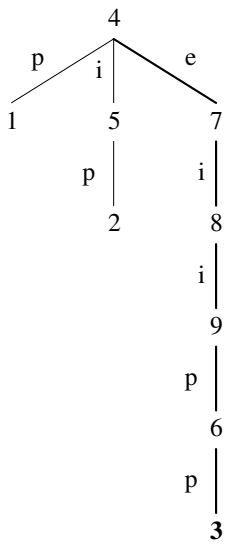
```
MODULE syvyyshaku
    Pane alkutila pinoon
    WHILE pino ei ole tyhjä AND lopputilaa ei ole saavutettu DO
        Ota pinosta tila s
        Olkoot s1, ...,sk ne vielä tutkimattomat tilat, joihin s:stä pääsee yhdellä siirrolla
        Aseta osoittimet tiloista s1, ...,sk tilaan s
        Lisää tilat s1, ...,sk pinoon
    ENDWHILE
    IF lopputila saavutettu THEN
        osoitinketju lopputilasta alkutilaan antaa ratkaisun (käänteisessä järjestyksessä)
    ELSE
        ratkaisua ei löydy
    ENDIF
ENDMODULE
```

Algoritmien ainoa ero on siis niissä käytettävä lista-abstraktio! Kummassakin algoritmissa täytyy vielä tarkentaa se, ts. missä järjestyksessä tilat s_1, ..., s_k viedään listaan. Tätä sekundaarista strategiaa kutsutaan ***heuristiikaksi***. Leveyshaussa heuristiikan merkitys on mitätön – se tarvitaan ainoastaan algoritmin deterministisyyden takaamiseksi. Sen sijaan syvyyshaun hyvyys voi riippua heuristiikasta paljonkin.

Esimerkki. Äärellinen hakuongelma. Tarkastellaan labyrinttitehtävää:



Ongelman tila kuvataan sijaintina labyrintin numeroiduissa ruuduissa. Alkutila on ruutu 4 ja lopputila ruutu 3. Siirrot ovat askelet, jotka vievät tilasta toiseen. Kuvatussa labyrintissa mahdollisia askelia on neljä: ylös/pohjoiseen, oikealle/itään, alas/etelään ja vasemmalle/länteen. Valitaan heuristiikaksi vaikkapa juuri tämä järjestys: p, i, e, l. Heuristiikka määrää ilmansuuntien kokeilujärjestyksen jokaisessa tilassa (siis aina yritetään edetä ensin pohjoiseen, seuraavaksi itään jne.). Ratkaistaan tehtävä ensin leveyshaulla. Muodostuva hakupuu on (lopputila ja ratkaisupolku on vahvennettu):



Jonon kehitys leveyshaussa on seuraava:

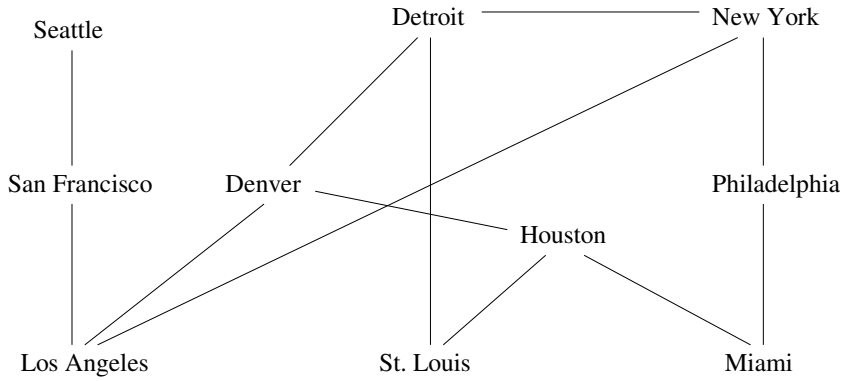
4	poistetaan 4, lisätään 1, 5 ja 7, saadaan:
1, 5, 7	alkion 1 poisto listan alusta, saadaan:
5, 7	alkion 5 poisto alusta ja alkion 2 lisäys listan loppuun, saadaan:
7, 2	
2, 8	
8	
9	
6	
3	

Koko labyrintti siis joudutaan käymään läpi. Kulkureitti labyrintissa nähdään jonon ensimmäisistä alkioista: 4,1,5,7,2,8,9,6,3.

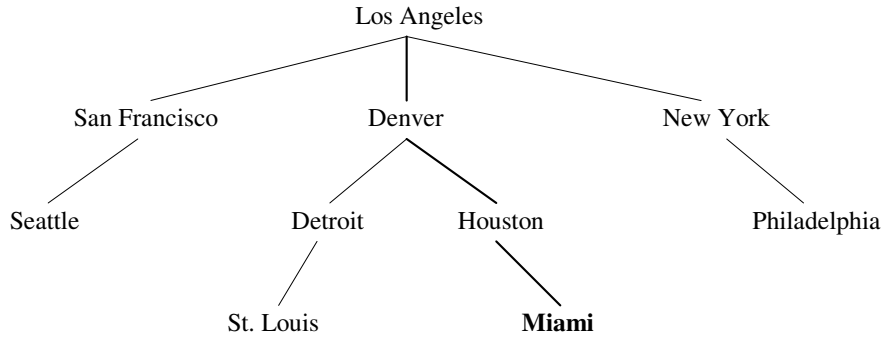
Jos tehtävä ratkaistaan syvyyshaulla samaa heuristiikkaa käyttäen, käydään nytkin koko labyrintti läpi — joskin eri järjestyksessä (kirjoita harjoituksena pinon kehitys näkyviin). Pinon käsittelyssä on huomattava myös se, että alkiot tulee ne laittaa pinoon huonommuusjärjestyksessä, koska tällöin paras jää päällimmäiseksi. Mutta jos heuristiikaksi valitaan e, i, l, p, ratkeaa tehtävä suoraan lyhintä reittiä. Tällöin pinon kehitys on:

4	
7, 5, 1	pinon huippu on ensimmäinen alkio 7, alkion 7 poisto ja alkion 8 lisäys listan alkuun:
8, 5, 1	
9, 5, 1	
6, 5, 1	
3, 5, 1	

Esimerkki. Äärellinen hakuongelma. Olkoot käytettävissä kuvan graafin esittämät lentoreitit. Miten päästään vähimmillä välilaskuilla Los Angelesista Miamiin?



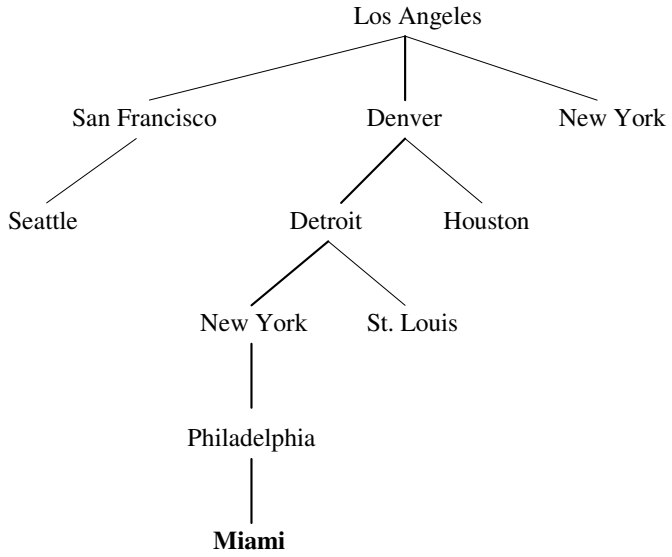
Ongelman tilaesitys on seuraava: tiloina ovat kaupungit, alkutila Los Angeles ja lopputila Miami. Siirrot ovat kaupunkien väliset lennot. Heuristiikka voidaan määritellä nytkin ilmansuuntien mukaan: pyritään ensin pohjoiseen, ja kokeillaan sitten lähinnä myötäpäivään seuraavaa lentoa. Ratkaistaan tehtävä ensin leveyshaulla. Hakupuu:



Jonon kehitys:

Los Angeles
San Francisco, Denver, New York
Denver, New York, Seattle
New York, Seattle, Detroit, Houston
Seattle, Detroit, Houston, Philadelphia
Detroit, Houston, Philadelphia
Houston, Philadelphia, St. Louis
Philadelphia, St. Louis, Miami

Menetelmä siis todella löytää parhaan ratkaisun. Jos hakua jatkettaisiin, löytyisi seuraavalla askelella toinen ratkaisu: Los Angeles – New York – Philadelphia – Miami. Leveyshaku löytää kaikki korkeintaan kolmen lennon ratkaisut ennen ensimmäistäkään neljän tai useamman lennon ratkaisua. Käytetään sitten syvyyshakua samalla heuristiikalla:

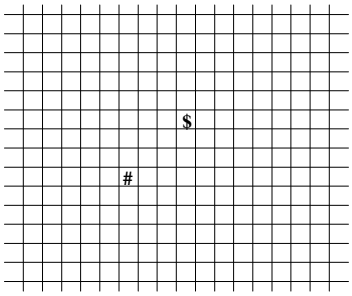


Pinon kehitys:

Los Angeles
San Francisco, Denver, New York
Seattle, Denver, New York
Denver, New York
Detroit, Houston, New York
New York, St. Louis, Houston, New York
Philadelphia, St. Louis, Houston, New York
Miami, St. Louis, Houston, New York

Haku etenee ensin San Franciscoon ja edelleen Seattleen, mutta sitten joudutaan perääntymään takaisin Los Angelesiin. Seuraavaksi koetetusta Denverin suunnasta ratkaisu sen sijaan löytyy – mutta ei paras ratkaisu. Optimaalinen ratkaisu Los Angeles – New York – Philadelphia – Miami löytyisi kyllä syvyyshaullakin, jos vain osattaisiin heti alkutilassa kokeilla ensin New Yorkin suuntaa, mutta haku ei peräänny valitsemaltaan Denverin reitiltä, ellei saavuta umpikujaan. Näin heuristiikan merkitys on syvyyshaussa ratkaiseva.

