

## *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. Sivulla 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 yleisiä kommentteja voi lähettää alla olevaan sähköpostiosoitteeseen. Kiitän Lasse Bergrothia viimeisistä kommentteista.

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.

## Sisällysluettelo

<b>1</b>	<b>Johdanto</b>	<b>5</b>
1.1	Terminologiaa	5
1.2	Tietokone: rakenne ja toimintaperiaate	6
1.3	Tietokoneen ominaisuuksia	9
1.4	Ohjelmointikielet, ohjelmat ja ohjelmistot	10
1.5	Algoritminen ongelmanratkaisu	12
<b>2</b>	<b>Algoritmien suunnittelu</b>	<b>15</b>
2.1	Algoritmin perusvaatimukset	15
2.2	Ohjelmointikielet	17
2.2.1	Syntaksi ja semantiikka	18
2.3	Algoritmien asteittainen tarkentaminen	19
2.4	Imperatiivinen paradigma	21
2.4.1	Muuttuja, tyyppi, lause, lauseke ja asetuslause	22
2.4.2	Lauseiden suoritusjärjestys	23
2.5	Ohjausrakenteet	24
2.5.1	Peräkkäisyys	24
2.5.2	Valintalauseet ja totuusarvoiset lausekkeet	24
2.5.3	Toistolauseet	29
2.5.4	Lajitteluesimerkki	34
2.6	Modulaarisuus	36
2.6.1	Abstraktiot	36
2.6.2	Moduulit	37
2.6.3	Parametrit	41
2.6.4	Proseduurit ja funktiot sekä ohjelman rakenne	42
2.6.5	Yhteenvedo	50
2.7	Rekursio ja iteraatio	51
2.7.1	Rekursio	52
2.7.2	Iteraatio	54
2.7.3	Rekursio vai iteraatio?	57
2.8	Tieto- ja tallennusrakenteet	58
2.8.1	Tyyppi, abstrakti tietotyyppi ja sen implementointi	58
2.8.2	Tallennusrakenteet	59
2.8.2.1	Tietue	59
2.8.2.2	Taulukko	60
2.8.2.3	Linkitetty rakenne	62
2.8.3	Abstraktit tietotyypit	63
2.8.3.1	Lista	63
2.8.3.2	Listan toteutus	64
2.8.3.3	Puu	65
2.8.3.4	Binääripuut	66
2.8.3.5	Binääripuun toteutus	69
2.8.3.6	Graafi	71
2.8.3.7	Graafin toteutus	72
2.8.4	Abstraktien tietotyyppien implementoinnista	73
2.8.5	Oliokeskeinen ohjelmointi	75

<b>3</b>	<b>Algoritmitheoriaa</b>	<b>79</b>
<b>3.1</b>	<b>Tehtävän algoritminen ratkeavuus</b>	<b>79</b>
3.1.1	Laskettavuus	79
3.1.2	Church-Turingin teesit	80
3.1.3	Pysähtymisongelma ja muita algoritmisesti ratkeamattomia ongelmia	81
3.1.4	Yhteenvedo laskettavuudesta	83
<b>3.2</b>	<b>Kompleksisuus</b>	<b>83</b>
3.2.1	Kompleksisuuden kertaluokat	86
3.2.2	Hajota ja hallitse	88
3.2.3	Hanoin tornien ongelma	92
3.2.4	Kelvottomia ongelmia	95
3.2.5	$P$ ja $NP$	96
<b>3.3</b>	<b>Oikeellisuus</b>	<b>97</b>
3.3.1	Testaus	98
3.3.2	Oikeaksitodistaminen	98
3.3.2.1	Induktioperiaate	100
3.3.2.2	Invarianttien käyttö oikeaksitodistamisessa	102
3.3.2.3	Oikeaksitodistaminen peräkkäisten väitteiden avulla	103
3.3.2.4	Terminoituvuus	103
<b>4</b>	<b>Tietokoneen rakenne ja toiminta</b>	<b>105</b>
<b>4.1</b>	<b>Matemaattiset perusteet</b>	<b>105</b>
4.1.1	Lukujärjestelmät	106
4.1.2	Kokonaislukujen esittäminen	108
4.1.3	Liukulukujen esittäminen	111
4.1.4	Boolen algebra	111
<b>4.2</b>	<b>Fysikaaliset perusteet</b>	<b>113</b>
4.2.1	Puolijohteet	113
4.2.2	Puolijohdekomponentit	114
4.2.3	Loogiset piirit	116
<b>4.3</b>	<b>Tietokoneen komponentteja</b>	<b>120</b>
4.3.1	Yhteenlasku	121
4.3.2	Vähennyslasku	123
4.3.3	Kiikku	124
4.3.4	Rekisteri	127
4.3.5	Väylät	128
4.3.6	Ohjauslogiikka ja kello	129
<b>4.4</b>	<b>Mikro-ohjelmitava tietokone</b>	<b>130</b>
4.4.1	Mikro-ohjelmitavan tietokoneen rakenne	131
4.4.2	Yhteenvedo	133
4.4.3	Mikro-ohjelmointi	134
4.4.4	Kertolasku	138
4.4.5	Jakolasku	141
<b>4.5</b>	<b>Konekieli</b>	<b>141</b>
4.5.1	Mikro-ohjelmitu tulkki konekielelle	146
4.5.2	Monimutkaisemmat konekielet	149
4.5.3	Osoitustavoista	149

<b>4.6</b>	<b>Kommunikointi</b>	<b>151</b>
4.6.1	Syöttö- ja tulostuslaitteet	152
4.6.2	Keskeytyspohjainen siirräntä	152
4.6.3	Tietokoneverkot	153
<b>5</b>	<b>Systemiohjelmisto</b>	<b>156</b>
<b>5.1</b>	<b>Ohjelmointikielten kääntäminen</b>	<b>157</b>
5.1.1	Tulkitseminen	158
5.1.2	Kääntäminen	159
<b>5.2</b>	<b>Syntaksin määrittely</b>	<b>159</b>
5.2.1	Kieliopit	159
<b>5.3</b>	<b>Kääntäjän toiminta</b>	<b>162</b>
5.3.1	Selaaminen	163
5.3.2	Jäsentäminen	164
5.3.3	Koodin generointi	166
5.3.4	Symbolinen konekieli	171
<b>5.4</b>	<b>Käyttöjärjestelmät</b>	<b>172</b>
5.4.1	Vuoronvaihto	173
5.4.2	Ajoitus ja resurssien varaaminen	175
5.4.3	Muistinhallinta	175
5.4.4	Siirräntä	178
<b>5.5</b>	<b>Tiedostojärjestelmä</b>	<b>179</b>
5.5.1	Peräkkäisorganisaatio	180
5.5.2	Suora organisaatio	180
5.5.3	Indeksoitu organisaatio	182
<b>6</b>	<b>Epädeterminismi, rinnakkaisuus ja vapaajärjesteisyys</b>	<b>185</b>
<b>6.1</b>	<b>Epädeterministiset ongelmat</b>	<b>185</b>
6.1.1	Hakuprobleemat	185
6.1.2	Syvyys- ja leveyshaku	186
6.1.3	Pelien pelaaminen	190
<b>6.2</b>	<b>Rinnakkaisalgoritmit</b>	<b>193</b>
6.2.1	Rinnakkaistaminen	193
6.2.2	Rinnakkaisalgoritmien kompleksisuus	195
<b>6.3</b>	<b>Vapaajärjesteinen ohjelmointi</b>	<b>196</b>
6.3.1	Yleistetyt listat	197
6.3.2	Funktionaalinen ohjelmointi	198
6.3.3	Logiikkaohjelmointi	201

**Liitteet:** Mikro-ohjelmitava tietokone  
(löytyy samalta sivulta, josta tämä moniste on ladattavissa)

# 1 Johdanto

## 1.1 Terminologiaa

Tietojenkäsittelytieteen keskeisin käsite on algoritmi. *Algoritmilla* tarkoitetaan yksinkertaisesti jonkin tehtävän suorittamiseksi tarvittavien toimenpiteiden kuvausta. Yleensä tämä toimenpiteiden joukko on järjestetty niin, että toimenpiteet suoritetaan yksi kerrallaan peräkkäin tietyn lopputuloksen saavuttamiseksi. (Algoritmin tarkempaan määrittelyyn palataan luvussa 3.) Algoritmiin liittyy lisäksi toimenpiteet suorittava subjekti, "kone", joka voi olla esimerkiksi ihminen, robotti, tietokone tai jokin kuvitteellinen kone. Jotta tämä subjekti voisi toimia tehtävän ratkaisemiseksi algoritmissa kuvatulla tavalla, eli *suorittaa* algoritmin, täytyy algoritmi esittää tavalla, jota sen suorittaja ymmärtää. Ihmisen suoritettaviksi tarkoitetut algoritmit voidaan esittää luonnollisella kielellä, mutta yleensä on käytettävä eksaktimpaa ja yksinkertaisempaa keinoalasta kieltä, ns. formaalista kieltä. *Tietokoneet* saadaan suorittamaan algoritmeja, kun nämä esitetään sopivalla formaalisella kielellä, ns. *ohjelmointikielellä*. Jollakin ohjelmointikielellä kirjoitettua algoritmia sanotaan *tietokoneohjelmaksi* tai lyhyesti *ohjelmaksi*.

Tietojenkäsittelyalan termistö on uutta ja osin sekavaa. Puhutaan mm. tietoteollisesta alasta, joka on elinkeinoelämän keskeisimpiä osa-alueita modernissa tietoyhteiskunnassa.

Vaikka myös ihmisen toiminnan osuutta *tietojärjestelmissä* voidaan ainakin osittain tutkia ja mallintaa tieteellisesti mm. systeemiteorian keinoin, keskitytään perinteisessä *tietojenkäsittelytieteessä* (computer science) tietojärjestelmän eksakteihin komponentteihin eli tietokonejärjestelmään. Siinä missä perinteisessä tietojenkäsittelytieteessä algoritmeja ja niiden abstrakteja matemaattisia ominaisuuksia tutkitaan matemaattis-luonnontieteellisestä näkökulmasta, teknistieteellistä lähestymistapaa soveltavassa tietojenkäsittelytekniikassa (computer engineering) algoritmeja tutkitaan tietokoneohjelmoina, siis tietokonejärjestelmän osana. Näin ollen myös itse tietokone ja sen toimintaperiaate ovat tietojenkäsittelytieteen keskeisiä tutkimuskohteita.

Tutkimuskohdetta laajentamalla sekä hyväksymällä inhimillisestä komponentista johtuva epävarmuus ja tietynlainen epämääräisyys osaksi tarkasteltavaa kohdetta päädytään tietojärjestelmiä kokonaisuutena tutkivaan *tietojärjestelmätieteeseen* (information systems science). Sen tarkoituksena on mm. kehittää ja mallintaa tietojärjestelmiä käyttäen eksakteja mallintamismenetelmiä. Yhdessä nämä kaksi tieteenalaa muodostavat kokonaisuuden, jota Suomessa kutsutaan *tietojenkäsittelytieteiksi*.

*Tietotekniikka* puolestaan on yleisnimitys, joka on pitkälti korvannut aiemmin yleisesti käytetyn termin ATK (automaattinen tietojenkäsittely). Toisinaan tietotekniikka-termin katsotaan kattavan varsinaisten tietojenkäsittelytieteiden lisäksi tietoliikennetekniikan ja osittain myös digitaalielektroniikan.

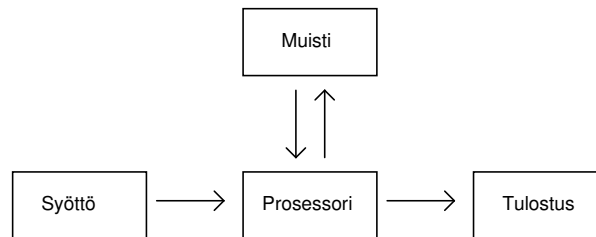
Viime aikoina tietokoneiden käyttö on laajentunut hyvin voimakkaasti perinteisten sovellusalojen ulkopuolelle. Tietokoneet ovat tulleet osaksi jokapäiväistä kulutuselektroniikkaa, ja raja varsinaisten tietokoneiden ja automaattisia toimintoja sisältävien

laitteiden ja koneiden välillä on hämärtynt. Tietokone on tavallaan piilotettu osaksi erilaisia elektroniikkaa sisältäviä laitteita, kuten puhelimia, viihde-elektroniikkalaitteita, ajoneuvoja tai vaikkapa hissejä. Tällöin puhutaan *sulautetuista järjestelmistä* (embedded systems). Järjestelmä sisältää toimilaitteet ja niiden ohjelmistot. Käsite on noussut esiin erityisesti matkaviestinten rajun kehityksen yhteydessä, mutta myös autoissa ja muissa liikennevälineissä laitteiden toimintaa ohjaavien tietokoneiden merkitys kasvaa hyvin nopeasti. Sulautetut järjestelmät muodostavatkin nykyään kaikkein nopeimmin kasvavan prosessori- ja ohjelmistotuotannon sovellusalueen. Sulautetut järjestelmät sisältävät periaatteessa normaalin tietokonejärjestelmän komponentit. Esimerkiksi nykyaikaisen matkapuhelimen sisältämä prosessori vastaa muutaman vuoden takaisen mikrotietokoneen prosessoria. Toisin kuin varsinaisissa tietokoneissa, sulautettujen järjestelmien prosessorit ovat yleensä toiminnoiltaan käyttötarkoitukseensa sovitettuja ns. mikrokontrollereita tai erikseen räätälöityjä integroitua piirejä (Application Specific Integrated Circuit – ASIC). Muistin määrä ja laatu vaihtelevat melkoisesti. Usein järjestelmän toimilaitteille asetetaan erityisiä fyysisiä vaatimuksia, kuten pieni koko, mekaaninen kestävyys sekä korkeiden tai matalien lämpötilojen sieto. Myös niiden syöttö- ja tulostuslaitteet poikkeavat tavallisista. Tyypillisiä sulautettujen järjestelmien syöttölaitteita ovat erilaiset ympäristöä havainnoivat anturit, kuten lämpö- tai kiihtyvyyssanturit. Järjestelmän ”tulostus” tapahtuu usein erilaisina säätötoimintoina, kuten venttiilin säätämisenä tai mekaanisen liikkeen tuottamisena. Toki sulautettu järjestelmä voi sisältää myös näytön ja näppäimistön – esimerkkinä vaikkapa matkapuhelin.

## 1.2 Tietokone: rakenne ja toimintaperiaate

*Tietokone* (engl. computer) on kone, joka voi ratkaista hyvinmääriteltyjä tehtäviä rutiininomaisesti suorittamalla suuria määriä yksinkertaisia perusoperaatioita suurella nopeudella. Se on kone, joka voi suorittaa ainoastaan sellaisia tehtäviä, jotka voidaan määrittellä niillä yksinkertaisilla operaatioilla, joita se osaa suorittaa. Jotta tietokone saadaan suorittamaan tehtävä tai ratkaisemaan ongelma, sille on kerrottava, mitä operaatioita sen tulee suorittaa, eli sille tulee kuvata, miten tehtävä suoritetaan tai ongelma ratkaistaan. Tällaista tehtävänkuvausta sanotaan *algoritmiksi* (engl. algorithm). Algoritmi on siis menetelmä, jonka mukaisesti tehtävä suoritetaan. Algoritmi muodostuu toimenpiteistä, joiden onnistunut suoritus johtaa annetun tehtävän tekemiseen tai asetetun ongelman ratkaisemiseen. Algoritmit eivät ole luonteenomaisia pelkästään automaattiselle tietojenkäsittelylle, vaan myös jokapäiväisiä ihmisen suorittamia toimintoja (ruoanlaitto, soittaminen, rakentaminen) voidaan kuvata algoritmeilla (vrt. reseptit, nuotit, rakennusohjeet). Tehtävän suoritusta sanotaan *laskuksi* (computation) tai *prosessiksi* (process) ja tehtävän suorittajaa, ”koneetta”, joka voi olla esimerkiksi ihminen tai tietokone, nimitetään *prosessoriksi* (processor). Prosessori siis synnyttää prosessin suorittamalla prosessia kuvaavan algoritmin. Prosessi vaatii aina *resursseja* (resource), jollaisia ovat erityisesti laskentaan kuluva aika sekä erilaiset laitteistoresurssit, kuten muistitila. Resurssien tarve vaihtelee algoritmeittain ja prosessoreittain. Termi tietokone on jossain määrin harhaanjohtava, kyseessä on vain nopea ohjelmitava *laskija*. Koneen ohjelmitavuus tekee koneesta ikään kuin älykkään koneen – tietokoneen – kaltaisen.

Tietokoneen pääkomponentit ovat **prosessori** eli **suoritin** (central processing unit, CPU) ja **muisti** (memory) sekä erilaiset **siirräntä-** eli **syöttö- ja tulostuslaitteet** (input and output devices, I/O devices). Prosessori suorittaa osaamiaan perusoperaatioita. Muisti voidaan jakaa keskus- ja oheismuistiin. **Keskusmuistissa** säilytetään algoritmia, joka kuvaa suoritettavat operaatiot ja niiden järjestyksen, sekä tietoa (data), johon operaatiot kohdistetaan. **Oheismuistia** (esim. kiintolevy) tarvitaan suurten tietomäärien pitkäaikaiseen varastointiin. **Syöttölaitteilla** algoritmi ja käsiteltävä tieto syötetään muistiin, ja tulostuslaitteilla tietokone tulostaa käsittelyn tulokset. Syöttö- ja tulostuslaitteet samoin kuin oheismuisti eivät ole välttämättömiä tietokoneen toiminnalle (algoritmien suorittamiselle), mutta ilman niitä tietokone olisi kommunikointikyvytön ja näin muodoin hyödytön musta laatikko.

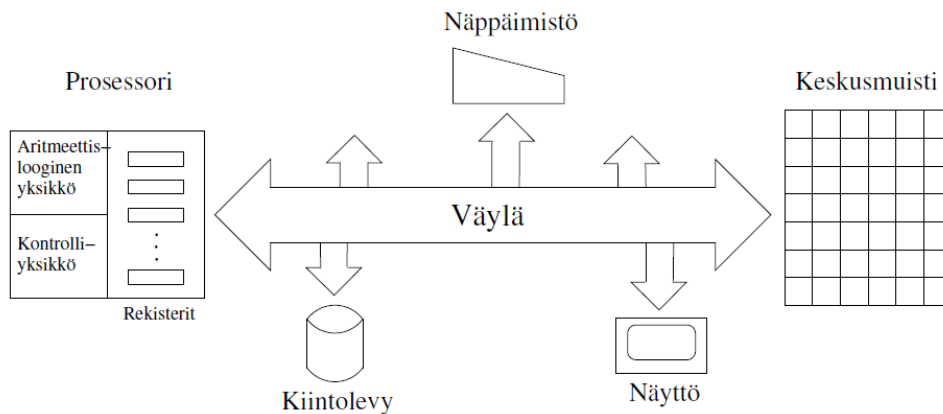


Tietokoneen toiminnan kannalta keskeisimmät komponentit ovat prosessori ja keskusmuisti, joka on jaettu pysyväismuistiin (ROM, Read Only Memory) ja käyttömuistiin eli työmuistiin (RAM, Random Access Memory). ROM-muisti sisältää yleensä ohjelman, jolla tietokoneen laitteisto alustetaan ja käyttöjärjestelmä ladataan tietokonetta käynnistettäessä.

Tietokonelaitteisto koostuu useista eri komponenteista, joiden välistä tiedonsiirtoa hoitaa johtimien kokoelma, jota kutsutaan **väyläksi** (engl. bus). Prosessorin, muistien ja laitteiden välinen data (tieto) kulkee aina väylän kautta. Tietokoneen toimintaperiaate yksinkertaistettuna on seuraava: prosessori noutaa suoritettavan ohjelman konekieliset käskyt ja tarvittavan datan keskusmuistista, suorittaa käskyissä määritetyt toiminnot ja siirtää käsitellyn datan takaisin keskusmuistiin (tai muihin muistilaitteisiin/tulostuslaitteelle). Tietokoneen toimintanopeus riippuu huomattavasti prosessorin nopeudesta. Prosessorin nopeuteen vaikuttaa puolestaan useampi tekijä, esimerkiksi kellotaajuus (ilmaistaan useimmiten gigahertseinä, GHz) ja prosessoriytimien sekä prosessorin sisällä olevan välimuistin määrä.

Prosessorin tärkeimmät osat ovat:

- **Aritmeettis-looginen yksikkö** (arithmetic and logic unit, ALU), jossa ovat dataa käsittelevät ja muokkaavat loogiset piirit. ALUssa suoritetaan siis esimerkiksi laskutoimitukset.
- **Rekisterit**, joihin prosessorissa käsiteltävä data muistista haetaan. Rekisterit ovat erityisen nopeita muistipaikkoja, joiden käsittely on huomattavasti nopeampaa kuin keskusmuistin.
- **Kontrolliyksikkö** (control unit, CU) eli ohjausyksikkö, jonka rekisterit sisältävät koneen tarvitsemaa kontrollitietoa. Kontrolliyksikkö nimensä mukaisesti pitää kirjaa siitä, missä vaiheessa ohjelman suoritus on.



Tietokone käsittelee siis dataa. Mitä data tarkoittaa ja miten tämä data tallennetaan tietokoneen muistiin? Käsitteet **data**, **tieto** ja **informaatio** määritellään usein epämääräisesti. Ne ovat sanoja, joita käytetään usein toistensa synonyymeinä, vaikka ne tarkoittavatkin eri asioita. Asiaa vaikeuttaa vielä se, että englannin kielessä ei ole tarkkaa vastinetta sanalle ”tieto”, vaan sana ”knowledge” on suppeampi. ATK-sanakirja määrittelee sanat seuraavasti:

- **data**: asian säännönmukainen esitys viestitettävässä tai käsitteilykelpoisessa muodossa (usein koodattu tietyllä tavalla), joka on koneellisesti luettavissa.
- **informaatio**: datan ihmiselle tuottama mielle tai merkitys.
- **tieto**: ihmisen ajattelun kohde tai tulos.

Oleellista on erottaa, että data on asian esitysmuoto tietokoneessa. Sen sijaan informaatio ja tieto mielletään usein samaksi asiaksi. Data voi olla esimerkiksi jono peräkkäisiä kirjaimia (esim. auto) tai bittijono, joilla ei ole mitään tiettyä merkitystä ennen datan tulkintaa. Tieto on taas ihmisen käsittelemä asia, datan tulkinta. Näin ollen tietokone suorittaa datan käsittelyä eikä tietojen käsittelyä.

Tietokoneen toiminta perustuu ns. **kahden olotilan periaatteeseen**: kaksi toisistaan erilaista vaihtoehtoa, esim. 'johdossa' on (arvo=1) tai ei ole jännitettä (arvo=0), on helppo toteuttaa ja havaita laitteistolla. Tällöin kaikki tieto on esitettävä tietokoneessa käyttäen arvoja 0 ja 1, joita kutsutaan **biteiksi**. Kokonaisluvut esitetään tietokoneessa käyttäen 2-järjestelmän esitystä ja merkit (esim. kirjaimet) koodataan bittijonoksi käyttäen tiettyä koodausta. Desimaaliluvut ja kokonaisluvut tallennetaan tietokoneen muistiin eri tavalla: esim. lukujen 1.0 ja 1 bittiesitykset ovat erilaiset. Teksti tallennetaan merkeittäin, jolloin jokaisella merkillä on oma koodinsa, joka esitetään bittijonona. Tietokoneessa voidaan käsitellä numeerisen ja tekstimuotoisen tiedon lisäksi myös kuvaa ja ääntä. Kuvan ja äänen käsittely (digitaalisessa) tietokoneessa vaatii tiedon muuttamisen ensin digitaaliseen muotoon eli bittijonoksi. Esimerkiksi CD-levyllä oleva musiikkikin on vain bittijono (siis dataa)! Kaikki data esitetään siis bittijonona, joten jokaiseen bittijonoon tulee liittää aina tieto siitä miten bittijono tulkitaan. *Tässä vaiheessa riittää tietää, että erityyppisillä tiedoilla (esim. 1.0 ja 1) on erilaiset bittiesitykset, ja näihin esitystapoihin palataan luvussa 4.*



Tietokoneessa on suuri määrä piirejä, joiden avulla bitit tallennetaan tietokoneen muistiin. Muisti voidaan ajatella järjestetyksi lokerikoksi, jossa jokaisella lokerolla on oma osoitteensa niin, että ensimmäisen lokeron osoite on nolla. Näitä lokeroita kutsutaan *muistipaikoiksi*. Tavanomainen muistipaikan koko on kahdeksan bittiä ja tätä kutsutaan *tavuksi*.

### 1.3 Tietokoneen ominaisuuksia

Tietokoneet ovat tulleet erottamattomaksi osaksi nykyaikaista yhteiskuntaa. Eräs tapa havainnollistaa tietokoneiden valtavaa yhteiskunnallista merkitystä on verrata tietokoneiden aiheuttamaa muutosta teolliseen vallankumoukseen. Teollisuusvallankumous laajensi ihmisten fyysisiä voimavaroja oleellisesti. Ihminen korvattiin koneella suurta fyysistä voimaa tarvitsevien tai toistuvien toimintojen suorituksessa – lihasvoimaa käyttämättä saadaan esimerkiksi suuri nosturi nostamaan satojen tonnien tavaraeriä. Tietokonevallankumous on puolestaan laajentanut ihmisten henkisiä voimavaroja. Lukuisten ihmisten ajattelua ja muistia kuormittavien toistuvien toimintojen suoritus on siirretty tietokoneelle. Aivoja juurikaan kuormittamatta saadaan tietokone tekemään esimerkiksi työlästä laskentaa tai monimutkaisia johtopäätöksiä sekä varastoimaan ja läpikäymään suuria tietomääriä.

Vertauksesta paljastuu myös eräs automatisoinnin mahdollinen haittapuoli: aiheuttavako tietokoneet uhan ihmisen älylliselle kapasiteetille säilyvyydelle samoin kuin muut koneet ovat aiheuttaneet uhan ihmisen fyysiselle kunnolle vapauttaessaan ihmisen lihas-työstä? Idea on tietysti siinä, että (tieto)koneet eivät korvaa vaan laajentavat ihmisen fyysisiä ja henkisiä voimavaroja, niin että aikaisemmin mahdottomat (tai taloudellisesti kannattamattomat) toimenpiteet tulevat mahdollisiksi. Toisaalta (tieto)koneet vapauttavat ihmiset ikävien rutiinitöiden sekä vaarallisten töiden suorittamisesta. Tietysti kaikki koneet voivat palvella mitä tahansa tarkoituspäästä sen laillista tai moraalista oikeutusta kysymättä. Perinteisesti suurimpia – ellei suurin – automaattiseen tietojenkäsittelyyn panostava inhimillisen toiminnan haara on ollut sotateollisuus. Muihin koneisiin nähden verrattoman monipuolisuutensa ansiosta tietokone onkin varsinainen hyvän ja pahan tiedon puu, jonka käyttömahdollisuuksia rajoittavat eniten käyttäjät, sekä tietoisesti että tiedostamattaan.

Laitteistosta riippumatta tietokoneilla on joitakin yleisiä ominaisuuksia, joiden vuoksi niiden käyttö on niin nopeasti yleistynyt. Verrataanpa näitä ominaisuuksia ihmisen kykyihin:

- **Nopeus**

Tyypillisen tietokoneen keskusyksikkö voi suorittaa miljardeja yksinkertaisia operaatioita sekunnissa. Monimutkaisimpiakin laskutoimituksia sisältävien algoritmien suorittaminen on siksi erittäin nopeaa. Ihmisen aivojen "laskentanopeus" on häviävän pieni verrattuna tietokoneen nopeuteen. Suorituksen nopeus lienee suurin yksittäinen tekijä, mikä saa tietokoneiden toiminnan tuntumaan monista ihmisistä niin "ihmeelliseltä". Mutta nopeimmatkaan tietokoneet eivät korvaa vaan ainoastaan täydentävät ihmistä. Monet ihmiselle helpot tehtävät ovat nimittäin koneelle äärimmäisen vaikeita ja päinvastoin. Ihmiselle helppoja ja koneelle vaikeita tehtäviä ovat esimerkiksi monet motoriset toiminnot, kuten käveleminen tai polkupyörällä ajo, ja useat kielen

käyttöön liittyvät toiminnot, kuten puhuminen, kuullun tai luetun ymmärtäminen tai vaikkapa runojen kirjoittaminen. Ihminen pystyy myös yhdistelemään asioita ja suorittamaan useita toimintoja samanaikaisesti, kun taas tietokoneet suorittavat yleensä yhden operaation kerrallaan. Tietokoneiden suuresta nopeudesta huolimatta on kuitenkin olemassa monia tehtäviä, jotka ovat niin vaikeita, että niitä ei voida suorittaa järkevissä ajassa nopeimmallakaan koneella.

- **Luotettavuus**

Tietokoneet eivät tee virheitä siinä mielessä kuin ihminen tekee. Ne eivät laskussaan milloinkaan poikkea annetusta algoritmista. Ne kykenevät väsymättä toistamaan suorituksensa kerrasta toiseen tarkalleen samanlaisena ja yhtä tehokkaasti. Kuten kaikki koneet, tietokoneetkin saattavat toki jumiutua tai vioittua. Yleensä tietokoneen tekemät virheet aiheutuvat suoritettavassa algoritmista esiintyvistä virheistä, virheellisestä syöttötiedosta tai sähköviasta. Tietokoneiden ongelmana on, etteivät ne (käskenettä) arvioi väli- tai lopputulosten järkevyyttä, jolloin virhe voi paljastua vasta niin myöhään, että vahinko on jo ehtinyt tapahtua.

- **Muisti**

Yksi tietokoneen merkittävistä ominaisuuksista on kyky varastoida suunnattomia määriä tietoa siten, että tiedot voidaan nopeasti hakea myös käsiteltäviksi. Kun tietokoneen tärkein tehtävä alun perin oli laskutoimitusten tekeminen (vrt. tietokoneen alkuperäinen englanninkielinen nimitys "computer"), on tietojen lyhyt- tai pitkäaikaisesta varastoinnista tullut yhä tärkeämpi sovelluskohde (vrt. tietokoneen nykyinen nimitys kielissä, joissa aluksi käytettiin vierasperäistä lainasanaa; esim. suomen "tietokone", ruotsin "dator", ranskan "ordinateur"). Tietokoneen muistin kapasiteetti ja hakunopeudet vaihtelevat tallennusvälineen mukaan. Tietokoneen muisti on järjestetty siten, että tietoalkio voidaan hakea muistista ainoastaan, jos tietoalkion paikka tallennusvälineellä tiedetään tarkasti. Ihmisen muistin ongelma ei ole niinkään kapasiteetti, vaan tiedon haku. Ihmisen muistissa avain tiedon hakuun ei nykyäskäytännön mukaan ole tiedon sijainti aivoissa, vaan erilaiset assosiatiiiviset suhteet, tiedon liittyminen muuhun tietoon. Toisaalta juuri tämä eroavaisuus mahdollistaa ihmiselle tyypillisen asioiden vapaan yhdistelemisen, luovuuden.

- **Kustannukset**

Kustannukset ovat kaikkein merkittävin tekijä, miksi tietokoneita käytetään niin paljon hyväksi eri aloilla. Tietokoneita käyttämällä saavutetaan usein merkittäviä säästöjä verrattuna samojen tehtävien suorittamiseen ihmistyövoimalla.

Jättäkäämme tässä kuitenkin enemmän filosofiset pohdiskelut, ja keskittykäämme tarkastelemaan niitä matematiikan, luonnontieteiden ja korkean teknologian perusteita, jotka ovat tehneet tietokoneet mahdollisiksi.

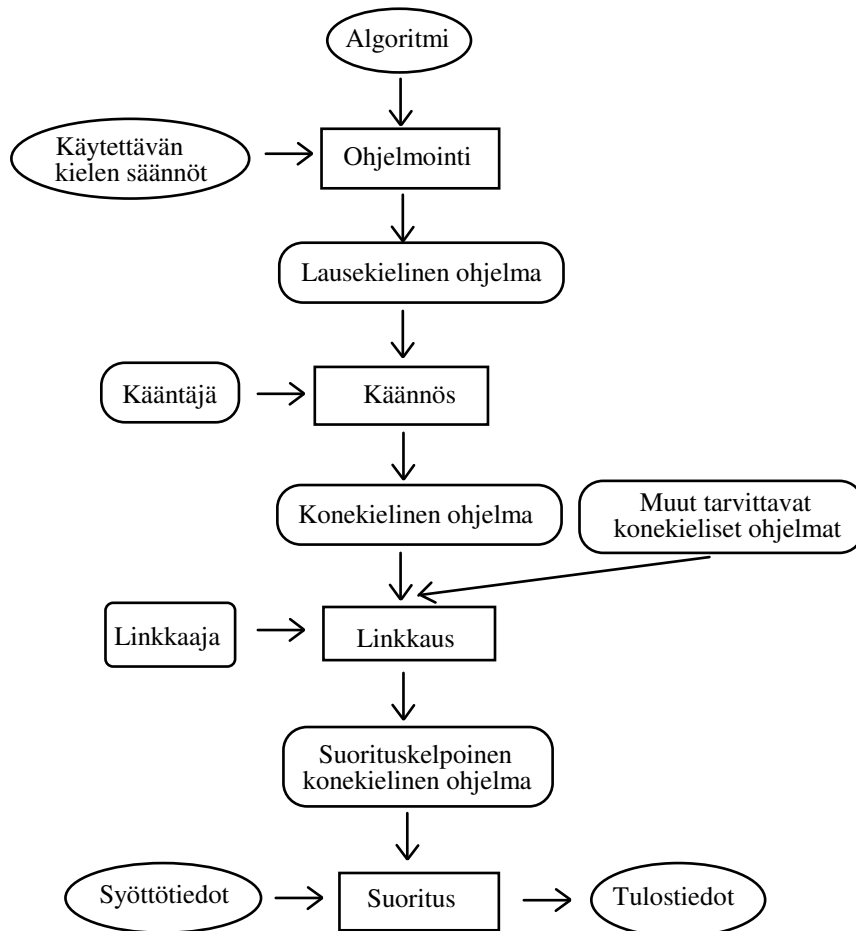
## **1.4 Ohjelmointikielet, ohjelmat ja ohjelmistot**

Ohjelma kirjoitetaan ohjelmointikielellä (programming language), ja toimenpidettä, jossa algoritmi muutetaan ohjelmaksi nimitetään ohjelmoinniksi (programming). Jokainen algoritmin askel esitetään ohjelmassa käskynä, lauseena. Ohjelma muodostuu

lausejonosta, jonka jokainen lause määrittää yhtä tai useampaa tietokoneen suoritettavaksi tarkoitettua perusoperaatiota. Tällaisia kieliä kutsutaan usein lausekieliksi. Tehtäviä voidaan ratkaista myös muullakin tavalla kuin käyttäen deterministisiä algoritmeja, joissa tiedetään aina tarkalleen mikä on seuraava suoritettava toimenpide, jolloin puhutaan ns. imperatiivisesta ohjelmoinnista. Epädeterminististä ohjelmointia tarkastellaan luvussa 6, ja sitä ennen tarkastelemme aina käskypohjaista determinististä ohjelmointia, jollaista myös konekieli on.

Yksinkertaisin ohjelmointikieli on kullekin prosessorille ominainen konekieli (machine language), jonka käskyjä tietokone osaa tulkita suoraan. Konekieli on kieli, joka on suunniteltu ja toteutettu yhdessä itse prosessorin kanssa. Konekielen käskyillä voidaan kuvata vain primitiivisiä toimenpiteitä, jotka kone osaa suorittaa. Konekieliset käskyt ovat hyvin yksinkertaisia (esimerkiksi "laske kaksi kokonaislukua yhteen"), joten niillä voidaan yleensä esittää ainoastaan hyvin pieniä algoritmin osia. Niinpä laajan algoritmin esittämiseen tarvitaan hyvin paljon konekielisiä käskyjä. Ohjelmointi konekielellä onkin erittäin työlästä, mutta onneksi vain hyvin harvoin tarpeellista. Konekielen toteuttamiseen ja konekieliseen ohjelmointiin tutustutaan monisteen neljännessä luvussa.

Ohjelmoinnin helpottamiseksi on kehitetty korkean tason kieliä (high level language) eli lausekieliä. Ne ovat koneista riippumattomia ohjelmointikieliä, joiden käsitteistö poikkeaa merkittävästi konekielistä. Nimensä mukaisesti korkean tason kielten käsitteet ovat abstraktimpia kuin konekielten käsitteet, jotka ovat luonnollisesti tiukasti sidoksissa koneen rakenteeseen ja toimintaan. Korkean tason kielet on pyritty suunnittelemaan siten, että tehtävien esittäminen algoritmeina olisi ihmiselle mahdollisimman helppoa ja luontevaa. Lausekielet eivät ole konekohtaisia, mutta sen sijaan jossain määrin tehtäväkohtaisia: jotkin kielet on suunniteltu nimenomaan tietyn tyyppisiä tehtäviä varten, mutta suuri osa lausekielistä on yleiskäyttöisiä. Jokaisella lausekieliselä lauseella voidaan siis esittää paljon suurempi osa algoritmista kuin konekäskyillä. Tietokone ei kuitenkaan ymmärrä korkean tason kieltä suoraan, vaan tällaisella kielellä kirjoitettu ohjelma tulee muuntaa (translate) konekielelle ennen suoritusta. Tämä voidaan suorittaa kääntämiseen tai tulkintaan tarkoitettulla tietokoneohjelmalla, *kääntäjällä* (compiler) tai *tulkilla* (interpreter), joiden toimintaan tutustutaan tarkemmin monisteen viidennessä luvussa. Kääntämisen ja tulkitsemisen välinen ero on siinä, että kääntäjä muuntaa koko ohjelman kerralla kohdekieliseen muotoon, kun taas tulkki tulkitsee ohjelmaa lause kerrallaan sen suorituksen aikana. Ero on vastaava kuin esimerkiksi EU-asiakirjan kääntämisellä kieleltä toiselle ja EU-parlamentaarikon puheenvuoron simultaanitulkkauksella. Usein tarkasteltava käännetty ohjelma ei ole yksinään suorituskelpoinen, vaan se tarvitsee toimiakseen muiden jo käännettyjen ohjelmien apua. Silloin suoritetaan vielä ns. *linkkaus*, jossa käännettyyn ohjelmaan liitetään tarvittavat käännetyt muut (apu)ohjelmat. Linkkaus suoritetaan siihen tarkoitukseen tehdyllä tietokoneohjelmalla. Lopputuloksena on suorituskelpoinen konekielinen ohjelma. Monissa tietokonejärjestelmissä ohjelman käänнос, linkkaus ja suoritus voidaan suorittaa yhdellä komennolla (esim. run). Tehtävän suorittaminen tietokoneella korkean tason kieltä käyttäen sisältää siis seuraavat vaiheet:



Korkean tason kielten käyttö helpottaa ohjelmointia ja vaikeuttaa vastaavasti kielen tulkintaa tai kääntämistä. Mutta koska työlään ja tarkkuutta vaativan ohjelmointityön suorittaa hidas ja virheitä tekevä ihminen ja toisaalta mekaanisen kääntämisen tai tulkinnan nopea ja luotettava tietokone, mahdollisimman suuri osa ohjelmointia olisi syytä pyrkiä suorittamaan korkean tason kielellä.

Tietokoneen ohjelmistot koostuvat *sovellusohjelmistosta* (application software) ja *systemiohjelmistosta* (system software). Sovellusohjelmistolla tarkoitetaan valmisohjelmia sekä käyttäjien omatekoisia ohjelmia. Systemiohjelmistolla tarkoitetaan esimerkiksi kääntäjiä ja tulkkeja, jotka huolehtivat korkean tason kielellä kirjoitetun ohjelman muuntamisesta konekielelle, sekä *käyttöjärjestelmiä*, jotka huolehtivat mm. syöttö- ja tulostuslaitteiden hallinnasta ja tietojen pitkäaikaissäilytyksestä. Käyttöjärjestelmä huolehtii siis laitteiden ja sovellusohjelmien lisäksi erityisesti tiedostojen hallinnasta. Systemiohjelmita käsitellään tarkemmin luvussa 5.

## 1.5 Algoritminen ongelmanratkaisu

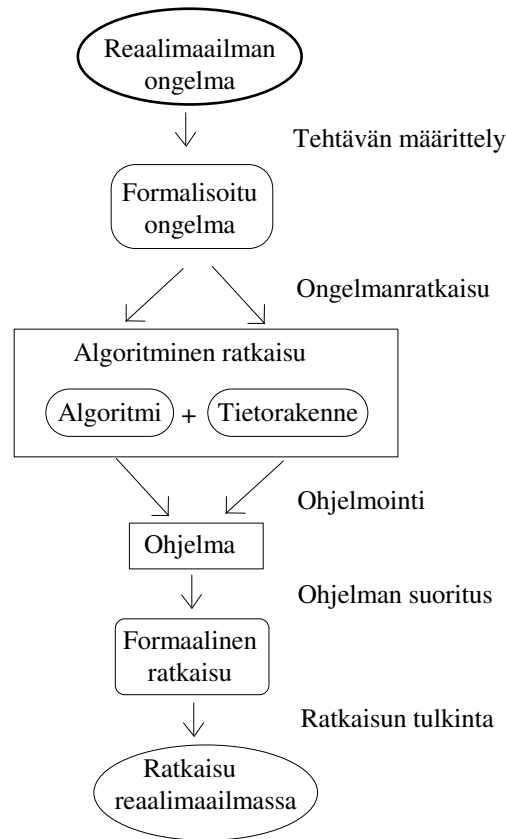
Jonkin ongelman ratkaisemiseksi laaditun algoritmin suorittaminen tietokoneella on systemaattinen toimenpide, joka voidaan yleisellä tasolla kuvata seuraavana algoritmina:

1. esitä algoritmi ohjelmana sopivalla ohjelmointikielellä
2. suorita ohjelma tietokoneessa

Vaikka algoritmin suorittaminen onkin itsessään systemaattinen, automatisoitavissa oleva toimenpide, niin algoritmien muodostaminen ei ole algoritmisen toimenpide, vaan luovuutta vaativa älyllinen ponnistus (vrt. kakkureseptin kehittäminen). **Ei siis ole mahdollista kirjoittaa algoritmia algoritmin muodostamiseksi.** Algoritmisen ongelmanratkaisu etenee seuraavasti (ks. kuva). Suoritettava tehtävä tai ratkaistava ongelma voi syntyä ongelman ratkaisijan suorasta havainnosta, tai tehtävä voi olla kuvattu esimerkiksi kirjallisesti. Joka tapauksessa tehtävä on yleensä lähtöisin reaali maailmasta. Niinpä tehtävä on luonnollista myös kuvata reaali maailman käsitteillä. Jotta ongelma voitaisiin ratkaista tietokoneella, tehtävänmäärittely pitää ensin formalisoida, ts. ongelma pitää esittää täsmällisesti formaaleilla käsitteillä ja mekanismeilla siten, että myös ongelman ratkaisu voidaan kuvata formaalisesti. Algoritmisen ratkaisun muodostaminen vaatii eksaktin ja aukottoman tehtävänmäärittelyn, koska algoritmin suorittaja ei enää voi kysellä tarkennuksia tai pohtia "mitähän tässä nyt oikein loppujen lopuksi halutaan?". Usein tehtävän määrittelyyn suhtaudutaan liian huolimattomasti. Koska tehtävän asettaja ei useinkaan tunne algoritmista ongelmanratkaisua kovin hyvin, joutuu tietojenkäsittelyalan ammattilainen usein aloittamaan saamansa toimeksiannon tehtävänmäärittelyn tarkentamisella.