Esimerkki. Ääretön hakuongelma. Tarkastellaan hakua äärettömässä ruudukossa. Kustakin ruudusta päästään siirtymään mihin tahansa kahdeksasta viereisestä ruudusta. Oletetaan, että aluksi ollaan ruudussa (#). Löydetäänkö ruudussa ($) odottava aarre?



Jos käytetään syvyyshakua, valittu heuristiikka ratkaisee, ts. mihin kahdeksasta ilmansuunnasta lähdetään ensin. Jos valittu suunta on oikea (koillinen), maali löytyy nopeasti. Mutta jos haku lähtee väärään suuntaan, ei lähelläkään sijaitsevaa maaliruutua koskaan saavuteta, koska haku ei äärettömässä hakuavaruudessa koskaan peräänny. Leveyshaku sen sijaan etenee ruudukossa kerroksittain: jokainen korkeintaan n askelen päässä oleva ruutu tutkitaan ennen siirtymistä n+1 askelen päähän. Näin tavoite löytyy varmasti, mutta jos se on vähänkään kauempana, haku voi kestää hyvinkin kauan, ja tavoitteen saavuttaminen voi käytännössä kaatua resurssipulaan. Hakumenetelmän valinta ei siis ole aina helppoa. Kumpaa menetelmää soveltaisit, jos olisit eksynyt sankkaan metsään ja haluaisit löytää takaisin ihmisten ilmoille?

6.1.3 Pelien pelaaminen

Myös pelin pelaaminen on epädeterministinen hakuongelma. Pelaajat tavoittelevat lopputilaa tekemällä tilasta toiseen vieviä siirtoja. Tarkastellaan kahden hengen ns. nollasummapelejä, joissa pelaajat siirtävät vuorotellen ja toisen voitto on toisen häviö. Nyt siirron vaikutus riippuu oleellisesti vastustajan siirroista, eikä hakua kannata tai edes voi ohjata leveys- tai syvyyshaulla, vaan joka tilanteessa on siirtojen hyvyys arvioitava uudelleen. Tällaiseen tehtävään sopii ns. ***heuristinen haku***, jossa kussakin tilassa valitaan siirto, jonka hyvyys arvioidaan parhaaksi. Siirtojen hyvyyttä arvioidaan erityisellä heuristisella ***arviofunktiolla***, joka liittää jokaiseen pelitilanteeseen numeerisen hyvyysarvon. Paras siirto on se, joka johtaa parhaaseen tilaan. Heuristinen haku on tavallaan kompromissi leveys- ja syvyyshaun välillä: tehokkuussyistä käytetään syvyyshakua, mutta sen epävarmuus koetetaan välttää valitsemalla sovellettava sekundaaristrategia joka tilassa uudelleen. Käytettävä heuristiikka ei siis ole kiinteä, vaan se parametrisoidaan tilan suhteen. Hakuavaruus on peleissä usein valtava (käytännössä ääretön, esimerkiksi shakissa peräti luokkaa 10^{120}), joten heuristisen haun menestys riippuu ratkaisevasti arviofunktiosta.

Olkoon f: S $\rightarrow$ **R** arviofunktio. Yleinen heuristinen pelialgoritmi voisi olla seuraavanlainen:

```
MODULE pelaa
        Olkoon alkutila s
        REPEAT
                Olkoot s → s_i (i = 1, ..., k) mahdolliset siirrot
                Valitse siirto siihen tilaan s_i, jolle f(s_i) on suurin
                Odota kunnes vastustaja on siirtänyt
                Merkitse uutta tilaa s:llä
        UNTIL voitto tai tappio
ENDMODULE
```

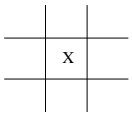
Esimerkki. Jätkänshakki. Jätkänshakissa alkutila=tyhjä ruudukko, muut mahdolliset tilat=kaikki mahdolliset pelitilanteet, lopputila=tilanne, jossa jommalla kummalla pelaajalla on kolmen suora tai ruudukko on täynnä. Pelaajat sijoittavat vuorotellen 3x3-ruudukkoon oman merkkinsä (X tai O). Se voittaa, kumpi ensinnä saa kolme omaa merkkiä samaan linjaan pystysuoraan, vaakasuoraan tai vinoittain. Tarkastellaan peliä pelaajan X kannalta. Numeroidaan ruudukko:

1	2	3
4	5	6
7	8	9

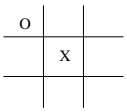
ja määritellään X:n arviofunktio kahden kokonaisluvun erotuksena seuraavasti:

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

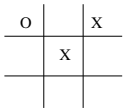
X siirtää ruutuun, jossa funktio saa maksiminsa. Merkitään uutta tilaa sen ruudun numerolla, johon X siirtää. Tarkoituksena on siis löytää sellainen uusi tila s, että $f_X(s)$ on mahdollisimman suuri. Jos X aloittaa, hänen kannattaa siirtää keskimmäiseen ruutuun (5), koska $f_X(5) = 8–4 = 4$; muut siirrot johtavat huonompaan hyvyyteen ($f_X(1) = 8–5 = 3$ ja $f_X(2) = 8–6 = 2$, mikä symmetrian nojalla riittää). Uusi tila on:



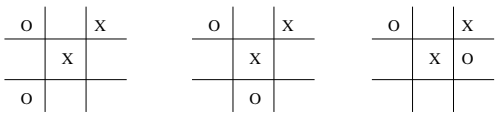
Oletetaan, että pelaaja O käyttää samaa arviofunktiota. Koska f_X arvioi hyvyyttä X:n kannalta, O pyrkii minimoimaan sen arvon. Koska $f_X(1) = f_X(3) = f_X(7) = f_X(9) = 5–4 = 1$ ja $f_X(2) = f_X(4) = f_X(6) = f_X(8) = 6–4 = 2$, O:n kannattaa siirtää johonkin nurkkaan. Siirtäköön hän vasempaan ylänurkkaan:



Seuraavaksi myös X:n kannattaa siirtää johonkin kulmaruutuun, sillä $f_X(3) = f_X(7) = f_X(9) = 5–2 = 3$ ja $f_X(2) = f_X(4) = f_X(6) = f_X(8) = 2$. Valitkoon hän oikean ylänurkan:



Nyt O:n tulisi ehdottomasti valita vasen alanurkka, muutoin hän häviää. Käytetty arviofunktio ei kuitenkaan pysty tätä yksikäsitteisesti osoittamaan, sillä kaikissa seuraavissa vaihtoehdoissa saadaan arviofunktiolle sama minimiarvo $f_X(s) = 1$:



Käytetty arviofunktio ei ole siis riittävän hyvä. Käyttämällä ns. minmax-tekniikkaa päästään parempaan tulokseen.

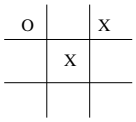
Minmax-tekniikka

Heuristiselle arviofunktiolle asetetaan siis sangen suuret vaatimukset. Pelin parantamiseksi on kaksi vaihtoehtoa: keksitään parempi arviofunktio, tai tarkastellaan tilannetta pidemmälle kuin vain yhden siirron päähän. Jälkimmäinen vaihtoehto vaatii vähemmän luovuutta, joten se sopii koneelle hyvin. Menetelmää kutsutaan ***minmax-tekniikaksi***. Nimitys tulee siitä, että siinä pelaaja valitsee siirron, joka johtaa tilaan, josta vastustajan kannalta parhaan siirron myötä päädytään tilaan, joka on pelaajan kannalta paras. Minmax-tekniikassa siis oletetaan vastustajan pelaavan hyvin, ja pyritään siirtämään siten, että huonoimmassa tapauksessa tilanne on vähiten huono. Minmax-tekniikkaan perustuva parempi pelialgoritmi on:

```
MODULE pelaaparemmin
    Olkoon alkutila s
    REPEAT
        Olkoot s → s_i (i = 1, ..., k) mahdolliset siirrot
        Laske kullekin s_i:lle luku
                g(s_i) = min {f_X(s_ij) | s_i → s_ij on vastustajan mahdollinen siirto}
        Valitse siirto siihen tilaan s_i, jolle g(s_i) on suurin
        Odota kunnes vastustaja on siirtänyt
        Merkitse uutta tilaa s:llä
    UNTIL voitto tai tappio
ENDMODULE
```

Algoritmissa esiintyvä funktio g määrää siis vastustajan edullisimman siirron tilassa s_i. Tällöin pelaajan kannattaa siirtää tilaan, jossa vastustajan paras seuraava siirto on mahdollisimman huono.

Esimerkki. Katsotaan miten käy kesken jääneessä jätkänshakki-pelissä, jos pelaajat käyttävät minmax-tekniikkaa. Tilanne oli siis:



ja O:n siirtovuoro. Muistettakoon, että käytettävä arviofunktio arvioi tilan hyvyyttä X:n kannalta, joten O pyrkii minimien maksimoimisen asemesta minimoimaan arviofunktion maksimin kussakin tilassa. O:lla on kuusi vaihtoehtoa: siirto ruutuun 2, 4, 6, 7, 8 tai 9. Merkitään tilaa, johon päädytään, s_i:llä, i = 2,4,6,7,8,9, ja edelleen tilaa, joka seuraa X:n siirtoa tilassa s_i ruutuun $j \in \{2,4,6,7,8,9\} - \{i\}$, s_{ij}:llä. Lasketaan g:n arvo kullekin s_i:lle:

s_2: $f(s_{24}) = 4–1 = 3$
$f(s_{26}) = 4–2 = 2$
$f(s_{27}) = 4–0 = 4$ maksimi, joten $g(s_2) = 4$
$f(s_{28}) = 4–1 = 3$
$f(s_{29}) = 4–1 = 3$

s_4: $f(s_{42}) = 4–2 = 2$
$f(s_{46}) = 4–2 = 2$
$f(s_{47}) = 4–0 = 4$ maksimi, joten $g(s_4) = 4$
$f(s_{48}) = 4–1 = 3$
$f(s_{49}) = 4–1 = 3$

s_6: $f(s_{62}) = 3–2 = 1$
$f(s_{64}) = 3–1 = 2$
$f(s_{67}) = 3–0 = 3$ maksimi, joten $g(s_6) = 3$
$f(s_{68}) = 3–1 = 2$
$f(s_{69}) = 3–1 = 2$

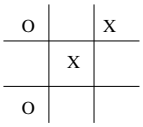
s_7: $f(s_{72}) = 3–2 = 1$
$f(s_{74}) = 3–1 = 2$
$f(s_{76}) = 3–2 = 1$
$f(s_{78}) = 3–1 = 2$
$f(s_{79}) = 3–1 = 2$ maksimi, joten $g(s_7) = 2$

s_8: $f(s_{82}) = 3–2 = 1$
$f(s_{84}) = 3–1 = 2$
$f(s_{86}) = 3–2 = 1$
$f(s_{87}) = 3–0 = 3$ maksimi, joten $g(s_8) = 3$
$f(s_{89}) = 3–1 = 2$

s_9: $f(s_{92}) = 3–2 = 1$
$f(s_{94}) = 3–1 = 2$
$f(s_{96}) = 3–2 = 1$
$f(s_{97}) = 3–0 = 3$ maksimi, joten $g(s_9) = 3$
$f(s_{98}) = 3–1 = 2$

Maksimien minimi on $g(s_7)=2$, joten O:n kannattaa siirtää ruutuun 7. Menetelmä siis todellakin antaa parhaan siirron.

Esitetyn kaltainen minmax-tekniikka ei kuitenkaan vielä takaa, että algoritmia noudattava pelaaja pelaisi parasta mahdollista peliä. Nimittäin seuraavassa tilanteessa



jossa X:llä on siirtovuoro, menetelmä epäonnistuu taas. Mistä apu? On kaksi vaihtoehtoa: minmax-tekniikan laajennus siten, että lasketaan n (n>1) siirtoparin päähän (minmax^n-tekniikka) tai parempi arviofunktio. Yleensä hyvä arviofunktio on vaikea kehittää, jolloin ainoaksi vaihtoehdoksi jää ensimmäinen: raa'an laskentavoiman hyväksikäyttö. Esimerkiksi shakkia ei kyetä algoritmisesti pelaamaan optimaalisella tavalla. Jätkänshakki on kuitenkin sen verran yksinkertainen peli, että parempi arviofunktio on helppo kehittää. Määritellään:

$$f_X^*(s) = \begin{cases} \infty, & \text{jos X:llä on tilassa s todellinen kolmen suora} \\ -\infty, & \text{jos O:llä on tilassa s todellinen kolmen suora} \\ f_X(s), & \text{muulloin} \end{cases}$$

'Äärettömäksi' voidaan valita sopiva hyvin suuri luku. Tällä arviofunktiolla ja yksinkertaisella minmax-menetelmällä jätkänshakkia kyetään pelaamaan optimaalisella tavalla.

6.2 Rinnakkaisalgoritmit

Tähän saakka käsiteltyjen algoritmien yhteydessä on oletettu, että niitä suorittaa yksi prosessori. Kuvatut algoritmit ovat kaikki olleet ***peräkkäisalgoritmeja*** (sequential algorithms). Nykyinen teknologia mahdollistaa myös useampia prosessoreita käyttävien koneiden rakentamisen. Näin saman tehtävän suoritus voidaan jakaa useammalle prosessorille, jotka suorittavat samaa tehtävää (tai tehtävän eri osia) samanaikaisesti rinnakkain. Tällöin puhutaan ***rinnakkaislaskennasta***. Jos tehtävän jakaminen osatehtäviin on mahdollista, tehtävän suorittaminen on nopeampaa, koska tehtävän suorittamiseen kuluva aika on suurimman osatehtävän suorittamiseen kuluvan ajan mittainen. Algoritmeja, joiden jakaminen osatehtäviin on mahdollista edellä kuvatulla tavalla, nimitetään ***rinnakkaisalgoritmeiksi*** (parallel algorithms).

6.2.1 Rinnakkaistaminen

Joskus tehtävä vaatii modifiointia rinnakkaiskäsittelyn mahdollistamiseksi. Puhutaan tehtävän ***rinnakkaistamisesta***. Jotkut tehtävät rinnakkaistuvat helposti, toiset vaikeammin. Jos rinnakkaistehtävän ratkaisussa joudutaan jossain vaiheessa käyttämään edellisen vaiheen tuloksia hyväksi, prosessorien on odotettava toisiaan saadakseen edellisen vaiheen tulokset käyttöön. Tällöin prosessorien on toimittava synkronisesti. Joskus tehtävän rinnakkaistaminen vaatii kokonaan uuden lähestymistavan käyttämistä, eikä rinnakkaisalgoritmi välttämättä lainkaan muistuta saman tehtävän ratkaisevaa peräkkäisalgoritmia. Myöskään tavalliset ohjelmointikielet eivät mahdollista rinnakkaisuuden kuvaamista kovin luonnollisella tavalla. Rinnakkaisohjelmointiin on olemassa omia erikoiskieliä, mutta niiden yleistymistä vaikeuttaa mm. se, että eri kielet ovat sitoutuneet erilaisiin rinnakkaisarkkitehtuureihin. Tässä yhteydessä käsitellään rinnakkaisuutta vain periaatteen tasolla kahden yksinkertaisen esimerkin avulla, lukuisiin ei-triviaaleihin yksityiskohtiin kajoamatta. Pseudokieleen tehdään vain yksi laajennus:

```
FOR i := a, ..., b PARDO <runko> ENDFOR
```

tarkoittaa, että silmukan runko suoritetaan kullakin i:n arvolla rinnakkain (yhtäaikaa).

Esimerkki. Kahden samanpituisen kokonaislukuvektorin erotus vastinalkioittain.

```
X      =    (12   23   11   9    -11   34   56   101)
Y      =    (0    1    200  200   2    4    2    49)
X – Y  =    (12   22   -189 -191 -13   30   54   52)
```

Tehtävän ratkaiseva peräkkäisalgoritmi:

```
MODULE peräkkäiserotus(vektorit X ja Y) RETURNS vektorin X–Y
        FOR k := 1, ..., X.length DO
                E[ k ] := X[ k ] – Y[ k ]
        ENDFOR
        RETURN E
ENDMODULE
```

Olkoon X.length=Y.length=n. Peräkkäisalgoritmin suoritusaika on suoraan verrannollinen vektorien pituuteen: erotus lasketaan n kertaa. Jos käytettävissä on n prosessoria, jokaista vähennyslaskua kohden on käytettävissä oma prosessorinsa ja n laskutoimitusta voidaan suorittaa yhtä aikaa. Näin saadaan seuraava rinnakkaisalgoritmi, joka on lähes identtinen peräkkäisalgoritmin kanssa:

```
MODULE rinnakkaiserotus(vektorit X ja Y) RETURNS vektorin X–Y
        FOR k := 1, ..., X.length PARDO
                E[ k ] := X[ k ] – Y[ k ]
        ENDFOR
        RETURN E
ENDMODULE
```

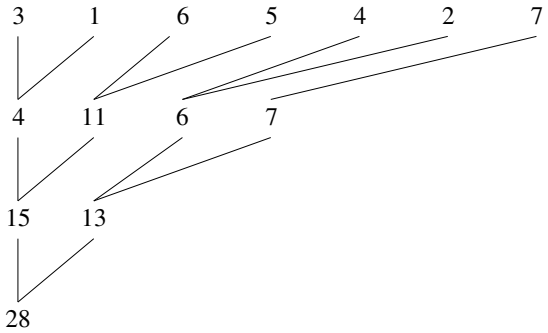
Tässä indeksi k viittaa sekä vektorin k:nteen alkioon että k:nteen käytössä olevaan prosessoriin. Rinnakkaisalgoritmin suoritusaika ei riipu n:stä: kukin prosessori k (k = 1, ..., n) suorittaa yhden vähennyslaskun. Näin n-kertaistamalla prosessorien määrä algoritmi saadaan suoritettua periaatteessa n kertaa nopeammin eli vakioajassa. Käytännössä suoritusaikasäästö on pienempi, sillä aikaa kuluu prosessorien väliseen kommunikointiin.