Formalisoitu ongelma ratkaistaan, ja ratkaisu kuvataan algoritmina. Ratkaisun löytymisen voi joskus olla hyvinkin vaikeata, ja sen etsimiseksi on olemassa useita yleisiä ongelmanratkaisumenetelmiä. (Joitakin yleisimpiä menetelmiä, kuten reduktiota, tarkastellaan seuraavassa luvussa.) Kuitenkaan etukäteen ei aina voi tietää, millä keinoin kulloinkin tarkasteltava ongelma ratkeaa, vaan ratkaisu pitää vain "keksiä". Juuri tästä syystä algoritmien muodostaminen ei koneelta onnistu. Kaikkein yksinkertaisimpia ongelmia lukuun ottamatta tehtävän suorittamiseksi tarvittavan toiminnan kuvaus ei pelkästään riitä, vaan algoritmi vaatii toimiakseen myös käsittelemänsä tiedon kuvauksen, tiedon rakennetta mallintavan *tietorakenteen*. Tietorakenteen valinta liittyy tehtävän formalisointiin ja se vaikuttaa usein huomattavasti, jopa ratkaisevasti algoritmin hyvyyteen.

Kun tehtävä on ratkaistu ja algoritmi muodostettu, algoritmi ja siihen liittyvä tietorakenne esitetään valitulla ohjelmointikielillä. Algoritmin ja siihen liittyvän tietorakenteen sijasta voidaan yhtä hyvin puhua tietorakenteesta ja siihen liittyvistä operaatioista – ero on vain näkökulmassa. Perinteisesti on painotettu nimenomaan jonkin tehtävän suorittamiseksi tarvittavia toimenpiteitä ja puhuttu niiden avuksi tarvittavista tietorakenteista, mutta yhä yleisemmin tarkastelun painopiste on tietorakenteissa ja niille suoritettavissa operaatioissa – jopa niin, että puhutaan 'olioista' ja niiden suorittamista operaatioista. Näkökulma ja painotukset määräytyvät paljolti ongelmanratkaisussa käytettävien välineiden, suunnittelumenetelmien ja ohjelmointikielen mukaan. Joka tapauksessa kirjoitettu ohjelma testataan huolellisesti eri syöttötiedoilla, jotta sen oikeellisuudesta voitaisiin vakuuttautua, ja lopulta hyväksi havaittu ohjelma suoritetaan annettua tehtävän tapausta (tai tapauksia) vastaavilla syöttötiedoilla. Suorituksen tuloksena saadaan formaaliseen ongelmaan formaalinen ratkaisu, joka on vielä tulkittava reaali maailman ratkaisuksi. Tämä on kuitenkin yleensä sangen suoraviivaista, koska tässä vaiheessa ongelmalle valittu formaalinen esitys jo tunnetaan.



Algoritmin rooli on tietojenkäsittelytieteessä keskeinen: ilman algoritmia ei ole ohjelmaa, ja ilman ohjelmaa ei tietokoneella ole mitään suoritettavaa.

Algoritmit ovat kielestä ja koneesta riippumattomia, joten algoritmien suunnittelua voi opiskella pelkäämättä tiedon vanhentumista, vaikka tietokoneet ja ohjelmointikielet kehittyvätkin koko ajan. Algoritmien ohella myös tietokoneilla, ohjelmointikielillä ja systeemiohjelmistoilla on oma merkityksensä. Tietokoneet mahdollistavat algoritmien nopean, halvan ja luotettavan suorituksen, ja teknologian kehittyminen vain parantaa algoritmien suoritushallintaa nopeuttamalla käytössä olevien algoritmien suoritusta ja tekemällä yhä monimutkaisempien algoritmien käytön mahdolliseksi. Uudet ohjelmointikielet puolestaan helpottavat ohjelmointia ja mahdollistavat yhä abstraktimpien ja monimutkaisempien algoritmien esittämisen luotettavasti tietokoneen ymmärtämässä muodossa. Käyttöjärjestelmien kehitys mahdollistaa uusilla sovellusaloilla tarvittavien erittäin monimutkaisten ohjelmisto- ja laitteistokokonaisuuksien hallinnan.

## 2 Algoritmien suunnittelu

### 2.1 Algoritmin perusvaatimukset

Algoritmi on täsmällinen ja yksityiskohtainen kuvaus tai ohje siitä miten tehtävä suoritetaan. Algoritmin suunnittelu perustuu tehtävän määrittelyyn. Vain yksikäsitteisesti ja riittävän täsmällisesti määritellylle tehtävälle voidaan laatia algoritminen ratkaisu. Monissa tapauksissa algoritmi on vuorovaikutuksessa ympäristöönsä hyväksyen syötteitä ja tuottaen tulosteita. Syötteiden ja tulosteiden tarkka kuvaus on tärkeä osa tehtävän määrittelyä.

**Esimerkki.** Tehtävä: "Etsi lukujen suurin yhteinen tekijä".

Tehtävä on melko selvästi lausuttu, koska matemaattinen käsite "suurin yhteinen tekijä" on täsmällisesti määritelty. Epäselväksi jää, monestako luvusta ja millaisista luvuista on kysymys. Tarkoituksena on ehkä ratkaista tehtävä "etsi kahden positiivisen kokonaisluvun suurin yhteinen tekijä". Matematiikasta tiedetään, että kahdella positiivisella kokonaisluvulla  $m$  ja  $n$  on aina yksikäsitteinen suurin yhteinen tekijä, merkitään  $\text{sy}(n,m)$ , jolla tarkoitetaan suurinta sellaista positiivista kokonaislukua, joka on sekä  $m$ :n että  $n$ :n tekijä eli jakolaskut  $n/\text{sy}(n,m)$  ja  $m/\text{sy}(n,m)$  menevät tasan. Näin saadaan algoritmin syötteet ja tulosteet määriteltyä. Syötteenä ovat positiiviset kokonaisluvut  $m$  ja  $n$ , ja tuloste on  $m$ :n ja  $n$ :n suurin yhteinen tekijä.

Aivan mitä tahansa ohjetta ei siis voida pitää algoritmina. Algoritmilta vaaditaan seuraavat ominaisuudet:

- **Yleisyys:** algoritmin on sovelluttava määritellyn tehtävän kaikkiin tapauksiin.
- **Deterministisyys:** tehtävän ratkaisun on oltava yksikäsitteisesti määritelty, ja joka vaiheessa on tiedettävä täsmälleen, mitä seuraavaksi tehdään.
- **Tuloksellisuus:** algoritmin on annettava aina oikea tulos. Tämä vaatimus voidaan edelleen jakaa kahtia:
  - **oikeellisuus:** algoritmin antama tulos on aina oikea, ja
  - **terminoituvuus:** algoritmi todella antaa aina tuloksen, ts. sen suoritus päättyy aina.

Yleisyysvaatimus tarkoittaa, että algoritmi todella ratkaisee asetetun tehtävän kaikissa tehtävänmäärittelyä vastaavissa tapauksissa. Esimerkiksi kahden positiivisen kokonaisluvun suurimman yhteisen tekijän määrittävän algoritmin tulee löytää ratkaisu riippumatta siitä, minkä kahden luvun suurinta yhteistä tekijää haetaan. Vastaavasti esimerkiksi ohjetta "vastaus on kolme" ei voida pitää yhteenlaskualgoritmina, vaikka se antaakin oikean tuloksen tehtävään "laske ykkösen ja kakkosen summa". Yleisyysvaatimus on luonnollisella tavalla sidoksissa tehtävän määrittelyyn. Kahden luvun suurimman yhteisen tekijän määrävän algoritmin ei tarvitse osata määrätä kolmen luvun suurinta yhteistä tekijää. Yleisyys voidaan toki aina saavuttaa muuttamalla tehtävänmäärittelyä sopivasti. Mutta algoritmia, joka hyväksyy rajoittamattoman monta positiivista kokonaislukua ja määrää niiden suurimman yhteisen tekijän, voidaan pitää yleisempänä kuin kahden luvun suurimman yhteisen tekijän määrävää algoritmia. Yleisyys onkin eräs algoritmien hyvyyskriteeri.

Deterministisyysvaatimus on tiukka vaatimus, jota ei aina voida helposti saavuttaa edes tehtävämäärittelyä modifioimalla kuten yleisyyttä. Jotkin tehtävät ovat luonteeltaan aidosti epädeterministisiä, eikä niitä näin voi suoraan ratkaista algoritmisesti. Esimerkiksi jos tiedetään, että ratkaisu löytyy soveltamalla jotakin kolmesta käytettävissä olevasta keinosta, on ihmisen helppo ratkaista ongelma, mutta algoritmia ei voida kirjoittaa, ennen kuin on päätetty, missä järjestyksessä keinoja kokeillaan. Toimenpiteiden suoritusjärjestys on siis aina kerrottava algoritmista. Se on kerrottava siinäkin tapauksessa, että sillä ei olisi mitään vaikutusta lopputulokseen. (Ns. vapaajärjesteiset algoritmit muodostavat tässä suhteessa poikkeuksen.) Epädeterministisiä ongelmia esiintyy monissa tavallisissakin yhteyksissä. Mainittakoon esimerkiksi sellaisten pelien pelaaminen, joiden idea on juuri epädeterministisyydessä: kussakin pelitilanteessa tunnetaan tehtävissä olevat siirrot, mutta ongelma on, mikä niistä kulloinkin pitäisi valita. Epädeterminististen ongelmien algoritmiseen ratkaisemiseen ja vapaajärjesteisiin algoritmeihin palataan kuudennessa luvussa.

Oikeellisuusvaatimus on hyvin luonnollinen vaatimus. Se on kuitenkin myös hyvin vaikea saavuttaa. Yleensä on helppo nähdä, että algoritmi toimii *tavallisesti* oikein, mutta toimiiko se *aina* oikein? Oikeellisuudesta vakuuttautumiseen on kaksi peruskeinoa: testaus ja oikeellisuuden todistaminen formaalisesti. Pelkän testauksen avulla ei yleensä voida saavuttaa täydellistä luotettavuutta. Itse asiassa useiden käytössä olevien algoritmien oikeellisuudesta ei (voida) olla täysin varmoja. Tämä pätee etenkin kaikkein monimutkaisimpiin tietokoneohjelmiin. Kuitenkin algoritmi voi olla hyödyllinen, vaikka sen oikeellisuudesta ei oltaisi täysin vakuuttuneita. Tällöin pitää vain ymmärtää, että virhemahdollisuus on olemassa. Algoritmi voi jopa olla hyödyllinen, vaikka sen tiedettäisiin toimivan toisinaan väärinkin; etenkin jos tiedetään, miten usein ja milloin algoritmin voidaan odottaa antavan virheellisen tuloksen. Toisinaan täysin oikeata tulosta ei edes tavoitella, vaan tarpeeksi tarkka likiarvo riittää. Tällöin riittävä oikeellisuus saavutetaan korjaamalla tehtävämäärittelyä niin, että vaaditaan algoritmin tuottavan tuloksen, joka eroaa oikeasta (tarkasta) tuloksesta korkeintaan sallitun virheen verran. Algoritmien oikeellisuuteen palataan kolmannessa luvussa.

Toinen algoritmien tuloksellisuuteen kuuluva ominaisuus, terminoituvuus, on usein ilmeistä. Toisinaan algoritmin suorituksen ei ole tarkoitustaan päättävä, vaan algoritmi kuvaa ikuista, päättymätöntä prosessia. Päättymättömiä prosesseja ovat esimerkiksi kirjaston kirjaluettelon päivittäminen, joukkoliikennevälineiden ajanvaraus, pankkien maksuliikenteen hoito, oman terveydentilan säilyttämisestä huolehtiminen, liikennevalojen ohjaus, varhaisvaroitustutkien valvonta, potilaan valvonta sairaalan teho-osastolla jne. Vaikka tällaiset prosessit ovatkin kokonaisuudessaan päättymättömiä, ne tyypillisesti sisältävät useita päättyviä osaprosesseja. Algoritmin suoritus ei välttämättä aina pääty, vaikka suorituksen olisi tarkoitus päätyä. Näin voi käydä esimerkiksi algoritmista olevan virheen takia. Monimutkaisten algoritmien terminoitumisesta on joskus vaikea vakuuttautua. Lisäksi voidaan kysyä, miten pitkän ajan kuluessa algoritmin suorituksen tulee päättyä. Käytännössä useinkaan ei riitä, että suoritus päättyy joskus, mahdollisesti hyvinkin pitkän ajan kuluttua, vaan sen halutaan päättyvän jossakin määrääjässä. Jotkin tehtävät voidaan ratkaista hyvinkin monella eri tavalla, ja eräs luonnollinen kriteeri ratkaisujen vertailuun on juuri algoritmin suorittamiseen kuluva aika. Myös tehtäviä voidaan vertailla samalla kriteerillä: toisia tehtäviä ei kerta kaikkiaan ole mahdollista ratkaista yhtä tehokkaasti kuin toisia. Näitä laskettavuuteen ja kompleksisuuteen liittyviä asioita tarkastellaan lähempää kolmannessa luvussa.



## 2.2 Ohjelmointikieliet

Algoritmin suorittamiseksi se on esitettävä prosessorin ymmärtämässä muodossa. Tietokone ei pysty ymmärtämään luonnollisella kielellä, esimerkiksi suomen kielellä, kirjoitettuja algoritmeja. Luonnollisissa kielissä on useita ominaisuuksia, jotka tekevät ne huonoiksi algoritmien esittämiseen:

- laaja sanasto
- monimutkaiset kielioppisäännöt
- lauseiden merkitys riippuu kontekstista
- perustuu viime kädessä puhuttuun kieleen
- käyttö vaatii maailmankuvan

Ensimmäiset kaksi seikkaa eivät tietenkään ole esteitä, vaan ainoastaan hidasteita. Itse asiassa tietokone on erinomaisen hyvä monimutkaisten sääntöjen mukaisesti laadittujen rakenteiden kieliopillisesta oikeellisuudesta huolehtimiseen. Kielen monimutkaisuus vaikeuttaa algoritmien laatimista ja aiheuttaa helposti virheitä, lisäksi lopputuloksen selkeys kärsii. Laajaa sanastoa käytetään luonnollisessa kielessä vivahteiden erottamiseen kuvailevassa ilmaisussa. Hyvin määritellyssä tehtävässä myös vivahteet täytyy määritellä täsmällisesti, vuolaskaan kuvailu ei riitä. Näin ollen luonnollinen kieli on liian epämääräistä ja tarpeettoman monimutkaista.

Lauseiden merkityksen riippuvuus asiayhteydestä eli kontekstista antaa luonnolliselle kielelle monimerkityksisyyttä, jota voidaan hyödyntää esimerkiksi leikillisessä ilmaisussa. Algoritmien esittämisessä monimerkityksellisyys on yleensä vain haitallista. Kokonaan kontekstista vapaita eivät kaikki formaalisetkaan kielet ole. Esimerkiksi Pascal-kielessä esiintyvät symbolit : ja = tietyssä merkityksessä, mutta jos ne pannaan peräkkäin (:=), niiden merkitys muuttuu. Vastaavasti symbolilla + tarkoitetaan toisinaan kokonais- ja toisinaan reaali- ja kompleksilukujen (liukulukujen) yhteenlaskua, jotka toteutetaan tietokoneessa eri tavoilla. Jokaisen kirjoitelman merkitys on ohjelmointikielissä kuitenkin aina yksikäsitteinen ja helposti asiayhteydestä määrättävissä. Näin ei ole asia luonnollisissa kielissä: esimerkiksi sanan "kaivosaukko" merkitys voi olla hyvinkin vaikeasti selvitettävissä ("Oletko nähnyt kaivosaukkoa kaivosaukossa? – En, mutta kaivossa kyllä.").

Elekielen (body language) merkitys on luonnollisessa kielessä suuri, ja joissakin muissa kielissä ja kulttuureissa vielä suurempi kuin Suomessa. Eleitä ja ilmeitä ei tietenkään esiinny kirjoitetussa tekstissä – joskin tietyissä alakulttuureissa esimerkiksi välimerkkejä käytetään tietynlaiseen hymistelyyn, vaikkapa =:). Maailmankuvan tarve tekee luonnollisen kielen käsittelystä tekoälyongelman: reagoidakseen oikein koneen täytyisi ymmärtää, mitä tekstissä sanotaan. Useinkaan tekstissä ei sanota kaikkea tarpeellista, vaan viesti on tarkoitettu yhdistettäväksi vastaanottajan aikaisempaan tietämykseen aiheesta. Esimerkiksi preposition "by" merkitys englanninkielisissä lauseissa "the church was built by the river" ja "the church was built by the parishioners" nojaa täysin lauseissa tarkasteltujen olioiden (kirkko, joki ja seurakuntalaiset) entuudestaan tunnettuihin ominaisuuksiin (joki ei rakenna kirkkoja, mutta sijaitsee pysyvästi jossakin, päinvastoin kuin seurakuntalaiset, jotka toimivat mm. rakentajina, mutta eivät määritä sijaintia).

Algoritmit on siis esitettävä formaalisella kielellä, eli ohjelmana, joka on kirjoitettu jollakin ohjelmointikielillä. Erilaisia ohjelmointikieliä on satoja. Ohjelmointikielten lukuisuus johtuu osittain siitä, että ohjelmointi on verrattain uusi ilmiö, eikä ole löydetty

yhteisesti hyväksytyä parasta tapaa kirjoittaa ohjelmia. Eri sovellusalueilla käytettävät algoritmit poikkeavat merkittävästikin toisistaan, joten eräät ohjelmointikielien on suunniteltu juuri tietentyypisten algoritmien esittämiseen. Myös koneiden ominaisuudet vaihtelevat, joten ohjelmointikielten lukuisuus on perusteltua. Toinen syy uusien ohjelmointikielten syntyyn on halu kehittää entistä parempia kieliä. Tämä on tosin johtanut toisinaan myös "pyörän keksimiseen" uudelleen. Kuten luonnollisilla kielillä, myös ohjelmointikielillä on oma sanastonsa eli aakkostonsa ja kielioppisääntönsä, jotka ilmaisevat miten aakkostoa käytetään. Yleensä aakkosto muodostuu matemaattisista symboleista ja eri tarkoituksiin varatuista sanoista. Kielioppisäännöt ovat täsmällisiä ja riittävän yksinkertaisia, jotta tietokone pystyy tehokkaasti tulkitsemaan ohjelmat.

Ohjelmointikielen suunnittelussa tulisi ottaa huomioon seuraavat tavoitteet:

- Kielen tulee helpottaa algoritmien ilmaisemista tietokoneen ymmärtämässä muodossa.
- Kielen tulee olla helposti tietokoneen ymmärrettävissä.
- Kielen tulee olla helposti ihmisten ymmärrettävissä, jotta ohjelmia voidaan tarvittaessa helposti muuttaa.
- Kielen tulee minimoida virhemahdollisuudet ohjelmaa kirjoitettaessa.
- Kielellä kirjoitettua ohjelmaa tutkimalla pitäisi olla helppo vakuuttautua siitä, että ohjelma todella kuvaa juuri halutun prosessin.

## 2.2.1 Syntaksi ja semantiikka

Prossessorin tulee pystyä tulkitsemaan algoritmi, jos se aikoo suorittaa algoritmin kuvaaman prosessin. Toisin sanoen prosessorin tulee ymmärtää muoto, jossa algoritmi esitetään ja kyetä suorittamaan vastaavat operaatiot.

Algoritmin esityksen ymmärtäminen jakaantuu kahteen vaiheeseen:

1. Prossessorin tulee ymmärtää ne symbolit, joilla algoritmi esitetään, eli prosessorin tulee tuntea algoritmin esityksessä käytetyn kielen aakkosto ja kielioppi.
2. Prossessorin tulee ymmärtää jokaisen askeleen merkitys.

Kielen *syntaksilla* tarkoitetaan oppia kielen lauseiden muodosta eli kielioppisääntöjä, jotka määräävät miten kielen symboleja saa laillisesti käyttää. **Syntaksi määrää, millaisia ovat oikein muodostetut kielen lauseet.** Se vastaa siis kysymykseen: "Miten?". Ohjelma on syntaktisesti oikein, jos se noudattaa kielen syntaksia. Syntaksivirheellä tarkoitetaan kielen syntaksin rikkomista. Syntaksivirhe estää lauseen suorittamisen, ja usein yksikin virheellinen lause estää koko ohjelman suorittamisen — ainakaan kokonaisuudessaan. (Tämä riippuu ohjelmointiympäristöstä.)

**Esimerkki.** Syntaksivirheitä:

- a + = b (väärin Pascalissa, mutta oikein C:ssä)
- Te kuuntelee luentoa. (predikaatin muoto on väärä)

Kielen *semantiikalla* tarkoitetaan oppia lauseiden merkityksestä eli tietynmuotoisten ilmaisujen merkitystä kielessä. **Semantiikka määrää, mitä kielen lauseet tarkoittavat.** Se vastaa siis kysymykseen: "Mitä?". Ohjelmointikielien on suunniteltu siten, että ne ovat syntaksiltaan ja semantiikaltaan huomattavasti yksinkertaisempia kuin

luonnolliset kielet. Luonnollisesta kielestä poiketen syntaksi ja semantiikka pidetään ohjelmointikielissä erillään toisistaan. Algoritmin askeleet voivat olla syntaktisesti oikeita, mutta semanttisesti merkityksettömiä.

**Esimerkki.** Semanttisesti kelvollisia/virheellisiä lauseita:

- Apina söi banaanin. (sekä semanttisesti että syntaktisesti oikein)
- Banaani söi apinan. (syntaktisesti oikein, mutta semanttisesti väärin)
- Kirjoita vuoden ensimmäisen kuukauden nimi. (sekä syntaktisesti että semanttisesti oikein)
- Kirjoita vuoden 13. kuukauden nimi. (syntaktisesti oikein, mutta semanttisesti väärin)

Voidakseen tulkita algoritmin prosessorin tulee:

1. tuntea algoritmin kuvauksessa käytetyt symbolit
2. osata liittää kuhunkin askeleeseen sen merkitys suoritettavien operaatioiden muodossa
3. kyetä suorittamaan operaatiot

Ohjelman syntaksivirheet huomataan vaiheessa 1. Jotkin semanttiset virheet havaitaan vaiheessa 2, kun taas toiset niistä havaitaan vasta 3. vaiheessa. Esimerkiksi lauseen "Kirjoita vuoden n:n kuukauden nimi" semanttista järkevyyttä ei voida tarkastaa, ennen kuin tiedetään n:n arvo. Ohjelmointikielen kääntäjä suorittaa vaiheet 1 ja 2 muuttaen ohjelman jokaisen askeleen sopiviksi operaatioiksi, jotka prosessori osaa suorittaa.

Toisin kuin luonnollisten kielten, ohjelmointikielten syntaksi määritellään nykyään täsmällisesti. Aikaisemmin näin ei ollut, vaan kielen syntaksi vain kuvailtiin enemmän tai vähemmän kattavasti, ja kielen kääntäjän kirjoittajan huoleksi jäi päättää yksityiskohdista. 1960-luvulta lähtien kielten syntaksi on määritelty käyttäen *kielioppia*. Esimerkiksi standardin mukaisen Pascal-kielen syntaksi määritellään täsmällisesti kielioppisäännöillä, joita on noin 150. Sen sijaan niin luonnollisten kuin ohjelmointikieltenkään semantiikkaa ei (joitakin harvoja poikkeuksia lukuun ottamatta) kyetä määrittelemään täsmällisesti, vaan kielen lauseiden merkityksen osalta joudutaan edelleen tyytymään enemmän tai vähemmän täydellisiin kuvailuihin.

Jos ohjelma on syntaktisesti oikein, mutta antaa virheellisen tuloksen, on ohjelmassa *looginen virhe*. Loogiset virheet voidaan havaita esimerkiksi vertaamalla algoritmin tai sen osan tuottamaa tulosta odotettuun tulokseen. Tämä luonnollisesti edellyttää, että tulos tunnetaan tai voidaan laskea muulla tavoin kuin algoritmin itsensä avulla. Tästä syystä loogiset virheet ovat yleensä paljon vaikeampia havaita kuin syntaktiset tai semanttiset virheet. Loogisten virheiden välttämiseksi ohjelman semantiikka kannattaa aina tuoda mahdollisimman selkeästi esille, mm. ohjelmassa käytettävien tunnusten (kuten muuttujien nimet) ja ohjelmakoodiin sijoitettavien kommenttien avulla.

## 2.3 Algoritmien asteittainen tarkentaminen

Algoritmien suunnittelu on yleensä melko vaikeaa, ellei kyseessä ole triviaali tehtävä. Toisinaan algoritmin muodostaminen annettuun tehtävään on hyvinkin vaikeaa. Tietokonealgoritmien suunnittelun vaikeus aiheutuu paljolti siitä, että tietokoneelta puuttuu päättelykyky. Kaikkein yleisimpiä virheitä etenkin kokemattomien ohjelmoijien algoritmeissa on se, että algoritmi tuottaa sinänsä ihan oikean tuloksen, mutta ei täsmäl-

leen annettuun tehtävän, vaan vain melkein. Niinpä tässäkin korostuu tehtävän määrittelyn tärkeys. Jotta tehtävän voisi ratkaista algoritmisesti, pitää se ensin ymmärtää täysin. Aloittelevan ohjelmoijan kannattaa hillitä koodausintoaan ja varata jokaisen pienenkin ohjelmointiprojektin alusta **riittävästi aikaa tehtävän määrittelylle** tai määrittelyn tarkentamiselle.

Algoritmien suunnittelun tulee pohjautua johonkin hyväksi koettuun menetelmään. Eräs sellainen on **asteittain tarkentava menetelmä** (stepwise refinement), jossa algoritmi suunnitellaan kokonaisuudesta osiin edeten (top-down design). Menetelmässä alkuperäinen prosessi jaetaan muutamaaan osaan, joille jokaiselle voidaan laatia oma algoritminsa, joka on pienempi ja yksinkertaisempi kuin koko prosessia kuvaava algoritmi. Nämä alialgoritmit jaetaan edelleen yhä pienempiin ja pienempiin osiin, kunnes osat ovat riittävän pieniä suoraan sellaisinaan ratkaistaviksi. Tyypillisesti algoritmien abstraktiotaso laskee jokaisella tarkentamisella. Tehtävän ratkaisu suunnitellaan ensin hyvin korkealla abstraktiotasolla, ja tätä ratkaisua konkretisoidaan asteittain, kunnes esityksen abstraktiotaso on algoritmien suorittajalle sopiva. Tässä mielessä ohjelmointia voidaan pitää tehtävän muunnoksena ihmisen ajattelua vastaavalta korkealta abstraktiotasolta koneen kykyjä vastaavalle matalammalle abstraktiotasolle.

Asteittaisen tarkentamisen menetelmä liittyy läheisesti reduktiiviseen ongelmanratkaisuun. **Reduktiolla** tarkoitetaan sitä, että ongelma jaetaan hyvin määriteltyihin osiin, kukin osaongelma ratkaistaan erikseen (mahdollisesti samalla tavalla, so. reduktiolla), ja osaongelmien ratkaisuihin kootaan alkuperäisen ongelman ratkaisu. Tällaista lähestymistapaa voidaan luonnehtia myös analyyttiseksi ja modulaariseksi. **Modulaarisuudella** tarkoitetaan sitä, että kunkin osaongelman ratkaisu kootaan itsenäiseksi osa-algoritmiksi eli algoritmimoduuliksi. Modulaarisuudella onkin nykyään hyvin merkittävä asema algoritmien suunnittelussa. Modulaarisuuteen ja moduuleihin palaamme hieman tuonnempana.

**Esimerkki.** Tarkastellaan esimerkkinä tehtävää, jossa on suunniteltava pikakaakon keittoalgoritmi robotille. Tämä alkuperäinen tehtävä 'keitä kupillinen kaakaota' voidaan jakaa esimerkiksi kolmeksi osatehtäväksi seuraavasti:

- (1) Keitä vettä
- (2) Pane kaakaojauhoja kuppiin
- (3) Kaada vettä kuppiin

Tämä jako ei ole ehkä tarpeeksi yksityiskohtainen, joten jaetaan kukin vaihe (1) – (3) vielä edelleen:

- (1.1) Täytä vesipannu
- (1.2) Aseta pannu liedelle
- (1.3) Lämmitä, kunnes vesi kiehuu
- (1.4) Siirrä pannu lieden sivuun
- (2.1) Ota kaakaopurkki esiin
- (2.2) Pane purkista 3 lusikallista kaakaota kuppiin
- (2.3) Pane lusikka kuppiin
- (2.4) Pane kaakaopurkki pois
- ...

Edelleen jotkin askeleet vaativat lisätarkennusta:

- (1.1.1) Aseta vesipannu vesihanalle
- (1.1.2) Avaa vesihana
- (1.1.3) Odota, kunnes pannu on täynnä
- (1.1.4) Sulje vesihana

Askelten tarkentamista jatketaan siihen saakka, kunnes askeleet ovat niin tarkkoja, että robotti voi ne tulkita ja suorittaa. Käyttäessään asteittaista tarkentamista algoritmien

suunnittelijan on tiedettävä milloin tarkentaminen lopetetaan. Tarkentaminen lopetetaan silloin, kun algoritmin kukin askel on riittävän yksinkertainen, jotta algoritmin suorittaja osaa sen toteuttaa. Tämä tietenkin riippuu algoritmin suorittajasta, oli se sitten robotti, ihminen tai tietokone. Algoritmin tarkentaminen lopetetaan, kun algoritmi on suoraan esitettävissä halutulla (ohjelmointi)kielellä.

**Esimerkki**algoritmimme asteittaisen tarkentamisen lopputulos voisi näyttää vaikkapa seuraavalta:

- (1) Keitä vettä
    - (1.1) Täytä vesipannu
      - (1.1.1) Aseta vesipannu vesihanan alle
      - (1.1.2) Avaa vesihana
      - (1.1.3) Odota, kunnes pannu on täynnä
      - (1.1.4) Sulje vesihana
    - (1.2) Aseta pannu liedelle
    - (1.3) Lämmitä, kunnes vesi kiehuu
      - (1.3.1) Pane liesi päälle
      - (1.3.2) Odota, kunnes vesi kiehuu
        - (1.3.2.1) Odota, kunnes pannu viheltää
      - (1.3.3) Pane liesi pois päältä
    - (1.4) Siirrä pannu lieden sivuun
- jne.

## 2.4 Imperatiivinen paradigma

Perinteisesti algoritmeja on pidetty jollekin koneelle annettavina komentosarjoina, ja siksi ne on yleensä kirjoitettu käskymuotoisina. Tätä vallitsevaa ohjelmoinnin suuntausta eli paradigmaa kutsutaan *imperatiiviseksi* tai *proseduraaliseksi*: algoritmit kuvaavat määrättyssä järjestyksessä suoritettavia toimenpidesarjoja. Tässä materiaalissa emme käytä mitään tiettyä ohjelmointikieltä, vaan käytämme ns. *pseudokieltä*, jossa on mukana yleensä imperatiivisissa ohjelmointikielissä olevat rakenteet. Myöskään käyttämämme syntaksi ei ole niin tarkkaa kuin oikeissa kielissä, jolloin voimme keskittyä enemmän semantiikkaan. Imperatiivisen ohjelmoinnin peruskäsitteistä ja pseudokielen syntaksista löytyy tiivistetty yhteenvedo samalla [www-sivulla](#), josta tämä monistekin on ladattavissa. Joistakin algoritmeista annetaan esimerkinomaisesti myös Java-versio.

Lyhyesti sanottuna imperatiivinen algoritmi koostuu peräkkäisistä lauseista (käskyistä), joita ovat: *asetuslause*, *valintalauseet* (IF) sekä *toistolauseet* (FOR-, WHILE- ja REPEAT-lause). Lisäksi käytössä on syöttö- ja tulostuslauseita, joiden avulla voidaan esim. lukea arvo näppäimistöltä johonkin muuttujaan (esim. lue(x)) tai tulostaa arvoja esim. näytölle (esim. tulosta(x)). Muuttuja tarkoittaa muistipaikkaa, ja algoritmin kirjoittaja saa ottaa käyttöönsä muuttujia (väli- ja aputuloksia varten) niin paljon kuin hän haluaa. Alussa muuttujilla on tietyt (syöttö)arvot, joiden perusteella on tarkoitus laskea tehtävän tulos. Tavoitteena on kirjoittaa sellainen lauseiden jono, joka saa aikaan sen, että valitsemassamme muuttujassa tai lausekkeessa on tehtävän tulos.

Imperatiivisen ohjelmoinnin *tärkeät käsitteet* ovat *muuttuja*, *tyyppi*, *lauseke* (*aritmeettinen tai totuusarvoinen*), *lause* (*asetus, valinta, toisto*) ja *moduuli* (=metodi, aliohjelma): sen määrittely ja kutsu.

## 2.4.1 Muuttuja, tyyppi, lause, lauseke ja asetuslause

Imperatiivinen algoritmi keskittyy kuvaamaan sitä prosessia, jonka tuloksena halutut tulosteet syntyvät. Prosessin etenemisen apuna käytetään **muuttujia** (*variable*), tietyn merkityksen omaavia arvoja, jotka muuttuvat prosessin aikana. Imperatiivisessa paradigmassa muuttuja tarkoittaa tietokoneen muistipaikkaa. Algoritmin käskyt muuttavat muistipaikkojen sisältöä, ja prosessin eli algoritmin tila määräytyy muuttujien senhetkisten arvojen mukaan. Niinpä muuttujan arvon asettaminen tai muuttaminen on keskeinen operaatio. Itse asiassa koko ohjelma on jono muuttujien arvoihin kohdistuvia viittauksia ja niitä hyväksi käyttäviä arvojen muutoksia.

Imperatiivinen suuntaus on vahvasti sidoksissa käytettävän ohjelmointikielen pohjalla olevaan abstraktiin konemalliin, joten ohjelmoijan on ajateltava tehtävän ratkaisua täysin koneen ehdoin. Tämä ei ole ristiriidassa sen seikan kanssa, että monet imperatiiviset ohjelmointikieliset ovat hyvin korkealla abstraktiotasolla ja että ne ovat tarkoitukseensa hyvin tehokkaita. Lähestymistavan ongelmat liittyvät siihen, että imperatiivisten kielten synnyn aikaan koneet olivat etusijalla ja ohjelmoijat taka-alalla, kun taas nykyään tilanne on päinvastoin. Ei-imperatiivisten paradigmojen tärkein etu onkin siinä, että ne tukevat paremmin ihmisen ajattelumallia. Imperatiivisen paradigman perusristiriidasta on mahdotonta päästä kokonaan eroon, mutta siihen liittyviä ongelmia voidaan oleellisesti lieventää käyttämällä hyvin korkean abstraktiotason kieliä. Kun kone 'viisastuu', sitä on helpompi käskä, vaikka se pohjimmiltaan edelleenkin toimii tyhmästi, konemaisesti.

Imperatiivisen algoritmin yksittäisiä käskyjä tai komentoja sanotaan **lauseiksi** (statement). **Lausekkeella** (expression) puolestaan tarkoitetaan luku- tai muita arvoja, muuttujaviittauksia ja erilaisia laskutoimituksia +, -, \* (kertolasku) ja / (jakolasku) tai muita operaatioita sisältävää ilmausta (kirjoitelmaa), jolla on sen osista määräytyvä yksikäsitteinen arvo. Lausekkeen ja lauseen välinen ero on siis se, että lausekkeella on arvo, lauseella ei. Lauseketta ei voi käyttää yksinään, vaan sitä on käytettävä lauseen osana. Lausekkeiden asema tosin hieman vaihtelee eri kielissä.

**Asetuslauseella** eli **sijoituslauseella** annetaan muuttujalle arvo tai muutetaan olemassa olevaa arvoa. Asetuslauseen yleinen muoto on

**muuttuja := lauseke**

esim.  $x := x + 2 * \text{summa}$

missä **muuttuja** on sen muuttujan nimi (kirjain/numerojono; esim. x, x1 tai summa), jonka arvo halutaan asettaa, ja **lauseke** on tälle muuttujalle sopivan arvon tuottava lauseke. Asetuslauseen semantiikka on seuraava: ensin lasketaan asetuslauseen oikean puolen saama arvo, ja sitten se asetetaan vasemman puolen muuttujan (uudeksi) arvoksi. Jos asetettavalla muuttujalla oli ennestään jokin arvo, se häviää. *Oikealla puolella olevan lausekkeen jokaisella muuttujalla tulee olla arvo.* Tässä on hyvä huomata, että monissa ohjelmointikielissä (esim. Java ja C) käytetään asetusoperaattorin := sijasta yhtäsuuruutta = (joka on huono valinta, koska silloin yhtäsuuruuden testaamiseen tulee käyttää jotain muuta tapaa; esim. ==). Joissakin kielissä oletetaan muuttujilla olevan jokin oletusarvo (esim. 0), mutta koska tämä ei ole yleinen käytäntö, niin oletamme, että *muuttujilla ei ole alkuarvoa.*

Jos lausekkeen arvo on luku, sanotaan sitä **aritmeettiseksi lausekkeeksi**; esim.  $3 * x + y$ , jos muuttujat x ja y ovat lukuarvoisia. Jos taas lausekkeen arvo on totuusarvo tosi (true) tai epätosi (false), sitä kutsutaan **loogiseksi** tai **totuusarvoiseksi** tai **ehtolausekkeeksi**;

esim.  $a < b + 1$ . Totuusarvoisiin lausekkeisiin palataan valinta- ja toistolauseiden yhteydessä.

**Esimerkki.** Tarkastellaan asetuslausetta. 1)  $x := y$ . Tässä  $y$ :llä tulee olla arvo, jolloin se sijoitetaan muuttujan  $x$  arvoksi.

2)  $x := 2 * x + 3 * y$ . Jos  $x$  ja  $y$  ovat lukumuuttujia, niin  $2 * x + 3 * y$  on aritmeettinen lauseke, jonka arvo sijoitetaan muuttujan  $x$  arvoksi. Lausekkeen arvo lasketaan koulusta tutulla tavalla vasemmalta oikealle niin, että aritmeettiset operaattorit  $*$  ja  $/$  ovat vahvempia (eli ne lasketaan ensin) kuin  $+$  ja  $-$ . Lisäksi lausekkeessa voi käyttää sulkeita laskujärjestyksen osoittamiseen (esim.  $y := x * (x - 1)$ ).

Muuttujiin ja lausekkeisiin liittyy *tyypin* käsite. Muuttujan tyyppi kertoo millainen arvo kyseiseen muistipaikkaan tallennetaan, ja tyyppi määrittää myös arvon tallennusmuodon tietokoneen muistiin. Pseudokielessämme emme ilmoita muuttujien tyyppiä, vaan tyyppi määräytyy sen mukaan millaista tietoa muuttujaan asetetaan. Näin on mm. kielessä Python. Java on taas vahvasti tyypitetty kieli, eli kaikki tarvittavat muuttujat tulee aina esitellä, jolloin pitää määrittellä niiden tyyppi; esim. `int x` eli esitellään muuttuja  $x$ , johon voidaan tallentaa vain (tietyllä arvoaluevälillä olevia) kokonaislukuja. Päätyypit ovat luvut (kokonaisluvut ja desimaaliluvut), merkit, totuusarvot (tosi/true ja epätosi/false) ja teksti eli merkkijonot. Tyypin käsitettä käsitellään tarkasti myöhemmin luvussa 2.8.

## 2.4.2 Lauseiden suoritusjärjestys

Imperatiiviseen paradigmaan liittyvä perusominaisuus on toimenpiteiden suorituksen määrätty järjestys. Suoritusjärjestystä ei yleensä voi muuttaa muuttamatta algoritmin merkitystä. Perusperiaate on se, että toimenpiteet suoritetaan siinä järjestyksessä kuin ne on kirjoitettukin: peräkkäin. Tämä vaatimus on suora seuraus nykyisten tietokoneiden (ns. von Neumann-koneiden) toimintaperiaatteesta: prosessori noutaa käskyjä muistista ja suorittaa niitä yhden kerrallaan ja peräkkäin. Peräkkäisyyttä ei usein edes mielletä laskun kontrollia ohjaavaksi rakenteeksi, eikä sitä yleensä osoiteta mitenkään. Sen sijaan käskyjen peräkkäisyydestä poikkeava suoritusjärjestys (vaihtoehtoiset toiminnot, toimenpiteiden toistaminen) osoitetaan algoritmissa erityisin rakentein.

Toimenpiteiden suoritusjärjestyksellä on keskeinen sija proseduraalisessa ohjelmointi-ajattelussa. Imperatiivisen suuntauksen lisäksi on esitetty myös muita, ns. *vapaajärjesteisiä* tai *deklaratiivisia* ohjelmoinnin paradigmoja. Niitä tarkastellaan monisteen kuudennessa luvussa. Tässä luvussa keskitytään imperatiiviseen ohjelmointiin. Algoritmien esittämiseen käytetään imperatiivista pseudokieltä, jossa proseduurien rakenne ja käskyjen suoritusjärjestys ilmaistaan tarkasti ja formaalisesti, mutta elementaaristen toimenpiteiden sisältö kuvaillaan vapaasti luonnollisella kielellä kulloistakin tarkastelua vastaavalla abstraktiotasolla.

Peräkkäisyydelle luonteva vaihtoehto on rinnakkaisuus. Vapausasteet ovat kuitenkin rinnakkaisuudessa paljon peräkkäisyyttä suuremmat. Rinnakkaisesti toimivia koneita on paljon vaikeampi kehittää kuin peräkkäiskoneita, ja erilaisia arkkitehtonisia ratkaisuja tutkitaankin kiivaasti. Vakiintuneen suoritusmallin puutteessa ei rinnakkaisalgoritmeille ole kehitetty vielä yleisesti hyväksyttyä semantiikkaakaan. Jatkossa keskitytäänkin peräkkäisyyteen, ja rinnakkaisuuteen palataan lyhyesti monisteen lopussa.

## 2.5 Ohjausrakenteet

Algoritmi koostuu lauseista, jotka suoritetaan algoritmin rakenteen määräämässä järjestyksessä. Algoritmin perusohjausrakenteet (flow of control) ovat *peräkkäisyys* (sequential), *valinta* (branching) ja *toisto* (repetition, loop). Perusrakenteet kuvaavat järjestyksen, jossa algoritmin sisältämät toimenpiteet suoritetaan. Siksi niitä kutsutaan myös ohjaus- tai kontrollirakenteiksi. Algoritmit rakennetaan näitä perusrakenteita yhdistelemällä. Ohjausrakenteet peräkkäisyys, valinta ja toisto riittävät minkä tahansa algoritmin esittämiseen.

Seuraavassa kerrotaan tarkemmin näiden lauseiden **syntaksi** ja **semantiikka**. Lauseen syntaksin esityksessä lihavoidut sanat (ns. kielen varatut sanat) kuuluvat aina lauseeseen (eli niiden tulee olla aina mukana lauseessa), mutta muu osuus on kuvailevaa ja siihen kirjoitetaan kuvauksen mukaiset osat. Näissä osissa käytetään ilmauksia (kursivoitu) *ehto*, *toiminto*, *toiminto1* ja *toiminto2*. *Ehto* on totuusarvoinen lauseke eli lauseke, jolla on arvo tosi tai epätosi. Sen sijaan *toiminto* koostuu yhdestä tai useammasta lauseesta tai voi myös puuttua.

### 2.5.1 Peräkkäisyys

Algoritmi on peräkkäisten toimenpidekokonaisuuksien eli askelten jono. Nämä askeleet suoritetaan yksi kerrallaan. Jokainen askel suoritetaan täsmälleen kerran: mitään askelta ei toisteta eikä mitään askelta ohiteta. Askelten suoritusjärjestys on sama kuin niiden kirjoitusjärjestys. Viimeisen askeleen suoritus päättää algoritmin suorituksen.