Esimerkki. Vektorin alkioiden summan laskeminen. Peräkkäisalgoritmi:

```
MODULE peräkkäissumma(vektori V) RETURNS V:n alkioiden summa
        s := 0
        FOR k := 1, ..., V.length DO
                s := s + V[ k ]
        ENDFOR
        RETURN s
ENDMODULE
```

Olkoon V.length=n. Tämänkin peräkkäisalgoritmin suoritusaika on suoraan verrannollinen n:n arvoon: summa lasketaan n kertaa. Tarkastellaan sitten rinnakkaisalgoritmia: Ideana on laskea lukuja yhteen pareittain. Ensimmäisellä kierroksella summataan $\lfloor n/2 \rfloor$ lukuparia yhtäaikaa $\lfloor n/2 \rfloor$ prosessoria käyttäen ($\lfloor n/2 \rfloor$ tarkoittaa pyöristystä alaspäin). Jos n on pariton, niin viimeinen luku kopioidaan sellaisenaan seuraavalle kierrokselle, johon siis tulee $\lceil n/2 \rceil$ (pyöristys ylöspäin) lukua summattavaksi. Pareja on tällöin $\lfloor \lceil n/2 \rceil /2 \rfloor$ ja tuloslukuja $\lceil n/4 \rceil$. Näin jatketaan, kunnes summattavia lukupareja on jäljellä yksi ja summaus voidaan suorittaa yhdellä prosessorilla. Koska yhteenlaskettavien parien lukumäärä (lähes) puolittuu joka kierroksella, joudutaan n luvun yhteenlaskemiseksi suorittamaan log n kierrosta. Myös prosessoreiden määrä puolittuu joka vaiheessa, joten prosessoreita tarvitaan eniten ensimmäisellä kierroksella, n/2 prosessoria.

Olkoon n = 7 ja V = [3 1 6 5 4 2 7]. 1. kierroksen jälkeen V = [4 11 6 7]. 2. kierroksen jälkeen V = [15 13]. 3. kierroksen jälkeen V = [28], jolloin vastaus = V[1] = 28. Muodostettavan (alkuperäistä puolta lyhyemmän) vektorin viimeinen alkio on siis laskettava eri tavalla kuin muut, jos yhteenlaskettavien määrä on sillä kierroksella pariton.



Rinnakkaisalgoritmi:

```
MODULE rinnakkaissumma(vektori V) RETURNS V:n alkioiden summa
    n := V.length
    FOR k := 1, ..., log2 n DO
        FOR i := 1, ..., ⌈n/2⌉ PARDO
            IF (i = ⌈n/2⌉) AND (n pariton) THEN
                V[ i ] := V[ n ]
            ELSE
                V[ i ] := V[ 2*i – 1 ] + V[ 2*i ]
            ENDIF
        ENDFOR
        n := ⌈n/2⌉
    ENDFOR
ENDMODULE
```

Koska kunkin kierroksen tulos on seuraavan kierroksen syöttötietoa, uusi kierros ei voi alkaa, ennen kuin edellinen päättyy. Prosessorien on siis työskenneltävä synkronisesti. Algoritmin suoritusaika on suunnilleen se aika, joka kuluu yhdeltä prosessorilta algoritmin suorittamiseen. Suoritusaika on siis T(n) ~ log n. Jos peräkkäisalgoritmi tarvitsee 1000 aikayksikköä 1000 luvun summaamiseen, rinnakkaisalgoritmi tarvitsee vain log 1000 = 10 aikayksikköä, mutta 500 prosessoriyksikköä. Jos yhteenlaskettavia lukuja on 1 000 000, rinnakkaisalgoritmi on jopa 50 000 kertaa nopeampi kuin peräkkäisalgoritmi, mutta toisaalta prosessoreita tarvitaan peräti 500 000.

6.2.2 Rinnakkaisalgoritmien kompleksisuus

Perinteisesti kompleksisuusteoria on keskittynyt peräkkäisalgoritmien resurssitarpeiden suoritusajan ja muistitilan arviointiin. Rinnakkaisuuden yleistyessä kompleksisuusteoriassa on kiinnostuttu myös rinnakkaisalgoritmien resurssitarpeista: suoritusajasta ja prosessorien lukumäärästä, 'prosessorikompleksisuudesta'. Kompleksisuuteen vaikuttavat myös prosessorien väliset kytkennät toisiinsa. Rinnakkaiskoneilla saavutettava nopeus ei ole ilmaista. Rinnakkaisalgoritmien kompleksisuustarkastelussa rinnakkainen aika ja prosessorien määrä käyvät kauppaa keskenään. Laskentanopeudessa ja kustannuksissa pyritään saavuttamaan tasapaino.

Naiivin lajittelun aikakompleksisuus on neliöllinen, ja parhaiden peräkkäisalgoritmien (esimerkiksi limityslajittelu) aikakompleksisuus on T(n) ~ n log n. Parhaan rinnakkaisen lajittelualgoritmin aikakompleksisuus on log n, jolloin tarvitaan n log n prosessoria. Näin paras rinnakkainen lajittelualgoritmi on aikakompleksiuudeltaan n kertaa parempi kuin paras peräkkäisalgoritmi, mutta prosessorikompleksisuudeltaan n log n-kertainen. Esimerkiksi Kiinan miljardiväestön kirjanpitäjä saisi lajitteluun kuluvan ajan pudotettua murto-osaan nykyisestä, mutta toisaalta prosessoreita tarvittaisiin valtava määrä.

Rinnakkaiskoneissa on paljon prosessoreita, kuten nykyisissä peräkkäiskoneissa on paljon muistia. Niissä voi olla miljoonia prosessoreita, joista kukin suorittaa jotakin

ongelman osaa samanaikaisesti. Tällaisten koneiden rakentaminen on tällä hetkellä mahdollista, mutta melko epätaloudellista. Rinnakkaiskonetta voidaan sikäli verrata ihmiseen, että ihmisaivoissa on ilmeisesti biljoonia aktiivisia prosessointiyksiköitä, jotka toimivat rinnakkain. Nykyisillä koneilla rinnakkainen lajittelu ei vielä ole realistinen vaihtoehto, mutta kunhan yksinkertaisia prosessoreita kyetään valmistamaan massoittain (kuten muistia tänään), tilanne voi muuttua nopeastikin.

Synkronisesti toimivat rinnakkaisprosessorit suorittavat laskentansa askel askeleelta yhteisymmärryksessä. On mahdollista rakentaa myös asynkronisia koneita, mutta synkroniset koneet ovat helpompia suunnitella ja rakentaa sekä ohjelmoida, koska ei tarvitse huolehtia ajoitusongelmista tai koneiden välisestä kommunikoinnista.

Kaikkien rinnakkaiskoneiden arkkitehtuuri on läheisessä suhteessa laskennalliseen kykyynsä. Valittaessa tietty rinnakkaiskonearkkitehtuuri päähuomiot ovat taloudellisia tai suhteessa ohjelmoinnin helppouteen. Annettaessa sama määrä prosessoreita yhdellä rinnakkaiskoneella suoritettavaa algoritmia voidaan simuloida toisella rinnakkaiskoneella samassa ajassa. Rinnakkainen aika on toisaalta verrannollinen peräkkäislaskennassa käytetyn muistin kanssa. Toisin sanoen mikä tahansa ongelma, joka voidaan ratkaista peräkkäiskoneella käyttäen vähän muistia, voidaan ratkaista rinnakkaiskoneella käyttäen vähän aikaa.

Yleisesti uskotaan, että samoin kuin on olemassa luonnostaan rekursiivisia ongelmia, on olemassa ***luonnostaan peräkkäisiä ongelmia*** (inherently sequential problems), joiden ratkaisua rinnakkaiskoneet eivät nopeuta. Tätä uskomusta ei kuitenkaan toistaiseksi ole todistettu.

6.3 Vapaajärjesteinen ohjelmointi

Tähän saakka esitetyt algoritmit ovat olleet ***proseduraalisia*** eli ***imperatiivisia***. Proseduraalisille algoritmeille on ominaista, että algoritmi esitetään toimintosarjana, proseduurina, jota seuraamalla haluttu tehtävä suoritetaan. Proseduraalisissa algoritmeissa kuvataan sekä ne toimenpiteet, jotka prosessorin tulee suorittaa, että toimenpiteiden suoritusjärjestys. Proseduraaliset algoritmit ovat dominoineet tietojenkäsittelytiedettä yli 60 vuoden ajan. Syy tähän on ilmeinen: proseduraaliset määritykset vastaavat hyvin tarkasti tavallisen von Neumann -tietokoneen toimintaa – tietokonehan suorittaa primitiivisiä operaatioita ohjelman deterministisesti määräämässä järjestyksessä.

Proseduraalisissa algoritmeissa voidaan havaita seuraavia puutteita:

- Algoritmia suunniteltaessa on kiinnitettävä huomiota algoritmin suoritusjärjestykseen: ehtolauseiden oikeaan sisentämiseen, toistojen oikea-aikaiseen lopettamiseen, toistolaskureista huolehtimiseen jne. Nämä lisäävät algoritmin pituutta ja mutkikkuutta sekä virhetodennäköisyyttä.
- Proseduraaliset ohjelmat ovat matemaattisesti kömpelöitä, joten suurten ohjelmien oikeaksi todistaminen on hyvin vaikeaa. Tähän ovat suurimpana syynä käskyjen ***sivuvaikutukset***. Asetuslause on erittäin keskeisessä asemassa proseduraalisessa ohjelmoinnissa, mutta se aiheuttaa muuttujien ns. ***läpinäkymättömyyden***: saman muuttujan eri esiintymät eivät välttämättä edusta samaa arvoa, mikä on matemaattisen todistustekniikan perusta.

Proseduraalisten algoritmien heikkouksien kumoamiseksi on kehitelty uusia ***vapaajärjesteisiä*** algoritmien määrittelytapoja. Ideana on, että ohjelmoija keskittyy tehtävän oikean ratkaisun määrittelemiseen, eikä hänen tarvitse välittää siitä, miten kone tuon oikean ratkaisun löytää. Tästä syystä vapaajärjesteistä, ei-proseduraalista ohjelmointia kutsutaan myös ***deklaratiiviseksi***, määritteleväksi ohjelmoinniksi. Vapaajärjesteiset algoritmit ovat epädeterministisiä: ohjelmoijan ei (periaatteessa) tarvitse tietää, miten algoritmin suoritus missäkin vaiheessa etenee. Ratkaisun etsiminen muodostaa epädeterministisen hakuprobleeman. Ratkaisua etsimiseen käytetään jotakin determinististä hakumenetelmää, usein syvyyshakua. Epädeterministisen luonteensa ansiosta vapaajärjesteiset algoritmit soveltuvat myös erittäin hyvin rinnakkaislaskentaan.

Vapaajärjesteinen ohjelmointi voidaan jakaa kahteen paradigmaan: matemaattisen funktion käsitteeseen perustuvaan ***funktionaaliseen ohjelmointiin*** (functional programming) ja matemaattisen relaation käsitteeseen perustuvaan ***logiikkaohjelmointiin*** (logic programming). Molemmissa oppisuunnissa tiedon rakenteen esittämiseen käytetään erittäin joustavia dynaamisia listarakenteita, yleistettyjä listoja.

6.3.1 Yleistetyt listat

Normaalisti listan alkiot ovat keskenään samaa tyyppiä (kokonaislukulista, sanalista, ...). Yleistetyssä listassa tätä vaatimusta ei ole, ja lisäksi listat voivat olla hierarkkisia (rekursiivisia), jolloin listan alkioina voi esiintyä alilistoja. Listavakiot esitetään luettelemalla listan ja sen alilistojen alkiot hakasuluissa, esimerkiksi:

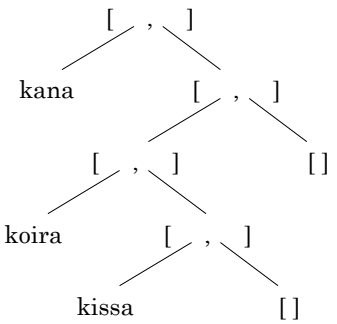
```
[koira kissa kana]
[3 7 14 2]
[norsu]
[tässäkin on [hieman monimutkaisempi] lista]
[ ]
```

Yleistetyt listat ovat rakenteeltaan kaksiosaisia: listan ensimmäistä alkiota sanotaan listan ***pääksi*** (head) ja listan muiden kuin ensimmäisen alkion muodostamaa alilistaa listan ***hännäksi*** (tail). Funktionaalista merkintää käyttäen esimerkiksi

```
head([koira kissa kana]) = koira
tail([koira kissa kana]) = [kissa kana]
head([14 23]) = 14
head([norsu]) = norsu
tail([norsu]) = [ ]
```

Graafisesti tällainen lista voidaan esittää binääripuuna, jossa kukin sisäsolmu edustaa listaa, jonka pää on solmun vasen alipuu ja häntä on oikea alipuu.

Esimerkki. [kana, [koira, kissa]]:



Määritelmä. ***Lista*** on joko

- ***tyhjä lista*** [ ] tai
- pari ***pää*** | ***häntä***, missä pää on listan ensimmäinen alkio ja häntä listan muiden alkioiden muodostama lista.

Erityisesti on ymmärrettävä, että listan pää on siis aina alkuperäisen listan alkio (ei välttämättä atominen), kun taas listan häntä on aina alkuperäisen listan alilista (mahdollisesti tyhjä lista). Merkintä X | L tarkoittaa listaa, jonka pää on alkio X ja häntä on lista L. Esimerkiksi

koira | [kissa kana] = [koira kissa kana]
norsu | [ ] = [norsu]
8 | 3 | 15 | [ ] = 8 | 3 | [15] = 8 | [3 15] = [8 3 15]

Jokaiselle listalle L ja alkiolle X on voimassa

head(X | L) = X
tail(X | L) = L

ja jos L ei ole tyhjä

head(L) | *tail*(L) = L.

Operaattori | on itse asiassa funktion *cons* lyhennysmerkintä: *cons*(*head*(L), *tail*(L)) = L. Funktio *cons* on binääristen listojen tärkein ja primitiivisin listakonstruktori.

6.3.2 Funktionaalinen ohjelmointi

Funktionaalista ohjelmointiajattelua sisältyy jo imperatiiviseen ohjelmointiin, jota tarkasteltiin monisteen toisessa luvussa. Puhdas funktionaalinen ohjelmointi perustuu Churchin λ-kalkyyliin (lambda calculus). Algoritmin tulos määritellään syöttötietojensa matemaattisena funktiona. Esimerkiksi algoritmi, joka laskee listassa olevien lukujen summan, ymmärretään funktiona, joka muuntaa lukujen muodostaman listan (syötteen) yhdeksi luvuksi (tuloste). Funktionaalisen algoritmin suunnittelussa yritetään siis saada määriteltyä funktio, joka on ongelman ratkaisu. Ohjelma on siis funktionaalinen abstraktio, ja ohjelman suoritus tarkoittaa tämän funktion soveltamista syöttötietoon.

Esimerkki. Listassa olevien lukujen summa. Lukujen summa voitaisiin laskea seuraavanlaisella sum-funktiolla:

```
sum(L) =
        if L = [ ] then 0
        else add(head(L), sum(tail(L)))
add(x, y) = x + y
```

Huomaa, että funktio määritellään rekursiivisesti. Rekursiolla on keskeinen rooli funktionaalisessa ohjelmoinnissa – se vastaa toistoa proseduraalisessa ohjelmoinnissa.

Lukujen 3, 4 ja 5 summan suoritus:

```
sum([3 4 5])
        = add(3, sum([4 5]))
        = add(3, add(4, sum([5])))
        = add(3, add(4, add(5, sum([ ]))))
        = add(3, add(4, add(5, 0)))
        = add(3, add(4, 5))
        = add(3, 9)
        = 12
```

Listassa L olevien lukujen tulon laskenta voidaan määritellä vastaavasti:

```
mult(L) =
        if L = [ ] then 1
        else times(head(L), mult(tail(L)))
times(x, y) = x * y
```

Edelleen on mahdollista määritellä yleinen listan redusointifunktio, joka toimii *sum*- ja *mult*-funktioiden tavoin redusoiden listan yhdeksi alkioksi jollakin operaatiolla *op*. Määriteltäessä redusointifunktiota *reduce* määritellään operaatio *op* (edellä *add* tai *times*), ja operaation tulos *base* kohdistettaessa se tyhjään listaan (edellä 0 tai 1):

```
reduce(op, base, L) =
        if L = [ ] then base
        else op(head(L), reduce(op, base, tail(L)))
```

Nyt voidaan *sum*- ja *mult*-funktiot määritellä *reduce*-funktion avulla seuraavasti:

```
sum(L) = reduce(add, 0, L)
mult(L) = reduce(times, 1, L)
```

Funktio *largest*, joka etsii positiivisia lukuja sisältävän listan suurimman alkion, voidaan määritellä seuraavasti:

```
largest = reduce(max, 0, L)
max(x, y) = if x > y then x else y.
```

Funktionaalinen ohjelma muodostuu funktionaalisella ohjelmointikielellä kirjoitetuista funktioista. Vanhin ja tunnetuin funktionaalinen ohjelmointikieli on Lisp, muita ovat esimerkiksi ML, Hope, Miranda ja (osittain) Logo.