Aiemmin kuvattu kaakaonkeittoalgoritmi on esimerkki peräkkäisyydestä. Peräkkäisyys ei yksinään riitä, vaan algoritmin yleisyysvaatimuksen mukaisesti on kyettävä toimimaan eri tilanteissa. Mitä tapahtuisi kaakaonkeittoalgoritmissa, jos kaakaopurkki on tyhjä? Pysähtyisikö robotti ihmettelemään vai tarjoaisiko se kupin kuumaa vettä? Entä miten robotti käsittelisi useamman kaakaokupin keittopyynnön? Keittäisikö se vettä erikseen kutakin kuppia kohden? Entäpä jos kaakaon halutaan lisätä sokeria, maitoa tai vaikkapa rommia?

### 2.5.2 Valintalauseet ja totuusarvoiset lausekkeet

Yleispätevän algoritmin kirjoittaminen pelkkää peräkkäisyyttä käyttäen ei onnistu. Saman ongelman ratkaisemiseksi tulee eri tilanteissa toimia eri tavoin, ja niinpä algoritmissakin on tyypillisesti varauduttava vaihtoehtoihin toimintoihin. Ongelman kussakin yksittäisessä tapauksessa on tehtävä *valinta* eri toimintavaihtoehtojen välillä. Valinta ei saa (eikä se itse asiassa voikaan) olla satunnainen, vaan sen tulee olla deterministinen, so. johonkin yksiselitteiseen valintaehtoon perustuva. Epädeterministisiin ongelmiin ja niiden ratkaisemiseen palataan monisteen kuudennessa luvussa.

Yksinkertaisin valinnan muoto on valinta jonkin toiminnon tekemisen ja sen tekemättä jättämisen välillä. Tällainen valintarakenne on IF...THEN-lause:

**IF** *ehto* **THEN** *toiminto* **ENDIF**      esim. IF  $a < b + 1$  THEN  $a := b + 1$  **ENDIF**

Valintarakenteen *ehto* on totuusarvoinen lauseke (so. arvo on tosi/true tai epätosi/false) ja se määrää tilanteen, jossa *toiminto* suoritetaan. Jos *ehto* pitää paikkansa, *toiminto*



suoritetaan. Muussa tapauksessa suoritusta jatketaan IF...THEN-lausetta seuraavasta lauseesta. Yleisempi valinta on sellainen, jossa valitaan kahden vaihtoehdoisen toiminnon välillä:

**IF** *ehto* **THEN** *toiminto1* **ELSE** *toiminto2* **ENDIF**

Jos *ehto* toteutuu, suoritetaan *toiminto1*, muutoin suoritetaan *toiminto2*. Joka tapauksessa suoritetaan tarkalleen toinen vaihtoehdoisista toiminnoista, minkä jälkeen jatketaan IF...THEN...ELSE-lausetta seuraavasta lauseesta.

Lauseet sisennetään usein niin, että niiden haarat tulevat selvästi ilmi, esimerkiksi:

```

IF ehto THEN
    toiminto1
ELSE
    toiminto2
ENDIF

```

Tässä *ehto* on siis totuusarvoinen lauseke (esim.  $a < b$ ), mutta se voi sisältää myös loogisilla operaattoreilla (ne kirjoitetaan pseudokielessämme kuvaavasti: **AND**, **OR** ja **NOT**) yhdistettyjä totuusarvoisia lausekkeita. Lausekkeissa voi olla myös **vertailuoperaattoreita** *suurempi*  $>$ , *pienempi*  $<$ , *suurempi tai yhtäsuuri*  $\geq$ , *pienempi tai yhtäsuuri*  $\leq$  ja *erisuuret*  $\neq$ . Lausekkeissa voi käyttää tarvittaessa myös sulkeita osoittamaan laskujärjestyksen.

Ohjelmointikielien syntaksi totuusarvoisten eli loogisten lausekkeiden suhteen on hyvin rajoitettua. Esimerkiksi emme voi kirjoittaa IF  $a < b < c$  THEN ... , vaan tämä tulee kirjoittaa muodossa IF  $a < b$  AND  $b < c$  THEN.... Operaattorit toimivat 'nimensä mukaisesti': esimerkiksi jos kaksi loogista lauseketta on yhdistetty AND-operaattorilla (edellä  $a < b$  AND  $b < c$ ), niin kyseinen lauseke on tosi vain, jos kumpikin lauseke on tosi. Vastaavasti lauseke " $a < b$  OR  $b < c$ " on tosi, jos  $a < b$  tai  $b < c$ , eli tällöin vain toisen tarvitsee olla tosi, jotta koko lauseke olisi tosi. Lisäksi esimerkiksi NOT ( $a < b$ ) tarkoittaa samaa kuin  $a \geq b$ .

Lauseet tulee jakaa riveille ja sisentää rivit niin, että lause on helposti ymmärrettävissä. Usein valintarakenne sisennetään siten, että lauseen aloittava IF- ja mahdollinen ELSE- sekä päättävä ENDIF-sana kirjoitetaan eri riveille sisentämällä ne alkamaan samasta sarakkeesta. IF-lauseissa (ja myös toistolauseissa) olevat *toiminto*, *toiminto1* ja *toiminto2* voivat olla mitä tahansa lausejonoja eli koostuvat yhdestä tai useammasta peräkkäisestä lauseesta, tai ne voivat olla myös tyhjiä. Kyseiseen kohtaan voidaan siis kirjoittaa mm. asetuslauseita, valintalauseita ja toistolauseita. **Näin ollen lauseet sisältävät osinaan lauseita.**

Monimutkaiset lauserakenteet on syytä varustaa **kommenteilla**. Ohjelman suorituksen kulkuun vaikuttamattomat, lukijalle tarkoitetut selitykset eli kommentit sijoitetaan tässä monisteessa merkkiparien (\* ja \*) väliin. Niiden välissä olevan tekstin on vain tarkoitus selventää algoritmin toimintaa, eikä sillä ole mitään vaikutusta algoritmin toimintaan.

**Esimerkki.** Esimerkkejä valintalauseen käytöstä 'elävässä elämässä'.

```
IF sataa THEN avaa televisio ELSE mene ulos ENDIF
IF sataa THEN
    avaa televisio
ELSE (* ei sada *)
    IF aurinko paistaa THEN
        lähde rannalle
    ELSE (* ei sada ja aurinko ei paista *)
        lähde terassille
    ENDIF
ENDIF
```

**Esimerkki.** 1) Tarkastellaan seuraavaksi tilannetta, jossa meillä on kaksi lukumuuttujaa  $x$  ja  $y$ , joihin on tallennettu kaksi lukua. Seuraavassa on IF-lause, joka sijoittaa näistä pienimmän arvon muuttujaan min. Jos luvut ovat yhtä suuria, niin muuttujaan min tallennetaan ko. luvun arvo.

```
IF x < y THEN min := x ELSE min := y ENDIF
```

Yllä on IF-lause, jonka THEN- ja ELSE-haara sisältävät asetuslauseen. Tämä voitaisiin kirjoittaa myös ekvivalentissa muodossa:

```
IF x >= y THEN min := y ELSE min := x ENDIF
```

2) Tarkastellaan seuraavaksi tilannetta, jossa meillä on kolme lukumuuttujaa  $x$ ,  $y$  ja  $z$ . Tehtävänä on tallentaa muuttujaan min pienimmän arvo. Koska prosessori voi verrata ainoastaan kahta lukua yhtäaikaa, ratkaisu perustuu tietenkin parittaisiin vertailuihin. Seuraavassa on IF-lause, joka sijoittaa näistä pienimmän arvon muuttujaan min.

```
IF x < y THEN
    IF x < z THEN min := x ELSE min := z ENDIF
ELSE (* nyt x >= y *)
    IF y < z THEN min := y ELSE min := z ENDIF
ENDIF
```

Edellä oleva on jo hiukan monimutkainen ja vaatii avuksi paperia ja kynää. Tehtävän ratkaisu voidaan kirjoittaa myös seuraavassa helpommin ymmärrettävässä muodossa:

```
min := x (* alkuasetus muuttujalle min, jota muutetaan alla tarvittaessa *)
IF y < min THEN min := y ENDIF
IF z < min THEN min := z ENDIF
```

**Esimerkki.** Tarkastellaan mitä seuraavat melko mielivaltaiset sisäkkäiset IF-lauseet saavat aikaan. Algoritmin toiminta on kommentoitu itse algoritmiin. Lopuksi  $b=1$  ja  $a=12$  eli  $a$ :ta ei muuteta.

```
a := 12
b := 5
IF a < b THEN
    a := a + b (* ei suoriteta *)
ELSE
    b := a + b (* tämä lause suoritetaan eli b=17 *)
ENDIF
IF a < b THEN (* ehto on tosi, joten suoritetaan vain alla oleva IF-lause *)
    IF b > 15 THEN b := 1 ELSE b := 2 ENDIF (* suoritetaan THEN-haara, jolloin b=1 *)
ELSE
    IF a > 15 THEN a := 1 ELSE a := 2 ENDIF (* tätä IF-lausetta ei suoriteta lainkaan *)
ENDIF
```

**Esimerkki.** Tarkastellaan seuraavaksi sitä, miten päätellään lukujen  $x$ ,  $y$  ja  $z$  suuruusjärjestys. Ratkaisussa asetetaan muuttujaan pienin näistä pienimmän luvun arvo, muuttujaan keskimäinen näistä keskimäinen ja muuttujaan suurin näistä suurimman luvun arvo. Koska prosessori voi verrata ainoastaan kahta lukua yhtäaikaa, ratkaisu perustuu parittaisiin vertailuihin. Ratkaisu on nyt huomattavasti monimutkaisempi kuin edellä ollut pelkän minimin määrittäminen. Itse asiassa näin monimutkaisia sisäkkäisiä IF-lauseita on jo erittäin vaikeata ymmärtää ja tällaisia tulisi välttää. IF-lauseen haaroihin on lisätty kommentteja helpottamaan lauseen ymmärrystä (lisää puuttuvat kommentit!), kuten monimutkaisissa algoritmeissa on tapana tehdä. Tämä voitaisiin ratkaista myös seuraavasti: kopioidaan muuttujien arvot apumuuttujiin, jotka lajitellaan järjestykseen. Jatkossa esitetään useampikin lajittelualgoritmi.

```

IF x < y THEN
  IF x < z THEN (* Tällöin x<y ja x<z, mutta y:n ja z:n suhdetta ei tiedetä *)
    pienin := x
    IF y < z THEN (* x<y, x<z ja y<z *)
      keskimäinen:=y
      suurin := z
    ELSE (* x<y, x<z ja y>=z *)
      keskimäinen := z
      suurin := y
    ENDIF
  ELSE (* x<y, x>=z *)
    pienin := z
    keskimäinen := x
    suurin := y
  ENDIF
ELSE (* x>=y *)
  IF y < z THEN
    pienin := y
    IF x < z THEN
      keskimäinen := x
      suurin := z
    ELSE
      keskimäinen := z
      suurin := x
    ENDIF
  ELSE
    pienin := z
    keskimäinen := y
    suurin := x
  ENDIF
ENDIF
ENDIF

```

Koska IF-lauseen päättää aina ENDIF-sana, IF-lauseen THEN- ja mahdollisen ELSE-osarakenteen alku- ja loppukohtat tulee yksikäsitteisesti määrättyä. Siitä huolimatta lauseet sisennetään helpottamaan koko lauseen ymmärtämistä (ks. ed. esimerkki).

Tällainen valintalause on kaikissa ohjelmointikielissä, joskin sen syntaksi vaihtelee. Seuraavassa annetaan esimerkin vuoksi IF-lauseen syntaksi Javalla ja Pythonilla, mutta jatkossa tarkastelemme vain pseudokieltä. Javassa IF-lause on muotoa

**if** (*ehto*) { *toiminto1* } **else** { *toiminto2* }      esim. **if** ( $x >= y$ ) {*min* = *y*;} **else** {*min* = *x*;}

Tässä tulee kirjoittaa myös sulkeet (, ) , { ja }, ja varatut sanat **if** ja **else** tulee kirjoittaa pienillä kirjaimilla. Pythonissa

```

if ehto:
    toiminto1
else:
    toiminto2

```

Tässä tulee kirjoittaa kaksoispisteet sekä sisennys ja riveillejako tulee tehdä yo. tavalla. Nimittäin Pythonissa ei ole käytössä lausesulkuja {, } ja ENDIF, jolloin THEN-haaran ja ELSE-haaran osat käyvät ilmi siitä, miten lause jaetaan riveille ja rivien sisennyksillä. Sen sijaan Javassa ja pseudokielessämme voidaan lauseet jakaa riveille ja sisentää vapaasti.

Huolellisesta esityksestä ja sisennyksistä huolimatta mutkikkaat valintarakenteet tekevät algoritmista helposti epäselkeän. Joissakin tapauksissa usean sisäkkäisen IF...THEN...ELSE-rakenteen käyttö on vältettävissä käyttämällä CASE-monivalintarakennetta:

```
CASE mikä OF
    tapaus 1: toiminto 1
    tapaus 2: toiminto 2
    ...
    tapaus n: toiminto n
    OTHER: toiminto n+1
ENDCASE
```

Tällainen CASE-rakenne on merkitykseltään sama kuin seuraava ns. ketjutettu IF-lause:

```
IF mikä = tapaus 1 THEN toiminto 1
ELSE IF mikä = tapaus 2 THEN toiminto 2
ELSE
    ...
    ELSE IF mikä = tapaus n THEN toiminto n
    ELSE toiminto n+1
ENDIF
ENDIF
ENDIF
ENDIF
```

CASE-rakenteen tarkka syntaksi ja semantiikka vaihtelevat eri ohjelmointikielissä. Esimerkiksi OTHER-osa voi olla paitsi eriniminen, myös pakollinen, vapaaehtoinen tai puuttua kokonaan. Vakavampia rajoituksia voidaan asettaa esimerkiksi mikä-osan ja tarkasteltavien tapausten muodolle. Tässä monisteessa käytetään kaikkia rakenteita vapaasti rajoituksitta, kunhan niiden täsmällinen merkitys käy selväksi. Usein mikä-osa on muuttuja tai lauseke, jolla on jokin arvo (joissakin kielissä sen tulee olla kokonaisluku).

**Esimerkki.** Algoritmi, joka selvittää, montako päivää on kuluva kuussa (muuttuja k) ja sijoittaa sen muuttujaan päivienmäärä. Karkausvuoden selvittämiseen tarvitaan myös vuosiluku (muuttuja v).

```
CASE k OF
    huhtikuu, kesäkuu, syyskuu, marraskuu: päivienmäärä := 30
    helmikuu:
        IF v on karkausvuosi THEN päivienmäärä := 29 ELSE päivienmäärä := 28 ENDIF
    OTHER: päivienmäärä := 31
ENDCASE
```

Karkausvuoden määrittämiseksi tulee kirjoittaa oma algoritminsa, mutta se sivuutetaan tässä yhteydessä.

### 2.5.3 Toistolauseet

Peräkkäisyys ja valintakaan eivät riitä algoritmeissa tarvittavan yleisyyden saavuttamiseksi. Nimittäin algoritmin pitää pystyä kuvaamaan rajoittamattoman monta erilaista prosessia. Riittävä yleisyys saavutetaan *toiston* (repetition) eli *iteration* avulla. Toistoa nimitetään myös *silmukaksi* (loop) tai *toistorakenteeksi*.

Tietyissä tilanteissa toisto voidaan ymmärtää lyhennysmerkintänä: jos jotakin algoritmin osaa halutaan toistaa useamman kerran, käytetään toistorakennetta sen sijaan, että toistettava osa eli toiston runko kirjoitettaisiin algoritmiin monta kertaa peräkkäin. Tällaista toistoa sanotaan *definiitiksi*: toistokertojen lukumäärä tunnetaan etukäteen, eikä se voi muuttua toistorakenteen suorituksen aikana. Aina ei kuitenkaan ole kysymys tällaisesta toistosta. Toisto voi määräytyä erityisen toistoehdon avulla niin, ettei toiston määrää tiedetä ennen toistorakenteen suorituksen aloittamista. Tällöin puhutaan *indefiniittisestä toistosta*. Tällainen dynaaminen toistorakenne, jossa toistojen lukumäärä määräytyy toiston aikana, lisää aidosti kielen ilmaisuvoimaa.

Etenkin indefiniittia toistorakennetta käytettäessä on huolehdittava siitä, että toisto päättyy joskus ja juuri oikeassa kohdassa. Yksi yleisimmistä virheistä, joka algoritmeissa esiintyy, on virheellinen toiston päättymisehto. Jos prosessi on päättymätön, ei toiston tietenkään ole tarkoituskaan päättyä. Tällöin prosessia kuvaavassa algoritmissa tulee käyttää silmukkarakennetta, josta toiston ikuisuus käy selkeästi ilmi.

Toistorakenteen runkoa merkitään seuraavassa lyhyesti merkinnällä toiminto. Se koostuu tyypillisesti useista lauseista.

#### Definiitti toisto (toistojen lukumäärä tiedetään etukäteen)

Definiitti toisto voidaan edelleen jakaa *yksinkertaiseen* toistoon:

**REPEAT N TIMES**

*toiminto* (\* silmukan runko, joka voi koostua useasta lauseesta \*)

**ENDREPEAT**

jossa *toiminto* pysyy jokaisella suorituskerralla samanlaisena, ja *askeltavaan* toistoon:

**FOR** jokaiselle listan L alkiolle **DO**

*toiminto*

**ENDFOR**

jossa *toiminto* suoritetaan kerran kutakin listan L alkiota kohden. Toiminnossa voidaan lisäksi viitata (ja tyypillisesti viitataankin) vuorossa olevaan listan alkioon, joten toistettava *toiminto* voi kohdistua eri objekteihin eri toistokerroilla. Yleensä merkitään tarkasteltavan listan alkiot näkyviin luettelona esimerkiksi seuraavalla tavalla:

**FOR**  $i := 1, 2, \dots, N$  **DO**

*toiminto*

**ENDFOR**

Tässä *toiminto* suoritetaan N kertaa siten, että ensimmäisellä kerralla  $i=1$ , toisella suorituskerralla  $i=2$  jne. Usein kirjoitetaan näkyviin vain silmukkamuuttujan  $i$  ensimmäinen ja viimeinen arvo: **FOR**  $i := 1$  **TO**  $N$ . Tällöin silmukkamuuttuja saa arvot ilmoitettujen arvojen välistä yhden välein. Jos silmukkamuuttujan askel poikkeaa ykkösestä, se

voidaan merkitä näkyviin. Jos alku- ja loppuarvo ovat samat, suoritetaan silmukka kerran. Jos alkuarvo on loppuarvoa suurempi, ei silmukkaa suoriteta kertaakaan.

Valitettavasti vain harvoissa ohjelmointikielissä yksinkertaisin toiston muoto, REPEAT N TIMES *toiminto* ENDREPEAT, on käytettävissä. Tällöin toisto tulee rakentaa FOR-toistorakenteen avulla, jolloin tulee ottaa käyttöön (silmukan rungon kannalta turha) silmukkalaskuri *i*, joka laskee toistojen lukumäärän:

**FOR** *i*:=1,2, ..., N **DO** *toiminto* **ENDFOR**.

Tässä *toiminto* suoritetaan N kertaa siten, että ensimmäisellä kerralla *i*=1, toisella kerralla *i*=2 jne., ja tyypillisesti toiminnossa käytetään hyväksi *silmukkamuuttujan i* arvoa.

### **Indefiniitti toisto (toistojen lukumäärää ei välttämättä tiedetä etukäteen)**

Indefiniitit toistorakenteet voidaan niin ikään jakaa kahteen osaan.

**Alkuehtoinen** toisto (pre-tested loop):

**WHILE** *ehto* **DO**  
*toiminto*  
**ENDWHILE**

Silmukan runkoa (*toiminto*) toistetaan **niin kauan kuin** *ehto* on voimassa; ei siis välttämättä kertaakaan, jos *ehto* on heti epätosi. *Toiminnon* suorituksen (eli silmukan rungon viimeisen lauseen suorituksen) jälkeen testataan *ehdon* voimassaolo ja *toiminto* suoritetaan uudestaan, mikäli *ehto* on voimassa. Toiston suoritus siis loppuu, kun *ehto* on epätosi. Näin ollen toistorakenteen rungossa tulee tehdä sellaisia toimenpiteitä, jotka vaikuttavat *ehdon* voimassaoloon.

**Loppuehtoisessa** toistossa (post-tested loop):

**REPEAT**  
*toiminto*  
**UNTIL** *ehto*

*toimintoa* toistetaan **kunnes** *ehto* on voimassa, siis vähintään kerran. Rungon suorittamisen jälkeen tutkitaan *ehdon* voimassaolo, ja jos *ehto* on voimassa, niin toistoa ei enää jatketa. Toiston suoritus siis loppuu, kun *ehto* on tosi. Lopetusehto toimii siis päinvastoin kuin WHILE-lauseessa. REPEAT-lauseen runko suoritetaan siis aina vähintään kerran, kun taas WHILE-lauseen runkoa ei suoriteta välttämättä kertaakaan (jos *ehto* on heti epätosi).

Loppuehtoisesta toistosta on olemassa myös toinen muoto (esim. Javassa), ns. DO...WHILE-lause:

**DO**  
*toiminto*  
**WHILE** *ehto*

jossa toiston loppumista kontrolloiva totuusarvoinen lauseke *ehto* tulkitaan päinvastoin verrattuna REPEAT-rakenteeseen: toisto loppuu, kun *ehto* ei ole voimassa.

Jokainen REPEAT...UNTIL-lause voidaan korvata DO...WHILE-lauseella. Nimittäin lauseet

**REPEAT** *toiminto* **UNTIL** *ehto*

ja

**DO** *toiminto* **WHILE NOT** *ehto*

ovat ekvivalentteja eli toimivat samalla tavalla.

Definiitti toisto voidaan aina korvata indefiniitillä toistolla. Päinvastainen ei luonnollisestikaan pidä paikkaansa. Itse asiassa **riittää, että ohjelmointikielissä on vain alkuehtoinen WHILE-lause (kuten aina onkin), koska sen avulla voidaan toteuttaa kaikki muut toistorakenteet.** Selvytyden vuoksi on kuitenkin aina syytä käyttää prosessia parhaiten kuvaavaa toiston lajia.