Esimerkki. Hypoteettisella kielellä kirjoitettu funktionaalinen ohjelma, joka lajittelee sanalistan aakkosjärjestykseen.

```
sort(L) =                               (* palauttaa listan lajiteltuna *)
        if L = [ ] then [ ]
        else insert(head(L), sort(tail(L)))
insert(word, L)=                        (* palauttaa listan, jossa sana on lisätty oikealle paikalleen *)
        if L = [ ] then word | [ ]
        else if precedes(word, head(L)) then word | L
                else head(L) | insert(word, tail(L))
precedes(word1, word2) = word1 < word2      (* palauttaa totuusarvon *)
```

Prosessori suorittaa funktionaalista ohjelmaa soveltamalla jotakin funktiota syöttötietoihin (esimerkiksi sanalistaan). Ennen funktion soveltamista argumentteihinsa prosessori saattaa sieventää argumentteja soveltamalla niihin jotakin määriteltyä funktiota. Prosessi jatkuu, kunnes sievennystä ei voida enää jatkaa, ts. mitään funktiota ei voida enää soveltaa. Esimerkiksi sanalistan [koira kissa sika] lajittelu voisi tapahtua seuraavasti:

```
sort  ([koira kissa sika])
        = insert (head ([koira kissa sika]), sort (tail ([koira kissa sika])))
        = insert (koira, sort (tail ([koira kissa sika])))
        = insert (koira, sort ([kissa sika]))
        = insert  (koira, insert ( head ( [kissa sika]), sort (tail ([kissa sika]))))
        = insert (koira, insert (kissa, sort (tail ([kissa sika]))))
        = insert (koira, insert (kissa, sort ([sika])))
        = insert (koira, insert (kissa, insert (head ([ sika], sort ([ ]))))
        = insert (koira, insert (kissa, insert (sika, sort ([ ]))))
        = insert (koira, insert (kissa, insert (sika, [ ])))
        = insert (koira, insert (kissa, sika | [ ]))
        = insert (koira, kissa | sika | [ ])
        = head (kissa | sika | [ ]) | insert (koira, tail (kissa | sika | [ ]))
        = kissa | insert (koira, tail (kissa | sika | [ ]))
        = kissa | insert (koira, sika | [ ])
        = kissa | koira | sika | [ ]
        = [kissa koira sika]
```

Funktionaalisen ohjelman suoritusjärjestys on riippumaton ohjelman funktioiden määrittelyjärjestyksestä: edellä olevassa esimerkissä ***sort***-, ***insert***- ja ***precedes***- funktiot olisi voitu määritellä missä järjestyksessä hyvänsä. Prosessori selvittää suoritusjärjestyksen käyttämällä jotakin yleistä strategiaa etsiessään yksinkertaistettavia argumentteja ja sovellettavia funktioita. Näin funktionaalinen ohjelmointi poistaa ohjelmoinnista proseduraalisen ohjelmoinnin epäkohdan.

Esimerkkiohjelman suorituksen ei tarvitse edetä juuri esitetyllä tavalla. Ohjelman suoritusjärjestys ei vaikuta ohjelman lopputulokseen. Tämä väite perustuu Church-Rosserin kalkyylia koskevaan teoreemaan, jonka mukaan järjestys, jossa argumentteja sievennetään ja funktioita sovelletaan, ei vaikuta lopputulokseen. Kuitenkin tällä järjestyksellä voi olla vaikutusta siihen, miten nopeasti ratkaisu löytyy tai löytyykö se lainkaan. Edellisessä esimerkissä prosessori valitsi kolmannen askeleen jälkeen sievennettäväkseen ***sort***-funktion kutsun *sort*([kissa sika]), vaikka se olisi voinut valita funktion ***insert***, jolloin ohjelman suoritus olisi alkanut seuraavasti:

```
sort ([koira kissa sika])
        = insert (head ([koira kissa sika]), sort (tail ([koira kissa sika])))
        = insert (koira, sort (tail ([koira kissa sika])))
        = insert (koira, sort ([kissa sika]))
        = if precedes (koira, head (sort ([kissa sika]))) then koira | sort ([kissa sika])
                else head (sort ([kissa sika])) | insert (koira, tail (sort ([kissa sika])))
        = ...
```

Tämä suoritus on työläämpi, koska prosessori joutuu soveltamaan *sort*-funktiota useita kertoja listaan [kissa sika] ennen kuin lopputulos selviää. Koska prosessorin valitsema funktioiden suoritusjärjestys vaikuttaa suoritusaikaan, valintastrategioiden kehittämiseen on kiinnitetty paljon huomiota, mutta toistaiseksi mitään ehdottoman optimaalista ratkaisua ei ole löydetty.

Yhteenveto:

- Ohjelmoijan ei tarvitse määritellä funktionaalisen ohjelman suoritusjärjestystä, koska tämän määrää prosessorin strategia argumenttien yksinkertaistamiseksi ja funktioiden valitsemiseksi.
- Prosessorin valitsema funktioiden suoritusjärjestys ei vaikuta ohjelman lopputulokseen, mutta suoritusaikaan sillä voi olla vaikutusta.

- Funktionaalinen ohjelma soveltuu luonnollisella tavalla rinnakkaislaskentaan, koska ohjelman eri funktioita voidaan soveltaa samanaikaisesti.

Huolimatta funktionaalisen ohjelmoinnin monista eduista se ei ole saavuttanut suurta suosiota, etenkään Euroopassa. Yhdysvalloissa Lisp-kielellä on kyllä vankka jalansija tietyillä lähinnä tekoälyn piiriin luettavilla erityissovellusaloilla. Tärkein funktionaalisen ohjelmoinnin vaatimattoman suosion syy lienee se, että nykyisiä tietokoneita ei ole suunniteltu toteuttamaan funktionaalisia ohjelmia suoraan. Kun funktionaalinen ohjelma on käännetty suorituskelpoiseksi imperatiiviseksi ohjelmaksi, käännöksen tulos on usein hyvin monimutkainen ja hidas suoritettava.

6.3.3 Logiikkaohjelmointi

Logiikkaohjelmointi on viime vuosisadan alkupuoliskolla kehitettyyn ***predikaatti-kalkyyliin*** perustuva deklaratiivinen ohjelmointimenetelmä. Siinä ongelma kuvataan epädeterministisesti predikaattilogiikan avulla, ja ohjelman suoritus tapahtuu yleensä syvyyshaun ohjaamana päättelynä. Logiikkaohjelmointi poikkeaa tavanomaisesta imperatiivisesta ohjelmointiajattelusta funktionaalista ohjelmointiakin enemmän; onhan relaatio funktiotakin vieraampi käsite konventionaalisessa ohjelmoinnissa. Logiikka-ohjelmalla on tavallaan kaksinkertainen merkitys ─ sekä deklaratiivinen että proseduraalinen semantiikka: ohjelmoija tulkitsee ohjelmaa deklaratiivisesti ja kone proseduraalisesti. Deklaratiivinen tarkastelukulma sopii ongelman määrittelyyn mainiosti, mutta valitettavasti deklaratiivisen näkökulman yhteensovittaminen proseduraalisen kanssa ei aina onnistu täydellisesti.

Logiikkaohjelma samoin kuin ratkaistava ongelmakin esitetään tietynmuotoisina predikaattilogiikan kaavoina, ns. ***klausuuleina*** (clauses). Klausuulit ovat käännettyjä implikaatioita, ns. konditionaaleja. Tällöin implikaationuoli $\rightarrow$ korvataan symbolilla $\leftarrow$. Klausuuleja on kolmea tyyppiä:

1. Positiiviset klausuulit eli ***faktat***:
   $p \leftarrow$
2. Ehtoklausuulit eli ***säännöt***:
   $p \leftarrow q_1, q_2, \ldots q_n$
3. Negatiiviset klausuulit eli ***tavoitteet***:
   $\leftarrow q_1, q_2, \ldots q_m$

Konditionaalisymbolin ($\leftarrow$) vasenta puolta sanotaan klausuulin ***pääksi*** (head) ja oikeaa puolta ***rungoksi*** (body). Faktat ovat siis klausuuleja, joissa on tyhjä runko, ja tavoitteet ovat klausuuleja, joissa on tyhjä pää.

Tarkasteltavaa ongelma-aluetta koskeva tietämys esitetään sääntöinä ja faktoina, joita käytetään apuna tavoitteen todistamisessa. Faktat ovat tapauskohtaisia tosiasioita, ns. ***ekstensionaalista*** tietämystä. Säännöt puolestaan kuvaavat yleisiä lainalaisuuksia, ns. ***intensionaalista*** tietämystä, jonka avulla voidaan johtaa uusia faktoja. Jotakin aihealuetta koskeva (sääntöinä ja tosiasioina, so. logiikkaohjelmana esitetty) tietämys muodostaa ***tietämyskannan***. Ongelma esitetään mahdollisesti useasta osasta koostuvana ***tavoitteena*** (goal), joka pyritään tietämyskannan avulla osoittamaan todeksi tai epätodeksi.

Deklaratiivisen semantiikan mukaan konditionaalinuoli luetaan 'jos' ja pilkku luetaan 'ja'.

Esimerkki. Klausuuli

lintu (X) ← munii (X), siivekäs (X)

tulkitaan seuraavasti: "Mikä tahansa X on lintu, **jos** X munii **ja** X:llä on siivet". Symboli X on muuttuja, joka edustaa tässä tapauksessa mitä tahansa eläintä tai yleisemmin mitä tahansa objektia.

Logiikkaohjelmaa suorittaessaan prosessori yrittää osoittaa tavoitteen todeksi käyttäen annettuja klausuuleita systemaattisesti hyväkseen. Päättely noudattaa seuraavia sääntöjä:

1. Faktat ovat aina tosia.
2. Säännön pää toteutuu, jos sen rungon tavoitteet toteutuvat.
3. Tavoite toteutuu, jos sen jokainen osatavoite samastuu
   - faktan kanssa, tai
   - sellaisen säännön pään kanssa, jonka runko toteutuu.

Päättelyn suorituksessa säännössä tai tavoitteessa oleva muuttuja voidaan korvata vakioilla, joita kutsutaan muuttujan ***ilmentymiksi*** tai ***esiintymiksi*** (instances). Korvauksen avulla tavoite pyritään ***samastamaan*** jonkin faktan tai säännön pään kanssa. Esimerkiksi muuttujan X ilmentymä voi olla koira, kana tai sika. Jotta ohjelman muuttujat ja esiintymät erotetaan toisistaan, kirjoitetaan muuttujat isolla alkukirjaimella ja vakiot pienillä kirjaimilla. Klausuulin

lintu (X) ← munii (X), siivekäs (X)

proseduraalinen tulkinta on seuraava: proseduurin *lintu*(X) suorittamiseksi (eli sen osoittamiseksi, että X on lintu) on suoritettava proseduuri *munii*(X) (eli osoitettava, että X munii) ja proseduuri *siivekäs*(X) (eli osoitettava, että X:llä on siivet). Seuraavassakin esimerkissä X tarkoittaa eläintä:

matelija (X) ← munii (X), suomukas (X)

Klausuuli tulkitaan seuraavasti: "X on matelija, jos X munii ja jos X:llä on suomut" tai "sen osoittamiseksi, että X on matelija, on osoitettava, että X munii ja että X:llä on suomut".

Faktoista jätetään yleensä konditionaalinuoli pois, jolloin faktoja ovat esimerkiksi:

suomukas (python)
munii (kana)
siivekäs (kana)

Tavoite voisi olla esimerkiksi:

← lintu (kana)

mikä tarkoittaa "Onko kana lintu?". Usein tavoitteen nuoli korvataan kysymysmerkillä:

? lintu (kana)
? matelija (python)

Esimerkki. Logiikkaohjelma.

(1) lintu (X) ← munii (X), siivekäs (X)
(2) munii (kana)
(3) siivekäs (kana)

Tässä ohjelmassa on kolme klausuulia, joista kaksi jälkimmäistä on faktoja. Klausuulit on numeroitu myöhempiä viittauksia varten. Ohjelmaa voidaan käyttää esimerkiksi osoittamaan seuraava tavoite todeksi (tai epätodeksi):

? lintu (kana)

Prosessori suorittaa ohjelmaa yrittämällä päästä tavoitteeseen klausuulien avulla. Suoritusvaiheen alussa se tutkii, vastaavatko tavoite ja jonkin klausuulin pää toisiaan. Jos tavoite samastuu johonkin faktaan, tavoite tulee osoitetuksi, ja ohjelman suoritus päättyy. Jos tavoite samastuu jonkin ehtoklausuulin päähän, tavoite korvautuu tämän klausuulin rungolla, josta tulee uusi tavoite.

Kun esimerkin ohjelmaa suoritetaan tavoitteena ? *lintu*(kana), prosessori tutkii, vastaavatko tavoite ja ohjelman jonkin klausuulin pää toisiaan. Ainoa klausuulin pää, joka vastaa tavoitetta, on ensimmäisen klausuulin *lintu*(X). Klausuulin muuttuja X korvataan ilmentymällä kana, jolloin tavoite ja säännön (1) pää samastuvat ja säännön (1) rungosta saadaan uusi tavoite (nyt X = kana):

? munii (kana), siivekäs (kana)

Näin alkuperäinen tavoite "Onko kana lintu?" muuttui kaksiosaiseksi tavoitteeksi "Muniiko kana ja onko kanalla siivet?" Seuraavaksi prosessori yrittää toteuttaa kunkin osatavoitteen. Se tutkii, vastaako osatavoite jotakin klausuulia. Tässä tapauksessa tavoitteen ensimmäinen osa samastuu faktaan (2), jolloin tavoite redusoituu muotoon ? *siivekäs*(kana). Tämä uusi tavoite vastaa faktaa (3), joten algoritmin suoritus päättyy. Lopputulokseksi saadaan tosi, eli on päätelty, että kana on lintu. Ohjelma vastaa siis: true.

Esimerkki. Toinen logiikkaohjelma.

(1) lintu (X) ← munii (X), siivekäs (X)
(2) munii (python)
(3) siivekäs (kana)
(4) munii (kana)

Olkoon tavoite

? lintu (python)

eli "Onko python lintu?". Korvaamalla muuttuja X vakiolla python ensimmäisen klausuulin pää ja tavoite samastuvat, joten uudeksi tavoitteeksi saadaan (X = python):

? munii (python), siivekäs (python)

Näin alkuperäinen tavoite "Onko python lintu?" muuttuu tavoitteeksi "Muniiko python ja onko pythonilla siivet?" Koska ensimmäinen osatavoite samastuu faktaan (2), seuraavassa vaiheessa tavoite redusoituu muotoon:

? siivekäs (python)

Mikään klausuuleista ei kuitenkaan vastaa tätä tavoitetta (se ei samastu klausuuliin 3, koska klausuulissa esiintyvä 'kana' on vakio, ei muuttuja). Ohjelman suoritus päättyy tavoitteeseen pääsemättä ja vastaa: false. Tämä voidaan tulkita kahdella tavalla: joko ohjelman suorituksen perusteella ei kyetty osoittamaan, onko python lintu vai ei, tai sitten osoitettiin, että python ei ole lintu. Jälkimmäinen tulkinta on tarkoituksenmukainen siinä tapauksessa, että klausuulien ajatellaan sisältävän kaiken oleellisen tiedon (ns. *suljetun maailman olettamus*).

Tavoitteessa voi esiintyä myös muuttujia. Esimerkiksi tavoite

? lintu (Y)

tarkoittaa "Onko olemassa eläintä Y, joka on lintu?". Päästäkseen tavoitteeseen prosessori yrittää löytää jonkin eläimen Y, joka on lintu, käyttäen samanlaista samastus- ja korvausstrategiaa kuin edellä. Paitsi vakiolla, muuttuja voidaan samastuksen aikaansaamiseksi korvata myös toisella muuttujalla. Suoritus etenee seuraavasti:

- Klausuuli (1) ja tavoite samastuvat, kun korvataan muuttuja X muuttuja Y:llä, jolloin uudeksi tavoitteeksi saadaan: *munii* (Y), *siivekäs* (Y).
- Nyt faktat (2) ja (4) samaistuvat osatavoitteen *munii* (Y) kanssa, antaen Y=python tai Y=kana. Fakta (3) ja tavoite *siivekäs*(Y) samaistuvat, antaen Y=kana. Koska tavoitteessa *munii* (Y), *siivekäs* (Y) vaaditaan, että kumpikin osatavoite toteutuu, lopputuloksena saadaan: Y=kana.

Tässä esimerkissä tavoite osoitettiin todeksi etsimällä yksi sen toteuttava objekti. Usein on tarpeellista etsiä kaikki ilmentymät, jotka täyttävät tavoitteen:

Esimerkki. Logiikkaohjelma.

(1) isovanhempi (X, Y) ← vanhempi (X, Z), vanhempi (Z, Y)
(2) vanhempi (elizabeth, charles)
(3) vanhempi (charles, william)
(4) vanhempi (charles, henry)

Ensimmäisen klausuulin tulkinta on: "henkilö X on henkilön Y isovanhempi, jos X on jonkin henkilön Z vanhempi ja Z on henkilön Y vanhempi" tai "jotta voidaan osoittaa, että X on Y:n isovanhempi, on osoitettava, että X on Z:n vanhempi ja että Z on Y:n vanhempi". Jos halutaan tietää, onko Elizabeth Henryn isovanhempi, ohjelman suoritus etenee seuraavasti:

? isovanhempi (elizabeth, henry)

(1) X = elizabeth, Y = henry

? vanhempi (elizabeth, Z), vanhempi (Z, henry)

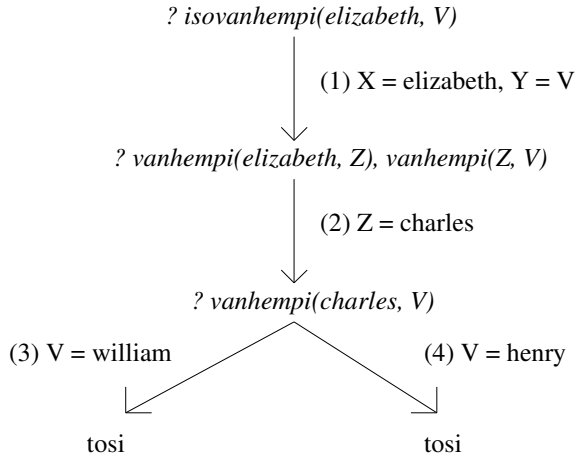
(2) Z = charles

? vanhempi (charles, henry)