**Huom.** Tässä monisteessa käytettyä valinta- ja toistorakenteiden päättävän END-alkuisen sanan (esim. ENDIF, ENDFOR, ...), ns. lausesulun, sijasta lauseiden yhteenkuuluvuus ilmoitetaan ohjelmointikielissä eri tavoin: esimerkiksi sulkeilla { ja }, tai sanoilla BEGIN ja END tai vain sanalla END. Huomaa, että REPEAT...UNTIL- ja DO...WHILE-rakenteissa ei tarvita loppua ilmaisevia sanoja, koska näissä sanat UNTIL ja WHILE ilmaisevat rakenteen lopun.

Edellä IF-lauseen tapaan *toiminto* tarkoittaa taas yhtä tai useampaa lausetta, eli toistorakenteen rungossa voi olla esim. IF-lauseita, asetuslauseita ja myös toistorakenteita (sisäkkäiset silmukat!).

**Esimerkki.** Osoitteenhakualgoritmi, joka sisältää korkean tason komentoja, joita ei ole suoraan ohjelmointikielissä.

On annettu henkilön nimi, jonka osoite tulisi hakea listasta, joka sisältää nimiä ja osoitteita. Vaikka listan nimien määrä tunnettaisiinkin, ei tarvittavien toistokertojen määrää tiedetä, koska ei tiedetä, monesko nimi haettava nimi on listassa (jos ollenkaan). Näin ollen tulee käyttää indefiniittia toistoa. Käytetään alkuehtoista toistoa, jossa toiston loppumista testaava totuusarvoinen lauseke koostuu kahdesta ehdosta, jotka on yhdistetty AND-operaatiolla. Tämä yhdistetty lauseke on tosi, jos kumpikin ehdoista on tosi.

```

WHILE annettu nimi ei ole löytynyt AND lista ei ole loppu DO
  Ota listalta seuraava henkilö
  IF tämän henkilön nimi = etsittävän henkilön nimi THEN
    Ota henkilön osoite tästä listan kohdasta
  ENDIF
ENDWHILE
    
```

**Esimerkki.** Kokonaisluvun  $n$ ,  $n \geq 0$ , kertoman (merkitään  $n!$ ) määrääminen ja sen tallennus muuttujaan  $k$ . Kertoma määritellään seuraavasti:  $0! = 1$ ,  $n! = 1 \cdot 2 \cdot \dots \cdot n$ , kun  $n > 0$ ; esim.  $4! = 24$ . Algoritmi esitetään sekä FOR- että WHILE-lauseella. Tästä näet myös sen, miten jokainen FOR-lause voidaan muuttaa ekvivalentiksi WHILE-lauseeksi. Tämä algoritmi ei sisällä korkean tason komentoja kuten edellinen. Tässä oletetaan, että ennen algoritmin suoritusta muuttujan  $n$  arvona on jokin ei-negatiivinen kokonaisluku.

```

k := 1
FOR i := 1, 2, ..., n DO
  k := k * i
ENDFOR

k := 1
i := 1
WHILE i <= n DO
  k := k * i
  i := i + 1
ENDWHILE
    
```

Jos  $n=0$ , niin kummankaan algoritmin toistorakenteen runkoa ei suoriteta kertaakaan, eli algoritmit toimivat myös kun  $n=0$ . Yllä olevasta nähdään myös, miten jokainen FOR-lause voidaan muuntaa WHILE-lauseeksi. WHILE-lauseessa silmukkalaskurin  $i$  arvoa tulee kasvattaa yhdellä, kun taas FOR-lauseessa se on 'sisäänrakennettu'. Tosin joissakin kielissä (esim. Javassa) FOR-lauseessa silmukkalaskurin muutos tulee ilmaista WHILE-lauseen tapaan.

**Esimerkki.** Tärkeä esimerkki käsitteistä. Tarkastellaan yllä olevaa (oik. puol. n!) algoritmia lause lauseelta:

Lause  $k:=1$  sisältää arimeettisen lausekkeen (vakion 1), asetusoperaattorin  $:=$ , muuttujan  $k$  ja kyseessä on asetuslause.

Lause  $i:=1$  sisältää arimeettisen lausekkeen (vakion 1), asetusoperaattorin  $:=$ , muuttujan  $i$  ja kyseessä on asetuslause. Seuraavana on lause

```
WHILE i <= n DO
  k := k * i
  i := i + 1
ENDWHILE
```

Kyseessä on alkuehtoinen WHILE-toistolause. Sen rungossa on kaksi asetuslausetta, joista ensimmäinen  $k := k*i$  sisältää muuttujat  $k$  ja  $i$ , aritmeettisen operaattorin  $*$ , aritmeettisen lausekkeen  $k*i$  ja asetusoperaattorin  $:=$  (tässä myös asetuslauseen oikealla puolella olevat  $k$  ja  $i$  ovat aritmeettisiä lausekkeita). WHILE-sanaa seuraava lauseke  $i <= n$  on ehto- eli looginen lauseke ja se sisältää vertailuoperaattorin  $<=$ .

Esitetään vielä lopuksi sama algoritmi Javalla:

```
int k = 1;
int i = 1;
while (i <= n)
{
  k = k * i;
  i = i + 1;
}
```

**Esimerkki.** Tarkastellaan seuraavaa esimerkkiä toistorakenteesta, jonka opetuksena on se, että toiston jatkuvuutta määrittelevä ehto tulee laatia harkiten.

```
i:=1
summa:=0
WHILE i <> 100 (* toisto loppuu kun i=100 *) DO
  summa:=summa+i
  i:= i+2
ENDWHILE
```

Nyt  $i$  käy läpi vain parittomia arvoja, joten algoritmi ei pysähdy iankaan. Jos ehto muutetaan muotoon  $i < 100$ , tuloksena on summa  $1+3+5+ \dots +99$ , jolloin silmukasta poistuttaessa  $i$ :n arvo on 101.

**Esimerkki.** Lukujonon maksimin määrittäminen. Algoritmi perustuu seuraavaan ideaan. Olkoon muuttujan maksimiehdokas arvo aluksi lukujonon ensimmäinen luku. Kaikki loput luvut käydään läpi silmukassa, ja jokaisen luvun kohdalla tarkastetaan, onko tarkasteltava luku suurempi kuin nykyinen maksimiehdokas, ja jos se on, asetetaan se uudeksi maksimiehdokkaaksi. Lopuksi muuttujassa maksimiehdokas on lukujonon suurin luku. Käytämme apuna lisäksi muuttujaa seuraava, jonka arvona silmukassa on vuorollaan aina lukujonon seuraava luku.

```
maksimiehdokas := lukujonon ensimmäinen luku
DO
  seuraava := lukujonon seuraava luku
  IF seuraava > maksimiehdokas THEN maksimiehdokas := seuraava ENDIF
WHILE lukujonoa ei ole käyty loppuun
maksimi := maksimiehdokas
```

Esimerkin algoritmi toimii, jos annetussa lukujonossa on vähintään kaksi lukua. Jos lukuja on vain yksi, algoritmia suoritettaessa joudutaan virhetilanteeseen. Algoritmi onkin parempi toteuttaa käyttäen alkuehtoista toistoa:

```
maksimiehdokas := lukujonon ensimmäinen luku
WHILE lukujonossa on lukuja DO
  seuraava := lukujonon seuraava luku
  IF seuraava > maksimiehdokas THEN maksimiehdokas := seuraava ENDIF
ENDWHILE
maksimi := maksimiehdokas
```



Algoritmissa käytetään ilmausta 'lukujonon seuraava luku', eli meillä tulee olla jokin mekanismi, jolla päästään seuraavaan lukuun. Jos nimet ovat tallennettu taulukkoon, niin tämä käy helposti. Ohjelmointikielissä viitataan taulukon tiettyyn alkioon yleensä indeksin avulla (tarkastellaan tarkasti luvussa 2.8.2.2). Olkoot lukujonon luvut taulukossa, jonka nimi on Taulu (se on muuttuja, joka tarkoittaa koko taulukkoa) ja olkoon taulukossa  $N$  kappaletta lukuja. Tällöin merkintä  $Taulu[i]$ , missä  $i$  käy läpi indeksit  $1, 2, \dots, N$ , tarkoittaa taulukon Taulu  $i$ :nnettä lukua. Huomaamme myös, että emme tarvitse muuttujaa maksimiehdokas, koska voimme käyttää sen sijasta muuttujaa maksimi:

```

maksimi :=Taulu[1]
i := 1
WHILE i<N DO
    i := i+1
    IF Taulu[ i ] > maksimi THEN maksimi := Taulu[ i ] ENDIF (* tässä i käy läpi arvot: 2,...,N *)
ENDWHILE

```

**Esimerkki.** Alkulukutesti. Algoritmi tulostaa tiedon siitä, onko muuttujassa  $n$ ,  $n > 1$ , oleva kokonaisluku alkuluku. Alkulukuja ovat ykköistä suuremmat kokonaisluvut, jotka ovat jaollisia vain ykkösellä ja itsellään; esim. 2,3,5,7,11,13,17,19,23 ... ovat alkulukuja. Algoritmin idea on seuraava: tutkitaan toistorakenteessa, onko  $n$  jaollinen  $t$ :llä, kun  $t$  kulkee  $2, \dots, n-1$ . Tässä oletetaan, että osamäärän  $n/t$  jakojäännös osataan määrätä suoraan, mikä pätee kaikissa ohjelmointikielissä (esim. Javassa tämä on  $n\%t$ ). Heti kun jakojäännös on nolla, voidaan tutkiminen lopettaa ja päätellään, että  $n$  ei ole alkuluku. Jos taas mennään loppuun saakka, eli jako ei mene tasan  $t$ :n arvoilla  $2, 3, \dots, n-1$ , tiedämme, että  $n$  on alkuluku.

```

IF n = 2 THEN
    tulosta: n on alkuluku
ELSE (* nyt n>2 *)
    t := 2 (* ensimmäinen tekijäehdokka t=2 *)
    REPEAT
        jakojäännös:= jakolaskun n/t jakojäännös (* tässä t on enintään n-1 *)
        t := t + 1 (* seuraava tekijäehdokka *)
    UNTIL jakojäännös=0 OR t = n (* toisto loppuu, kun jakojäännös=0 tai t=n *)
    (* silmukan loputtua joko jakojäännös=0, jolloin n ei ole alkuluku, tai t=n, jolloin n on alkuluku *)
    IF jakojäännös = 0
    THEN (* huomaa tässä IF-lauseen osien sisennys! *)
        tulosta: n ei ole alkuluku
    ELSE (* jakojäännös ei ole 0 *)
        tulosta: n on alkuluku
    ENDIF
ENDIF

```

Huomaa, että tapaus  $n=2$  täytyy tutkia erikseen. Miten algoritmi toimisi, ellei näin meneteltäisi? Algoritmissa käytetään toiston lopetusehtona yhdistettyä OR-ehtoa (looginen tai), joka toteutuu, jos jompikumpi tai molemmat osat toteutuvat. Tämän yhdistetyn ehdon toteuduttua on vielä testattava, kumpi ehdon osa aiheutti yhdistetyn ehdon toteutumisen. Huomattakoon vielä, että tekijäehdokka ei itse asiassa tarvitsisi tutkia  $n-1$ :een saakka, vaan vähempikin riittäisi, mutta palataan siihen myöhemmin. Tässä on myös tärkeätä, että  $n$ :n arvo on vähintään 2, kun algoritmin suoritus aloitetaan. Mitä mahtaa tapahtua, jos  $n=1$  tai pienempi. Tähän asiaan palataan, kun puhutaan moduulien alkuehdosta.

**Esimerkki.** Kahden positiivisen kokonaisluvun  $x$  ja  $y$  suurin yhteinen tekijä  $\text{sy}(x,y)$ . Algoritmi perustuu seuraaviin tosiseikkoihin (voit ottaa ne faktana, ellet tunne matemaattisia perusteita):

- 1) jos  $x > y$ , niin  $\text{sy}(x,y) = \text{sy}(x-y,y)$ ,
- 2) jos  $x < y$ , niin  $\text{sy}(x,y) = \text{sy}(x,y-x)$  ja
- 3) jos  $x=y$ , niin  $\text{sy}(x,y)=x=y$ .

Esimerkiksi  $\text{sy}(3,1) = 1$ ,  $\text{sy}(8,6) = 2$  ja  $\text{sy}(11,13)=1$ , koska 11 ja 13 ovat alkulukuja. Tehtävä ratkeaa seuraavasti: vähennetään toistuvasti aina suuremmasta arvosta pienempi kunnes arvot ovat samat, joka on haettu suurin yhteinen tekijä. Seuraavan algoritmin suorituksen jälkeen  $x=y=\text{sy}(x,y)$ :

```
WHILE x <> y DO (* toistetaan niin kauan, kun x ja y ovat erisuuria *)
  IF x > y THEN x := x - y ELSE y := y - x ENDIF
ENDWHILE
```

Esimerkiksi jos  $x=24$  ja  $y=54$ , niin algoritmia suoritettaessa muuttujien  $x$  ja  $y$  sisällöt muuttuvat seuraavasti:  $y:=54-24=30$ ,  $y:=30-24=6$ ,  $x:=24-6=18$ ,  $x:=18-6=12$ ,  $x:=12-6=6$ . Tällöin  $x=y=6$ , joten silmukan suoritus loppuu.

## 2.5.4 Lajitteluesimerkki

Seuraavassa esimerkissä tarkastellaan esitettyjen ohjausrakenteiden ja *asteittain tarkentavan suunnittelumenetelmän* käyttöä lajittelualgoritmissa (sorting algorithm), joka järjestää (lajittelee) nimilistan aakkosjärjestykseen. Tällöin tuloslistassa aakkosjärjestyksessä ensimmäinen nimi tulee ensimmäiseksi, toinen toiseksi jne. Lajittelu on hyvin tyypillinen tietojenkäsittelytehtävä. Lajittelumenetelmiä on olemassa useita. Tässä esimerkissä tarkastellaan ns. kuplalajittelua (bubble sort). Vaikka tässä yhteydessä tarkastellaankin nimilistan lajittelua aakkosjärjestykseen, menetelmä soveltuu minkä tahansa tyyppisen tiedon lajitteluun haluttuun järjestykseen (esim. kokonaislukulistan lajittelu nousevaan tai laskevaan suuruusjärjestykseen). Kuplalajittelun idea on seuraava: Nimelistaa käydään läpi nimi nimeltä, ja jokaista nimeä verrataan listassa seuraavana olevaan nimeen, ja jos nimet ovat väärässä järjestyksessä, niiden paikka vaihdetaan keskenään. Lista käydään läpi toistuvasti, kunnes listassa ei tarvinnut tehdä enää yhtään vaihtoa. Tällöin lista on aakkosjärjestyksessä. Oletamme että listassa on vähintään kaksi nimeä, jotta listassa olisi jotain lajiteltavaa. Alustava lajittelualgoritmi on seuraavanlainen:

```
WHILE lista ei ole lajiteltu DO
  Käy lista läpi alusta loppuun vertailemalla aina kahta peräkkäistä nimeä ja vaihda niiden sisältö
  keskenään, jos ne ovat väärässä järjestyksessä
ENDWHILE
```

Tarkastellaan esimerkkinä seitsemän nimen listaa:

<u>Alkuperäinen nimilista</u>	<u>1. läpikäynti (4 vaihtoa)</u>	<u>2. läpikäynti (3 vaihtoa)</u>	<u>3. läpikäynti (2 vaihtoa)</u>
Jussi	Jussi	Fredi	Bertta
Kati	Fredi	Bertta	Fredi
Fredi	Bertta	Jussi	Jaana
Bertta	Kati	Jaana	Jussi
Sami	Jaana	Kati	Kati
Jaana	Mari	Mari	Mari
Mari	Sami	Sami	Sami

Ensimmäisellä läpikäynnillä vaihdetaan ensin Kati ja Fredi keskenään ja edelleen Kati ja Bertta, sitten Sami ja Jaana ja lopuksi Sami ja Mari. Koska lista ei ole vielä järjestyksessä, tarvitaan vähintään toinen läpikäynti. Siinä tehdään kolme vaihtoa: Jussi ja Fredi, Jussi ja Bertta, sekä Kati ja Jaana. Vielä tarvitaan yksi läpikäynti, joissa tehdään kaksi vaihtoa: Fredi ja Bertta sekä Jussi ja Jaana. Kolmannen läpikäynnin jälkeen lista on järjestyksessä.

Yllä esitetyn algoritmin esitys on epätarkka (sisältää runsaasti luonnollista kieltä). Tarkennetaan algoritmia hiukan ja otetaan käyttöön apumuuttuja nimi. Kuitenkin algoritmi on vieläkin liian epätarkka, jotta tietokone osaisi suorittaa sen (miksi?):

```

WHILE lista ei ole lajiteltu DO
  Aseta tarkasteltava nimi := listan ensimmäinen nimi
  REPEAT
    IF tarkasteltava nimi seuraa aakkosjärjestyksessä listassa seuraavana olevaa nimeä THEN
      (* nimet ovat väärässä järjestyksessä *)
      Vaihda näiden nimien paikka listassa keskenään
    ENDIF
    Aseta tarkasteltava nimi := listan seuraava nimi
  UNTIL lista on käyty loppuun
ENDWHILE

```

REPEAT-toisto voidaan määrittellä edellistä tarkemmin, jos tiedetään, että lajiteltavia nimiä on  $n$  kappaletta. Tarkoittakoon lisäksi merkintä  $\text{nimi}[i]$  nimilistan  $i$ :nnettä nimeä. Tarvitsemme siis lukumuuttujan  $i$ , joka saa arvot  $1, \dots, N$ , missä  $N$  on nimien lukumäärä.

```

WHILE lista ei ole lajiteltu DO
  i := 1 (* aloita listan ensimmäisestä nimestä *)
  REPEAT
    IF nimi[i] seuraa aakkosjärjestyksessä nimeä nimi[i+1] THEN
      Vaihda näiden nimien paikka listassa keskenään
    ENDIF
    i := i + 1 (* siirry seuraavaan nimeen *)
  UNTIL i = N (* ollaan listan lopussa *)
ENDWHILE

```

WHILE-toiston ehto on vielä huonosti määritelty: mistä prosessori tietää toiston alussa, onko lista jo lajiteltu vai ei? Tarvitaan vähintään yksi listan läpikäynti, jotta selviää, onko lista lajiteltu. Tähän sopii lopetusehtoinen toistorakenne. Otetaan lisäksi käyttöön apumuuttuja vaihtoja, joka kertoo kuinka monta vaihtoa yhden läpikäynnin aikana on suoritettu (jos läpikäynnin jälkeen muuttujan vaihtoja arvo on 0, tiedämme että lista on järjestyksessä). Lisäksi testi 'seuraa aakkosjärjestyksessä nimeä' voidaan monissa ohjelmointikielissä kirjoittaa käyttäen operaattoria '>'. Saamme siis:

```

REPEAT
  vaihtoja := 0 (* vaihtoja ei ole suoritettu *)
  i := 1 (* aloita listan ensimmäisestä nimestä *)
  REPEAT
    IF nimi[i] > nimi[i+1] THEN
      Vaihda näiden nimien paikka listassa keskenään
      vaihtoja := vaihtoja + 1 (* suoritettiin vaihto *)
    ENDIF
    i := i + 1 (* siirry seuraavaan nimeen *)
  UNTIL i = N (* ollaan listan lopussa *)
UNTIL vaihtoja = 0 (* ei vaihtoja tällä kierroksella, joten lista on järjestyksessä *)

```

Nyt algoritmi on niin tarkka, että se voidaan suorittaa tietokoneella, kunhan se ensin kirjoitetaan jollakin oikealla ohjelmointikielellä. Tämän esimerkkialgoritmin asteittainen tarkentaminen muutti alkuperäistä algoritmia suuresti. Tämä on hyvin tavallista algoritmien suunnittelussa: tehtyjä päätöksiä saatetaan joutua muuttamaan. Usein muutos yhdessä kohdassa aiheuttaa muutoksia myös jossakin toisessa kohdassa. Muutosten leviämisen rajoittamiseksi algoritmit kannattaa rakentaa pienistä hyvin määritellyistä paloista, algoritmimoduuleista. Modulaarinen rakenne antaa algoritmeille muitakin merkittäviä etuja, joita tarkastelemme seuraavaksi. Huomaa lisäksi, että edellä esitetty algoritmi toimii, olipa tarkasteltavassa listassa nimiä tai lukuja.

## 2.6 Modulaarisuus

Tässä kappaleessa tarkastellaan modulaarisuusperiaatetta, jonka ymmärtäminen on abstraktin, käsitteellisen ajattelutavan omaksumisen ohella koko opintojakson tärkeimpiä tavoitteita. Modulaarisuusperiaate on tietokoneista, ohjelmointikielistä ja oppisuuntauksista riippumaton yleisesti hyväksytty suunnitteluperiaate, joka liittyy paitsi algoritmien suunnitteluun, myös yleiseen ongelmanratkaisuun ja muuhunkin tietojenkäsittelyalaan, kuten tietokoneen rakenteen ja toiminnan suunnitteluun.

### 2.6.1 Abstraktiot

Ihmiselle on luonteenomaista hankkia ja kehittää tietämystään siten, että tarkasteltavasta asiasta kehitetään aluksi enemmän tai vähemmän puutteellinen käsitteellinen malli, jota sitten täydennetään tietämyksen karttuessa. Luotua mallia kutsutaan **abstraktioksi** ja mallin luomiseen liittyvää henkistä prosessia **abstrahoinniksi**. Vaikka tarkasteltavaan asiaan liittyvän ymmärryksen kehittymisen varhaisessa vaiheessa syntyvät abstraktiot usein ovat varsin epätarkkoja, niiden merkitys ei ole suinkaan vähäinen. Muodostetut abstraktiot näet ohjaavat käyttäjänsä uuden tietämyksen hankinnassa. Onhan tunnettua, miten uudesta asiasta muodostettu ensikäsitys voi ratkaisevasti edesauttaa tai vaikeuttaa asian oppimista. Abstraktin ajattelun kyky, kyky "nähdä metsä puilta", on taito, joka on eri ihmisillä erilainen. Onneksi sitä voi kehittää, koska siitä on hyötyä hyvin monilla elämänalueilla.