(4)

true

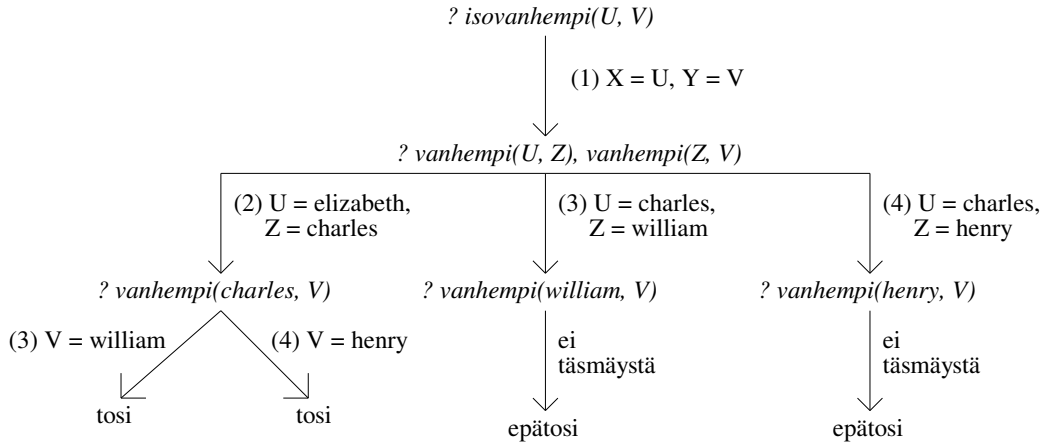
Tavoite on totta, eli Elizabeth on Henryn isovanhempi. Jos tavoite on "etsi kaikki Elizabethin lastenlapset" tai "onko Elizabeth (kenenkään) isovanhempi?", ohjelman suoritus etenee seuraavasti:



Ratkaisuja löytyy siis kaksi, eli tulokseksi saadaan, että Elizabethin lastenlapsia ovat William ja Henry.

Jos tehtävänä on etsiä kaikki tavoitteen täyttävät ilmentymät, prosessorin tulee käydä läpi kaikki ratkaisupolut. Kukin polku tutkitaan vuorollaan loppuun (lopputulos: tosi tai epätosi), minkä jälkeen palataan takaisin edelliseen sellaiseen haarautumiskohtaan, josta voidaan valita uusi polku. Tällaista menettelyä nimitetään *peruuttamiseksi* (backtracking). Sitä tarvitaan myös silloin, jos päädytään umpikujaan.

Tarkastellaan tavoitetta "etsi kaikki isovanhempi-lapsenlapsi -parit". Tällöin ohjelma etenee seuraavasti:



Samaa logiikkaohjelmaa voidaan siis käyttää joustavasti usealla eri tavalla tavoitteen muodosta riippuen. Esimerkiksi

? isovanhempi (elizabeth, henry)	"Onko Elizabeth Henryn isovanhempi?"
? isovanhempi (elizabeth, V)	"Etsi kaikki Elizabethin lastenlapset V."
? isovanhempi (U, william)	"Etsi kaikki Williamin isovanhemmat U."
? isovanhempi (U, V)	"Etsi kaikki isovanhempi-lastenlapsiparit (U, V)."

Esimerkki. Logiikkaohjelma, joka lajittelee listan nousevaan järjestykseen.

Klausuuli, joka määrittelee listan lajittelun siten, että lisätään listan pää sopivaan kohtaan lajiteltua häntää:

sort(H | T, S) ← sort(T, L), insert(H, L, S)

Klausuuli tulkitaan seuraavasti: "S on lista H | T lajiteltuna, jos L on T lajiteltuna, ja S on saatu siten, että lisätään H listaan L oikeaan paikkaan." Tämä klausuuli pätee kaikille muille paitsi tyhjälle listalle (jolla ei ole päätä eikä häntää). Tyhjät listat ovat aina järjestyksessä, joten määritellään fakta:

sort([ ], [ ])

Tämä tarkoittaa sitä, että tyhjä lista on lajiteltunakin tyhjä lista. Vielä pitää määritellä alkion lisäys lajiteltuun listaan. Tässä on huomattava kolme tilannetta:

1. Listaan lisätään alkio, joka edeltää aakkosissa listan ensimmäistä alkiota. Tämä määritellään klausuulilla

   insert(X, H | T, X | H | T) ← precedes(X, H)

Klausuuli tulkitaan seuraavasti: "Kun X sijoitetaan listaan H | T, tulokseksi saadaan lista X | H | T edellyttäen, että X edeltää aakkosissa H:ta."

2. Listaan lisätään alkio, joka seuraa aakkosissa listan ensimmäistä alkiota. Tämä määritellään klausuulilla

   insert(X, H | T1, H | T2) precedes(H, X), insert(X, T1, T2)

Klausuuli tulkitaan seuraavasti: "Kun X sijoitetaan listaan H | T1, tulokseksi saadaan lista H | T2 edellyttäen, että H edeltää aakkosissa X:ää ja että T2 saadaan tulokseksi, kun X lisätään listaan T1."

3. Lisätään alkio tyhjään listaan. Tämä määritellään faktalla

   insert(X, [ ], [X])

Lajitteluohjelma on kokonaisuudessaan seuraavanlainen:

```
sort(H | T, S) ← sort(T, L), insert(H, L, S)
sort([ ], [ ])
insert(X, H | T, X | H | T) ← precedes(X, H)
insert(X, H | T1, H | T2) ← precedes(H, X), insert(X, T1, T2)
insert(X, [ ], [X])
```

Ohjelmaa voidaan käyttää monella tavalla:

? sort ([koira kissa sika], [kissa koira sika])	"onko lista [kissa koira sika] listan [koira kissa sika] lajiteltu versio?"
? sort ([koira kissa sika], S)	"lajittele lista [koira kissa sika] listaksi S"
? sort (M, [kissa koira sika])	"etsi kaikki listan [kissa koira sika] permutaatiot M" eli "onko olemassa listaa M, joka olisi lajiteltuna [kissa koira sika]?"

Funktionaalisen ohjelman tavoin logiikkaohjelmassakaan klausuulien kirjoitusjärjestys ei vaikuta ohjelman antamaan lopputulokseen. Prosessori käyttää jotain ennalta määrättyä strategiaa etsiessään klausuuleita päästäkseen tavoitteeseen. Suoritusvaihe sisältää useita pohjimmiltaan epädeterministisiä valintoja:

- Prosessorin on valittava, mille osatavoitteelle haetaan seuraavaksi vastinetta. Jokin vaihtoehto voi aiheuttaa pitemmän toimenpiteiden sarjan kuin toinen, ja jokin vaihtoehto voi aiheuttaa sen, että ohjelma ei pääty koskaan. Voidaan kuitenkin osoittaa, että jos kaikki vaihtoehdot johtavat ohjelman päättymiseen, ne johtavat samaan lopputulokseen. Useimmissa tapauksissa on siis samantekevää, mikä vaihtoehto seuraavaksi valitaan.
- Jos valittu tavoitteen osa vastaa useita klausuuleita, prosessorin on valittava klausuuli, jonka perusteella se muodostaa uuden tavoitteen. Tälläkään valinnalla ei ole vaikutusta lopputulokseen: peruutus mahdollistaa kaikkien mahdollisten ratkaisujen löytymisen, ainoastaan ratkaisujen löytymisjärjestys voi vaihdella.
- Jos on käytettävissä useampia prosessoreita, eri vaihtoehtoja voidaan käydä läpi samanaikaisesti rinnakkain, mikä luonnollisesti vähentää peruutustarvetta ja nopeuttaa merkittävästi ohjelman suoritusta.

Logiikkaohjelman klausuulit voidaan kirjoittaa ohjelmaan missä järjestyksessä tahansa, joten ohjelmoijan ei tarvitse kiinnittää huomiota ohjelman suoritusjärjestykseen, vaan sen määräävät prosessorin täsmäys- ja peruutusstrategiat. Useimmissa logiikkaohjelmointikielissä ohjelmoija kuitenkin halutessaan voi (ja joissakin tapauksissa hänen täytyykin) vaikuttaa ohjelman suoritusjärjestykseen. Esimerkiksi Prolog-kielessä suoritus etenee niin, että tavoitetta käydään läpi vasemmalta oikealle, ja klausuuleita tutkitaan niiden kirjoitusjärjestyksessä.

Logiikkaohjelmiakaan ei voida sujuvasti toteuttaa nykyisissä von Neumann -tietokoneissa, koska niissä ei ole toteutettu logiikkaohjelmoinnin keskeisiä operaatioita: samastusta eli täsmäystä (tavoitteen sovittamista klausuuliin), korvausta ja peruutusta tyydyttävällä tavalla. Toteutus vaatii monimutkaisen ohjelmiston ja tietorakenteet, jotka lisäävät merkittävästi ohjelmien suoritusaikaa. Sen vuoksi onkin kehitetty omia juuri logiikkaohjelmointiin tarkoitettuja tietokoneita. Logiikkaohjelmointia käytetään etenkin Euroopassa ja Japanissa lähinnä tietämyskanta- ja asiantuntijajärjestelmissä, siis paljolti samantapaisissa tehtävissä kuin funktionaalista ohjelmointia USA:ssa.