Abstraktion tulee olla riittävä, mutta ei välttämättä täydellinen kuvaus tarkasteltavan asian tai ilmiön rakenteesta, ominaisuuksista ja käyttäytymisestä. Tämä tarkoittaa, että malli saa olla — ja tyypillisesti onkin puutteellinen, mutta kaikki tarkastelun kannalta relevantit (eli merkitykselliset) yksityiskohdat on voitava esittää sen avulla. Itse asiassa on vain hyvä, jos irrelevantit (eli merkityksettömät) yksityiskohdat on karsittu mallista pois. Esimerkiksi jonkun yrityksen työntekijät voitaisiin karakterisoida tietyillä ominaisuuksilla (nimi, virka-asema, ...) ja näin työntekijän abstrakti malli koostuisi näistä ominaisuuksista. Eri yksityiskohtien merkityksellisyys (eli relevanssi) mallissa riippuu kulloisestakin näkökulmasta ja siitä yhteydestä, missä mallia kehitetään.

Abstraktion tärkeä ominaisuus on sen abstraktiotaso: mitkä kohdetodellisuuden yksityiskohdat ja ominaisuudet on piilotettu mallin sisälle ja mitkä näkyvät ulospäin. Abstraktiota voidaan ajatella kuorena kuvaamansa asian ympärillä. Jotta abstraktio voisi kommunikoida ulkomaailman kanssa, on kuoreen rakennettu portteja tai liittymiä. Ympäristö näkee abstraktiosta vain nämä portit, ei kuoren sisäisiä yksityiskohtia. Ulkoisen maailman kannalta abstraktio on yhtä kuin sille määritellyt liittymät. Liittymien tarkka määrittely onkin sängen tärkeä osa abstrahointia. Abstraktion sisäinen rakenne puolestaan suunnitellaan siten, että kommunikointi muun maailman kanssa käy pelkästään muiden abstraktioiden liittymien kautta. Puhutaan mallin sisäisestä ja ulkoisesta esityksestä tai kuvauksesta. Oleellista on, että muiden kuin abstraktion itsensä ei tarvitse tietää sen sisäisestä rakenteesta mitään, vaan abstraktion ulkoinen kuvaus riittää.

Tietojenkäsittelytieteessä abstrahointia käytetään jatkuvasti kaikilla osa-alueilla. **Koneabstraktioin** kuvataan erilaisia tietokonearkkitehtuureja, tietokoneen fyysistä organisaatiota. Tällä tarkoitetaan tarvittavien perusoperaatioiden suorittamiseksi vaadittujen tietokoneen komponenttien fysikaalista toteutusta ja komponenttien konfigurointia toisiinsa nähden. Koneabstraktiolla voidaan myös kuvata koneen loogista toimintaa,

jolloin puhutaan suoritusmallista. Suoritusmalli luo pohjan ohjelmointikielen käskyjen merkityksen ja laskun tilan määrittelylle. Laskun tila puolestaan antaa mahdollisuuden tarkastella konetta käsitteellisessä mielessä, oleellisesti sen fysikaalisia perusteita korkeammalla abstraktiotasolla. Suoritusmalliin palataan monisteen luvussa 4.

Laskun tila liittyy aina johonkin kohtaan algoritmissa. Sanotaan, että algoritmin kontrolli on siinä kohdassa. Algoritmin kontrollin esittämiseen ja tutkimiseen käytetään erilaisia **kontrolliabstraktioita**. Kontrolliabstraktiot ovat ohjelmointikielistä riippumattomia yleisiä rakenteita ja mekanismeja. Edellisessä kappaleessa tarkastellut ohjausrakenteet ovat juuri tällaisia. Kontrolliabstraktiot yhdessä suoritusmallin kanssa mahdollistavat matemaattisten ominaisuuksien antamisen ohjelmille. Tällöin voidaan ohjelman suoritusta mallintaa esimerkiksi logiikan tai algebran keinoin ja soveltaa näille jo olemassa olevaa välineistöä ohjelmien analysointiin.

Kolmas tietojenkäsittelytieteessä yleisesti käytettävä abstraktiolaji on **tietoabstraktio**. Ratkaistaessa ongelmia tietokoneella joudutaan päättämään, miten ongelma esitetään käytettävän ohjelmointikielen keinoin, millaiseen muistirakenteeseen asioita tallennetaan laskun edetessä, ja miten tulokset esitetään sisäisesti ja ulkoisesti, ts. muistissa ja tulostuslaitteella. Usein tietoabstraktiot samastetaan tietorakenteisiin, mutta oleellista on, että tietoabstraktio kuvaa paitsi tiedon rakenteen, myös sen operaatiot. Jokaiseen tietorakenteeseen liittyy joukko operaatioita, rakenteen käsittelemiseen tarkoitettuja tapoja. Tietoabstraktio kapseloi tiedon rakenteen ja sen käsittelyyn tarkoitettut operaatiot yhteen siten, että abstraktio tulee ulkoisesti täysin määritellyksi, kun sen operaatiot kuvataan, ja ulkomaailma pääsee käsiksi abstraktion sisältämään tietoon ainoastaan näiden operaatioiden kautta. Vain näin voidaan luotettavasti turvata tietoabstraktion sisältämien tietojen helppo, tehokas ja virheetön käyttö. Tietorakenteita ja -abstraktioita tarkastellaan lähempää hieman tuonnempana (kappaleessa 2.8).

Kontrolli- ja tietoabstraktiot sekä suoritusmalli muodostavat algoritmitutkimuksen 'latinan' – kielen, jota kukaan ei käytä jokapäiväisessä ohjelmoinnissa, mutta joka on korvaamaton algoritmien ja ohjelmointikielten yleisten ominaisuuksien ja lainalaisuuksien analysoimisessa ja kehittämisessä.

## 2.6.2 Moduulit

Aiemmin kuvattiin miten algoritmeja voidaan suunnitella asteittain tarkentavalla menetelmällä. Jokaisessa tarkennusaskeleessa algoritmi jaetaan pienempiin komponentteihin, alialgoritmeihin, jotka voidaan sitten edelleen määrittellä yksityiskohtaisemmin samaa asteittaisen tarkentamisen menetelmää käyttäen. Tarkentaminen päättyy, kun jokainen algoritmikomponentti voidaan suoraan muuttua mutkitta, **triviaalisti**, esittää käytettävällä kielellä siten, että prosessori osaa sen tulkita.

Tällaista ongelmanratkaisuperiaatetta, jossa alkuperäinen ongelma ratkaistaan jakamalla se osiin, etsimällä osaongelmille ratkaisut (mahdollisesti samalla periaatteella) ja kokoamalla osaratkaisusta alkuperäisen tehtävän ratkaisu, sanotaan **reduktiiviseksi**. Reduktiivisessa ongelmanratkaisussa kokonaisuuden osittamisessa tehtävät rajauspäätökset ovat tärkeitä. On pyrittävä löytämään tehtävälle mahdollisimman luonteva ja hyvin määriteltä jako itsenäisiin osatehtäviin. Itsenäistä algoritmikomponenttia, joka voidaan suunnitella käyttöyhteydestä riippumattomasti ja liittää mihin tahansa algoritmiin, jossa kyseinen osatehtävä esiintyy, kutsutaan **moduuliksi** (module). Moduuli on siis algoritmiabstraktio. Moduulia nimitetään myös **metodiksi** (method), rutiiniksi

(routine), alirutiiniksi (subroutine) tai **aliohjelmaksi** (subprogram). Tiettyä moduulia oman tehtävänsä ratkaisussa käyttävän algoritmin (moduulin) sanotaan kutsuvan tätä moduulia. Useista moduuleista muodostuvaa algoritmia sanotaan **modulaariseksi**. Modulaarisissa algoritmissa jokainen ei-triviaali moduuli koostuu pienemmistä moduuleista.

Yleisesti ottaen modularisointiin kuuluu oleellisena osana myös käsiteltävän **tiedon modularisointi**, jopa niin, että oliokeskeisessä ohjelmoinnissa (object oriented programming) modularisointi tarkoittaa juuri käsiteltävän tiedon ja siihen liittyvien operaatioiden modularisointia. Tässä monisteessa modularisoinnin käsite on liitetty algoritmiseen modularisointiin, ja tietojen ja niihin liittyvien operaatioiden modularisointiin palataan myöhemmin kappaleessa 2.8 Tieto- ja tallennusrakenteet.

Algoritmia asteittain tarkennettaessa on kussakin vaiheessa saatu aikaan moduulit, jotka kuvaavat prosessin tietyllä abstraktiotasolla. Tarkentamisen edetessä algoritmin abstraktiotaso laskee ja konkreettisuus kasvaa. Abstraktiotason laskiessa algoritmin yksityiskohtien määrä kasvaa. Moduulin yksityiskohtien lukumäärän kasvaminen hallitsemattoman suureksi estetään jakamalla kokonaisuutta kuvaava algoritmi useisiin pienempiin komponentteihin. Moduulien lukumäärän ja yksittäisten moduulien koon välillä onkin löydettävä käyttökelpoisimman ratkaisun tuottava kompromissi. Yksittäisen moduulin koko voi vaihdella tehtävästä ja jossain määrin myös algoritmin laatijan mieltymyksistä riippuen melko suurestikin, mutta hyvänä nyrkkisääntönä voidaan pitää sitä, että kerralla tarkasteltavaa osakokonaisuutta pitäisi pystyä konkreettisesti tarkastelemaan kerralla. Tämä tarkoittaa sitä, että algoritmimoduulin tulisi paperille kirjoitettuna mahtua yhdelle sivulle tai näkyä kokonaisuudessaan tietokonepäätteen kuvaruudulla. Yleensä aloittelijat ovat taipuvaisia tekemään liian vähän ja liian suuria moduuleja. Hyvän moduulin ei tarvitse sisältää kuin muutama rivi algoritmia, esimerkiksi yksi valinta- tai toistorakennekokonaisuus alku- ja lopputoimenpiteineen. Eikä esimerkiksi toiston runkoakaan ole syytä kasvattaa turhan monimutkaiseksi, vaan on parempi käyttää toistettavan toiminnon tarkempaan kuvaukseen alimoduuleja.

Algoritmin tarkennuksessa syntyvät komponentit on syytä rakentaa siten, että ne ovat mahdollisimman riippumattomia muusta algoritmista ja toisistaan. Näin ne voidaan suunnitella erikseen, ajattelemta, missä yhteydessä niitä tullaan käyttämään. Riittää, että tunnetaan tarkasti se tehtävä, joka algoritmikomponentin tulee ratkaista. Tämän ansiosta algoritmin komponenttien suunnittelutyö voidaan jakaa ajallisesti, paikallisesti ja myös eri henkilöille, ja komponentteja voidaan käyttää useissa eri algoritmeissa.

**Esimerkki.** Kaakaonkeittoalgoritmi. Tarkennettu versiomme pikakaakon keittämiseen tarkoitettua algoritmista on seuraava:

```

Aseta vesipannu vesihanan alle
Avaa vesihana
Odota, kunnes pannu on täynnä
Sulje vesihana
Aseta pannu liedelle
Pane liesi päälle
Odota, kunnes pannu viheltää
Pane liesi pois päältä
Siirrä pannu liedon sivuun
Ota purkki kaapista
Avaa purkin kansi
Ota lusikkaan kaakaojauhetta
Kaada lusikassa oleva jauhe kuppiin
Ota lusikkaan kaakaojauhetta
Kaada lusikassa oleva jauhe kuppiin
Ota lusikkaan kaakaojauhetta
Kaada lusikassa oleva jauhe kuppiin
Pane lusikka kuppiin
Sulje purkin kansi
Pane purkki kaappiin
Nosta vesipannu kupin yläpuolelle
Kallista pannua niin, että vesi valuu kuppiin
Odota, kunnes kuppi on täynnä
Suorista pannu vaakasuoraan
Laske vesipannu liedelle
    
```

Algoritmi on rakenteettomuudessaan kaikkea muuta kuin selkeä: sitä on tarkasteltava hyvän aikaa, jotta pääsee jyvälle siitä, mitä siinä oikein tapahtuu. 'Turhanpäiväiset' yksityiskodot sumentavat suuret linjat. Alunperinhän, korkeimmalla abstraktiotasolla, algoritmi laadittiin kolmiosaisena:

```

Keitä vettä
Pane kaakaojauhoja kuppiin
Kaada vettä kuppiin
    
```

Lopullisesta versiosta tämä selkeä jako on tyystin hävinnyt. Kuitenkin on hyvin tärkeää, että koneen lisäksi myös ihminen – algoritmin laatija – ymmärtää algoritmin, koska muutoin algoritmin virheettömyydestä vakuuttautuminen samoin kuin algoritmin myöhempi muuttaminen käyvät hyvin vaikeaksi. Modulaarista esitystä käyttäen voimme yhdistää ristiriitaiset tavoitteet: esittää algoritmi riittävän yksityiskohtaisella tasolla, jotta robotti osaa sen suorittaa, mutta samalla säilyttää kuvaus niin korkealla abstraktiotasolla, että ihminenkin sen ymmärtää.

```

Moduuli 'Valmista pikakaakaota'
    Keitä vettä
    Pane kaakaojauhoja kuppiin
    Kaada vettä kuppiin
Moduuli 'Keitä vettä'
    Täytä vesipannu
    Aseta pannu liedelle
    Lämmitä, kunnes vesi kiehuu
    Siirrä pannu liedon sivuun
Moduuli 'Täytä vesipannu'
    Aseta vesipannu vesihanan alle
    Avaa vesihana
    Odota, kunnes pannu on täynnä
    Sulje vesihana
Moduuli 'Lämmitä, kunnes vesi kiehuu'
    Pane liesi päälle
    Odota, kunnes vesi kiehuu
    Pane liesi pois päältä
    
```

Moduuli 'Vesi kiehuu'  
Pannu viheltää

Moduuli 'Pane kaakaojauhoja kuppiin'  
Ota kaakaopurkki esiin  
Pane 3 lusikallista kaakaota kuppiin  
Pane lusikka kuppiin  
Pane kaakaopurkki pois

Moduuli 'Ota kaakaopurkki esiin'  
Ota purkki kaapista  
Avaa purkin kansi

Moduuli 'Pane lusikallinen kaakaota kuppiin'  
Ota lusikkaan kaakaojauhetta  
Kaada lusikassa oleva jauhe kuppiin

jne.

Moduulien rajauksessa ja suunnittelussa on hyvä ottaa huomioon seuraavat seikat:

- **Tehtäväkohtaisuus:** yksi moduuli ratkaisee yhden (osa)tehtävän, ei useampia.
- **Luonnollisuus:** mahdollisimman luontevien osatehtävien erottaminen algoritmista tuo merkittäviä etuja paitsi lisääntyvän selkeyden muodossa, myös moduulien yleisen käyttökelpoisuuden paranemisena.
- **Sopiva abstraktiotaso:** moduuli on hyvä rakentaa alimoduuleista, jotka ovat keskenään suunnilleen samalla abstraktiotasolla.

Tehtäväkohtaisuus on ohjenuora, josta etenkin aloittelijan on hyvä pitää kiinni. Liian suuret ja monimutkaiset moduulit syntyvät usein juuri siten, että niille on kohdistettu useampia kuin yksi tehtävä. Tämä on tavallaan ymmärrettävää; ajatellaanhan usein, että yhdistämällä kahden tehtävän ratkaiseminen 'lyödään kaksi kärpystä yhdellä iskulla'. Kertasäästö kuitenkin koston myöhempinä menetyksinä moduulin yleisessä käyttökelpoisuudessa. Vain algoritmille asetettujen erityisten tehokkuusvaatimusten vuoksi kannattaa tehtäväkohtaisuudesta tinkiä.

Sopivan abstraktiotason löytäminen annetun tehtävän ratkaisua muodostettaessa on aloittelijalle tyypillinen ongelma, etenkin jos ratkaisu tulisi esittää yksittäisistä ohjelmointikielistä riippumattomana kuvauksena. Tähän on hyvin vaikea antaa yleispäteviä neuvoja. Aloittelija voi lohduttautua sillä, että asiaan tarvitaan näkemystä, joka kehittyy ajan mittaan. Opintoihin liittyvien harjoitustehtävien ratkaiseminen on tässä suhteessa avainasemassa. Aluksi voi koettaa valita moduulin abstraktiotason täysin 'hihasta ravistamalla' ja keskittyä valitun tason johdonmukaiseen ylläpitämiseen. Tämä tarkoittaa sitä, ettei samaan moduuliin yhdistetä hyvin korkean ja hyvin matalan abstraktiotason operaatioita, vaan tarvittavat toimenpiteet kuvataan kussakin moduulissa yhtenäisellä abstraktiotasolla. Sopivan abstraktiotason määrittäminen on keskeinen osa tehtävän määrittelyä tai saadun määrittelyn tarkentamista.

Toinen kaakaonkeittoesimerkistä erityisesti huomattava seikka on, ettei moduulin tarvitse välttämättä käsittää kokonaista robotille annettavaa käskyä – ohjelmointikielen lausetta – vaan moduuli voi tarkentaa myös käskyn itsenäistä osaa – lauseketta. Moduulissa 'Vesi kiehuu' veden kiehumisen odottaminen on erotettu kiehumisen havaitsemisesta, koska kiehumisen havaitsemista voidaan pitää itsenäisenä tapahtumana, johon ei välttämättä aina liity odottamista. Eri prosessorit voivat myös suorittaa veden kiehumisen havaitsemisen eri tavalla.



Modulaarisuuden tärkeimmät edut ovat selkeys, josta seuraa edelleen luotettavuus ja helpompi ylläpito, sekä moduulien *yleiskäyttöisyys* eli uudelleenkäytettävyys (reusability). Yleiskäyttöisyyttä voidaan tarkastella yhtäältä moduulin itsensä ja toisaalta kokonaisuuden näkökulmasta:

- **Siirrettävyys:** moduulia voidaan käyttää paitsi siinä yhteydessä, mihin se alun perin suunniteltiin, myös missä tahansa muussa yhteydessä, jossa sama osatehtävä esiintyy.
- **Korvattavuus:** moduuli voidaan kaikissa yhteyksissään korvata toisella (esimerkiksi tehokkaammalla) moduulilla, joka ratkaisee saman tehtävän ilman, että kokonaisuudessa mikään muu muuttuu.

Lienee turha todeta, että modulaarisuudesta saatavien etujen hyödyntäminen edellyttää tarkkaa ja huolellista tehtävämäärittelyä jokaisen osatehtävän kohdalla.

### 2.6.3 Parametrit

Modulaarisuus lisää jo sinänsä algoritmin selkeyttä. Moduulien rajauksella tähdätään selkeyden lisäksi erityisesti moduulien yleiskäyttöisyyteen. Moduulien yleiskäyttöisyyttä voidaan vielä huomattavasti kasvattaa parametrisoinnilla.

**Esimerkki.** Toimenpiteen 'Pane kolme lusikallista kaakaojauhetta kuppiin' tosiasiallinen sisältö ei juuri eroa toimenpiteestä 'Pane kaksi lusikallista kahvijauhetta kuppiin' tai 'Pane kymmenen lusikallista tapettiliisterijauhetta vesiämpäriin'. Niinpä toimenpiteille kannattaakin kirjoittaa yhteinen, abstrakti moduuli, jota kutsuttaessa spesifoidaan mitattavan jauheen laatu ja määrä sekä astian laatu. Moduulin abstrahoimiseksi tarvitsemme *parametreja*. Esimerkiksi:

```
Moduuli 'Pane n lusikallista x-jauhetta astiaan y'
  Ota x-jauhepurkki esiin
  REPEAT n TIMES Pane lusikallinen x-jauhetta astiaan y ENDREPEAT
  Pane x-jauhepurkki pois
```

```
Moduuli 'Pane lusikallinen x-jauhetta astiaan y'
  Ota lusikkaan x-jauhetta
  Kaada lusikassa oleva jauhe astiaan y
```

Moduulia käytetään esimerkiksi seuraavasti:

```
'Pane kolme lusikallista kaakao-jauhetta astiaan kuppi'
'Pane kaksi lusikallista kahvi-jauhetta astiaan kuppi'
'Pane kymmenen lusikallista tapettiliisteri-jauhetta astiaan vesiämpäri'
```

Moduulin määrittelyssä esiintyviä parametreja (esimerkissämme  $n$ ,  $x$  ja  $y$ ) sanotaan **muodolliseksi parametreiksi**. Muodolliset parametrit ovat muuttujia. Niitä käytetään osoittamaan abstraktin toiminnan kohdetta tai toiminnan osaa, joka voi moduulin kutsukerroilla olla erilainen. Moduulin kutsussa nämä muuttuvat osat kiinnitetään antamalla tapauskohtaiset **todelliset parametrit** (esimerkissämme arvot 2, 3, 10, 'kaakao', 'kahvi', 'tapettiliisteri', 'kuppi' ja 'vesiämpäri'). Prosessori tulkitsee moduulin kutsun korvaamalla ensin muodolliset parametrit vastaavien todellisten parametrien arvoilla ja suorittamalla sitten moduulin rungon. Todelliset parametrit sisältävät moduulille välitettävän tiedon, joka voi siis vaihdella eri kutsukerroilla riippuen siitä, mitä moduulilla halutaan tehdä (tai laskea).

Todellisissa ohjelmointikielissä moduuleilla on kirjoittajan valitsema yksikäsitteinen nimi, ja lisäksi tulee noudattaa kielen syntaksia, jotta kääntäjä huomaa, että nyt alkaa

moduulin määrittely (alla sana MODULE). Lisäksi tulee valita yleistävät parametrit, jotka luetellaan moduulin nimen jälkeen sulkeiden sisällä: esim. yllä olevan moduulin otsikkorivi

Moduuli 'Laita  $n$  lusikallista  $x$ -jauhetta astiaan  $y$ '

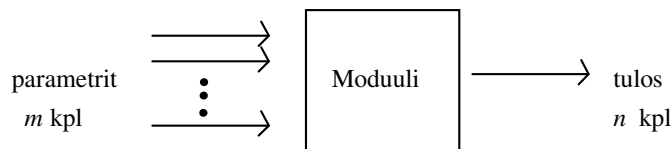
voitaisiin kirjoittaa muodossa

MODULE *laita*( $n, x, y$ ),

jolloin sen yllä oleva kutsu 'Laita *kolme* lusikallista *kaakao*-jauhetta astiaan *kuppi*' on muotoa *laita*(3, *kaakao*, *kuppi*).

## 2.6.4 Proseduurit ja funktiot sekä ohjelman rakenne

Matemaattisesti moduulia voidaan pitää funktiona, jolla on  $m$  syöttöparametria ja  $n$  tulosta.



Yleensä  $n=0$  tai  $n=1$ . Jos moduulilla on useita tuloksia, tulee niiden moduulin tehtäväkohtaisuusvaatimuksen nojalla muodostaa jokin kokonaisuus. Tällöin moduulilla voidaan ajatella monen tuloksen asemesta olevan yksi moniosainen tulos. Jotkin kielet tosin asettavat tällaisten rakenteisten tulosten käyttämiselle omat rajoituksensa.

Moduulia, jolla on tulos ( $n=1$ ), sanotaan *funktionaaliseksi* moduuliksi tai lyhyesti *funktioksi* (function). Jos  $n=0$ , sanotaan moduulia *proseduraaliseksi* moduuliksi, lyhyesti *proseduuriksi* (procedure). Tämän syntaktisen eron lisäksi nämä moduulityypit eroavat toisistaan myös semanttisesti.

Funktio-moduulin merkitys perustuu sen syöttöparametriensa perusteella tuottamaan **arvoon**. **Funktio palauttaa arvon ja funktion kutsu on siis lauseke**. Funktion palauttama arvo riippuu tietenkin parametreista. Funktion kutsu saa arvokseen sen arvon, jonka funktio palauttaa kutsussa annetuilla todellisilla parametreilla. Funktionaalinen algoritmimoduuli muuntaa syöttöarvonsa tulosarvoksi aivan kuten matemaattinen funktiokin. Matemaattisten funktioiden tavoin funktionaalisia moduuleja voidaan kytkeä peräkkäin siten, että moduulin tulos menee seuraavan moduulin syötteenä. Tämä vastaa funktioteoreettista *komposition* käsitettä. Funktioiden kompositioihin perustuvaa ohjelmointisuuntausta sanotaan funktionaaliseksi. Funktionaaliseen ohjelmointiin palataan myöhemmin monisteiden viimeisessä luvussa.

**Proseduuri ei palauta arvoa, joten myöskään proseduurin kutsulla ei ole arvoa, joten kutsu on siis lause**. Usein proseduurin aiheuttamaa vaikutusta sanotaan *sivuvaikutukseksi* varsinkin, jos se kohdistuu muihin objekteihin kuin pelkästään proseduurin parametreihin. Tällaisia ovat etenkin fyysiseen koneeseen kohdistuvat vaikutukset, kuten syöttö- ja tulostuslaitteiden ohjaukset. Nimitys sivuvaikutus kertoo myös siitä, että tällaisella moduulilla on laskennan matemaattiseen malliin vaikeasti istuvia ominaisuuksia. Proseduraalinen ohjelmointi onkin luonteeltaan nimenomaan imperatiivista, algoritmia suorittavan koneen käskemiseen perustuvaa.

Useimmissa tavanomaisissa (imperatiivisissa) kielissä on mahdollista kirjoittaa sekä funktionaalisia että proseduraalisia moduuleja. Koska funktionaalisen moduulin kutsu ei

ole kokonainen lause, vaan lauseke, ei sitä voi yleensä käyttää sellaisenaan, vaan funktion kutsun on aina esiinnyttävä jonkin lauseen, kuten asetuslauseen tai proseduurin kutsun osana. Monissa kielissä voidaan funktioita käyttää kuitenkin yksinään lauseen eli käskyn tavoin, jolloin funktion laskemaa arvoa ei käytetä eikä tallenneta mihinkään. Tämä on aloittelijan kannalta hyvin sekavaa, joten emme käytä pseudokielessämme tällaista tyyliä. Usein funktion kutsu esiintyy toisen funktion tai proseduurin kutsussa todellisena parametrina. Kielten suhtautuminen sivuvaikutuksien sallittavuuteen vaihtelee, mutta muut kuin moduulin parametreihin tai sen tulokseen kohdistuvat vaikutukset ovat erittäin vaarallisia, eikä niitä tulisi milloinkaan käyttää, vaikka käytettävä kieli sen sallisikin. Parametrien merkityksen olennainen osa on nimenomaan kertoa, mihin moduulin vaikutus kohdistuu.

Puhuttaessa moduuleista **tulee erottaa moduulin määrittely ja moduulin kutsu**. Moduulin määrittely (alkaa meidän pseudokielessä sanalla MODULE ja päättyy sanaan ENDMODULE) ei ole toiminnallinen, vaan nimensä mukaisesti se on vain määrittely. Tämän määrittelyn mukainen algoritmi voidaan suorittaa kutsumalla tätä moduulia jossain muussa moduulissa sen nimellä varustettuna moduulin todellisilla parametreilla. Moduulin kutsu (call, invocation) aiheuttaa moduulin suorittamisen tietyillä parametreilla. Kutsussa moduulin parametrit spesifioidaan, jolloin abstrakti parametrisoitu moduuli muuttuu **konkreettiseksi**, ja siitä tulee suorituskelpoinen.

## Funktionaalisen moduulin määrittely

Nyt olemmekin valmiit ottamaan käyttöön yleisen formaalisen esitystavan algoritmi-moduuleille. Funktion *määrittelyn* yleinen muoto on:

```

MODULE moduulin nimi (mp1, mp2, ..., mpN) RETURNS palautusarvo
    moduulin runko
ENDMODULE

```

missä *mp1, mp2, ..., mpN* ovat moduulin *muodolliset parametrit*, jotka ovat muuttujia. Termi muodollinen parametri viittaa siihen, että näillä parametreilla ei ole arvoa ennen kuin moduulia kutsutaan todellisilla parametreilla, joilla pitää olla arvot. Näin ollen tätä määrittelyä ei voi suorittaa – sehän on vain määrittely. *Palautusarvo* kuvaa arvoa, jonka moduuli palauttaa. Pseudokielessämme palautusarvo voidaan kuvata vapaasti luonnollisella kielellä, jotta moduulin tehtävä selviäisi jo moduulin otsikosta. Oikeissa ohjelmointikielissä tähän kirjoitetaan minkä tyyppisen arvon (esim. Javassa `int`, jos moduuli palauttaa kokonaislukuarvon) moduuli palauttaa. *Moduulin nimen* saa ohjelmoija valita vapaasti, mutta tietenkin pyritään valitsemaan nimi, joka kuvaa mahdollisimman hyvin moduulin toimintaa. Moduulin otsikko muodostaa moduulin ulkoisen kuvauksen perustan. Otsikon (rivi `MODULE ...`) lisäksi riittää *ulkoisessa kuvauksessa* selvittää ainoastaan moduulin muodollisten parametrien merkitys, ja se mitä moduuli tekee, mutta ei sitä, miten moduuli sen tekee (sisäinen kuvaus).

Moduulin runko puolestaan spesifioi sen, *miten* moduuli tehtävänsä ratkaisee. Se on siis algoritmiabstraktion sisäinen esitys. Moduulin runko koostuu lauseista, joissa voidaan viitata muodollisiin parametreihin, joilla runkoa suoritettaessa on kutsussa annettujen todellisten parametrien arvot. Moduulin sisäiset (lokaaliset) muuttujat ja niiden arvot eivät yleensä näy (pois lukien ns. globaalit muuttujat) moduulin ulkopuolelle, ja ne ovat olemassa vain kyseisen moduulin suorituksen ajan. Näin ollen eri moduuleissa voidaan käyttää samannimisiä muuttujia ja ne tarkoittavat kuitenkin eri muistipaikkoja.

Funktionaalisen moduulin rungossa on oltava yksi tai useampia RETURN-lauseita, joilla spesifioidaan moduulin palauttama arvo. RETURN-lauseen yleinen muoto on:

**RETURN** *lauseke*

RETURN-lause päättää moduulin laskun, antaa moduulin kutsulle lausekkeen määräämän arvon ja palauttaa kontrollin kutsukohtaan. Näin ollen funktion suorituksen tulee edetä RETURN-lauseeseen, jolloin funktion suoritus päättyy, ja funktion arvona on RETURN-lauseessa olevan lausekkeen arvo.

Eri kielissä funktiolle annetaan arvo eri tavoin. Jokaisessa kielessä on kuitenkin jokin tapa arvon antamiseen. Esimerkiksi Pascalissa funktiolle annetaan arvo asetuslauseessa, jossa funktion nimi esiintyy vasemmalla puolella, ja Eiffelissä on siihen tarkoitukseen erikseen varattu muuttuja Result. Huomattava periaatteellinen ero on kuitenkin siinä, suoritetaanko funktion runko aina loppuun vai eikö. Tässä monisteessa sovitaan, että kontrolli siirtyy kutsuvaan moduuliin heti kun kohdataan RETURN-lause (kuten esim. Javassa ja Pythonissa). Sen sijaan esimerkiksi Eiffelissä ja Pascalissa suoritetaan funktion runko aina loppuun, ja vasta silloin kontrolli siirtyy kutsuvaan moduuliin.

## Proseduraalisen moduulin määrittely

Sen yleinen muoto on:

```
MODULE moduulin nimi (mp1, mp2,..., mpN)  
    moduulin runko  
ENDMODULE
```

Otsikosta puuttuu palautusarvon kuvaus, koska moduuli ei palauta arvoa. Proseduurin runko on muuten samanlainen kuin funktion runko, mutta se ei sisällä RETURN-lauseita.

## Moduulin kutsu, parametrien välitys ja moduulin alkuehto

Funktion ja proseduurin kutsun ero on se, että funktionaalinen moduuli palauttaa arvon ja sen kutsun arvo on siis lauseke, kun taas proseduurin kutsu on lause eli käsky.

Niin funktionaalisen kuin proseduraalisenkin moduulin *kutsu* on muotoa:

```
moduulin nimi(tp1, tp2,..., tpN)
```

Kutsussa olevat *todelliset parametrit*  $tp1, tp2, \dots, tpN$  ovat lausekkeita, joille voidaan laskea arvot ennen moduulin suoritusta. Nämä todelliset parametrit välittyvät (usein kopioituvat) moduulin muodollisten parametrien arvoiksi, minkä jälkeen moduuli suoritetaan. Parametrien lukumäärän tulee olla sama kuin moduulin määrittelyssä olevien muodollisten parametrien lukumäärä, ja niiden tulee myös olla samassa järjestyksessä kuin vastaavat muodolliset parametrit moduulin määrittelyssä. Tietyissä tapauksissa muodollisen parametrin (esim. Javassa taulukon) muuttaminen moduulissa näkyy suoraan vastaavassa todellisessä parametrissa, jolloin ne tarkoittavat samaa muistipaikkaa. Tällöin ei siis suoriteta todellisen parametrin kopiointia moduulin muodollisen parametrin arvoksi. Usein kutsussa olevia todellisia parametreja kutsutaan myös moduulin *argumenteiksi*.

Seuraavassa emme yleensä kirjoita main-moduuleja emmekä puutu *parametrien välitysmekanismeihin*, jotka ovat erilaisia riippuen käytettävästä ohjelmointikielestä.

Kuitenkin seuraavassa esimerkissä kirjoitetaan Java-tyylinen main-moduuli ja selitetään parametrien välittymistä moduulien välillä. Tässä on hyvä huomata, että Javassa käytetään ns. *arvoparametrivälitystä*, joka on hyvin yksinkertainen: todellisen parametrin arvo (sillä siis pitää olla arvo!) kopioituu järjestyksessä vastaavan muodollisen parametrin arvoksi. Lisäksi kullakin moduulilla on omat alueensa omille muuttujilleen, jolloin kahden eri moduulin samannimiset muuttujat tarkoittavat eri muistialuetta ja sekaannuksen vaaraa ei ole.

Kaikki moduulit eivät ole tarkoitettut toimiviksi mielivaltaisilla parametrien arvoilla. Näin ollen moduulille tulisikin kirjoittaa kommenttina ennen otsikkoriviä ns. *alkuehto*, joka kertoo ne olosuhteet, joissa moduuli toimii oikein. Olosuhteet tarkoittavat yleensä rajoituksia parametrien arvoille. Alkuehto on tarkoitettu moduulin käyttäjälle ohjeeksi kertomaan sen, milloin moduuli toimii oikein. Systeemi ei (yleensä) tarkasta alkuehdon voimassaoloa, ja näin moduuli suoritetaan myös virheellisillä parametrien arvoilla, mutta tällöin algoritmi toimii väärin.

## Ohjelman rakenne

Tässä monisteessa sovitaan, että moduulin runko voi sisältää minkä tahansa moduulien kutsuja. Koska moduulit voivat viitata toisiinsa vapaasti, tällaista modulaarisuutta sanotaan *autonomiseksi*. Tällöin ohjelman suoritus aloitetaan kutsumalla jotain moduulia, joka sitten puolestaan kutsuu tarvitsemiaan moduuleja. Useimmissa kielissä tällaisella "päämoduulilla" on erityinen nimi (esimerkiksi Javassa main). Rajoittuneempi modulaarisuuden muoto on *hierarkkinen*, missä moduulin määrittelyt ovat sisäkkäisiä ja moduuli voi käyttää vain hierarkian samalla tasolla tai ulompana määriteltyjä moduuleja. Hierarkkiseen modulaarisuuteen kuuluu myös ns. *päämoduulin* eli *pääohjelman* (main program) käsite. Pääohjelma sisältää tyypillisesti useiden muiden moduulien kutsuja. *Nämä alimoduulit eivät ole yksinään suorituskelpoisia, vaan ohjelma suoritetaan aina suorittamalla pääohjelma*. Esim. Javassa tulee suorittaa tiedosto, joka sisältää main-moduulin (tosin Javassa puhutaan metodeista eikä moduuleista). Tässä materiaalissa kirjoitamme pääohjelman eli main-moduulin muodossa:

```

MODULE main()
    moduulin runko
ENDMODULE

```

**Esimerkki.** Kokonaisluvun  $n$ ,  $n \geq 0$ , kertoman määräävä moduuli ja sitä kutsuva main-moduuli. Moduulin tehtävänä on palauttaa kertoman arvo, joten moduulista on syytä tehdä funktio. Moduulin parametrina on tarkasteltava luku  $n$ . Toteutetaan algoritmi hiukan eri tavalla kuin aiemmin. Kerrotaan muuttuja  $k$  alenevasti arvoilla  $n, n-1, \dots, 2$ . Tässä käytetään kertojana muuttujaa  $n$ , jonka arvoa pienennetään yhdellä joka kierroksella.

```

(* Alkuehto: n >= 0 *)
MODULE kertoma(n) RETURNS n!
    k := 1
    WHILE n > 1 DO
        k := k * n
        n := n - 1
    ENDWHILE
    RETURN k
ENDMODULE

```

Vertaa moduulin määrittelyä aiemmin esitettyyn algoritmiin. Kutsu kertoma(0) palauttaa luvun 1, ja kutsu kertoma(5) palauttaa luvun 120. Moduuli kertoma on funktionaalinen, joten sen kutsu on lauseke (so. sillä on arvo), ja kutsu voidaan sijoittaa sellaiseen kohtaan, jossa syntaksin mukaan tulee olla lauseke, esim. asetuslauseen oikealle puolelle tai tulostuslauseeseen. Moduulin alkuehto vaatii, että  $n \geq 0$ . Tässä on hyvä huomata, että voimme kyllä suorittaa esim. kutsun kertoma(-1), jolloin moduuli palauttaa arvon 1 (WHILE-silmukka ohitetaan), joka on virheellinen.

Alla on Java-tyylinen main-moduuli, joka sisältää esimerkinomaisesti moduulin kutsuja. Olkoon lisäksi moduuli tulosta proseduri, joka vaatii yhden parametrin, jonka arvo tulostetaan. Tällainen moduuli on kaikissa ohjelmointikielissä valmiina.

```
MODULE main()
  x := kertoma(5)
  tulosta(x)
  k := 4 (* tämä on eri muuttuja kuin moduulin kertoma muuttuja k *)
  tulosta(x * kertoma(k) + kertoma(kertoma(k-1)))
ENDMODULE
```

Ohjelman suoritus alkaa main-moduulista. Tällöin lasketaan ensin lausekkeen kertoma(5) arvo. Arvoparametrivälitystavassa todellinen parametri 5 kopioidaan muodollisen parametrin n arvoksi, jolla moduuli kertoma suoritetaan. Moduulissa kertoma olevat muuttujan n ja k ovat olemassa vain moduulin kertoma suorituksen ajan, ja kun palataan kutsuvaan main-moduulin, niitä ei ole enää olemassa. Moduulin kertoma lopuksi suoritetaan lause RETURN k (missä k on nyt 120), jolloin palataan main-moduulin ja kutsu kertoma(5) korvataan palautusarvolla 120, joka sijoitetaan muuttujan x arvoksi. Sen jälkeen suoritetaan lause tulosta(x), joka tulostaa kuvaruudulle arvon 120. Seuraavaksi asetetaan muuttujan k arvoksi 4. Sen jälkeen tulee tulostuslause, jossa on useita moduulin kertoma kutsuja. Kutsussa kertoma(kertoma(k-1)) lasketaan ensin sisempi kutsu kertoma(k-1) eli kertoma(3), joka palauttaa arvon 6, ja sen jälkeen lasketaan kertoma(6). Metodi tulostaa lausekkeen  $120 \cdot 24 + 720$  arvon eli luvun 3600.

Tällä kurssilla ei ole tarkoituksena opetella Javaa, mutta seuraavaksi esitetään esimerkinomaisesti sama ohjelma Javalla. Sisennykset helpottavat vain lukemista, ja ne eivät ole pakollisia (mutta ohjelman luettavuuden kannalta kuitenkin välttämättömiä). Sen sijaan esim. aaltosuljemerkit ovat pakollisia. Merkkipari // tarkoittaa kommentin alkamista. Kielen syntaksin mukaiset varatut sanat on tapana lihavoida, kuten seuraavassakin tehdään. Moduulit (eli metodit, kuten Javassa sanotaan) esitetään Javassa luokan sisällä ja sijoitetaan fyysisesti usein samaan tiedostoon, joka voisi näyttää seuraavalta:

```
class Kertoma
{
  public static int kertoma(int n) // tässä int on funktion palautustyyppi
  {
    int k = 1;
    while (n > 1)
    {
      k = k * n;
      n = n - 1;
    }
    return k;
  } // kertoma-moduulin loppu

  public static void main(String[] args) // tässä void kertoo, että kyseessä on proseduri
  {
    int x = kertoma(5);
    System.out.println(x);
    int k = 4; // tämä on eri muuttuja kuin moduulin kertoma muuttuja k
    System.out.println(x * kertoma(k) + kertoma(kertoma(k-1)));
  } // main-moduulin loppu
} // koko luokan loppu
```

**Esimerkki.** Moduuli, joka tulostaa kertomat  $0!$ ,  $1!$ ,  $2!$ , ...,  $n!$ , kun n on moduulin parametri. Moduuli laskee useita kertoman arvoja, mutta ei palauta niitä kutsuvaan moduulin, vaan tulostaa ne. Näin ollen sen merkitys perustuu konstruoidujen arvojen tulostamiseen, siis tulostuslaitteeseen kohdistuvaan sivuvaikutukseen. Täten moduulista on syytä tehdä proseduri. Kertoman arvot voidaan laskea yllä olevalla kertoma-funktiolla.

```
(* Alkuehto: n >= 0 *)
MODULE kertomat(n)
  FOR i := 0, 1, ..., n DO
    tulosta(kertoma(i))
  ENDFOR
ENDMODULE
```

Esimerkiksi jos  $n = 5$ , moduuli tulostaa arvot 1, 1, 2, 6, 24 ja 120.

Kutsuvassa moduulissa kertomat-moduulin kutsu on kuin mikä tahansa käsky. Esimerkiksi kutsu

```
kertomat(1+3)
```

tulostaa arvot 1, 1, 2, 6 ja 24.

**Esimerkki.** Kahden positiivisen kokonaisluvun  $x$  ja  $y$  suurimman yhteisen tekijän määräävä algoritmimoduuli.

```
(* Alkuehto: x ja y ovat positiivisia kokonaislukuja *)
MODULE syt(x, y) RETURNS x:n ja y:n suurin yhteinen tekijä
  WHILE x <> y DO
    IF x > y THEN
      x := x - y
    ELSE
      y := y - x
    ENDIF
  ENDWHILE
  RETURN x
ENDMODULE
```

Esimerkiksi  $\text{syt}(3,1) = 1$ , ja  $\text{syt}(8,6) = 2$ . Huomaa, että yhtäsuuruusmerkin = käyttö on tässä yhteydessä matemaattisesti täysin korrektaa: onhan tarkoituksena nimenomaan lausua kahden arvon, syt-funktiomodulin kutsun ja lukuvakion, yhtäsuuruus.

**Esimerkki.** Funktio alkuluku. Tähän saakka on tarkasteltu vain tilanteita, jossa funktio laskee ja palauttaa vain lukuarvoja, mutta muuhunkin on tarvetta. Aiemmin esitettiin algoritmi, joka tutkii, onko annettu luku  $n$  alkuluku ja tulostaa tiedon siitä. Tällainen algoritmi ei tietenkään ole yleiskäyttöinen. Nimittäin usein on tarve vain tutkia, onko annettu luku alkuluku vai ei, minkä jälkeen saatua tietoa käytetään hyväksi (esim. jossain muussa moduulissa). Tällöin siis tarvitaan funktio, joka palauttaa totuusarvon eli arvon tosi (true) tai epätosi (false). Itse algoritmi on tietenkin sama, mutta nyt tulee palauttaa totuusarvo. Käytetään tässä kuitenkin DO...WHILE-toistoa edellä olleen REPEAT-toiston sijasta.

```
(* Alkuehto: n on vähintään 2 *)
MODULE alkuluku(n) RETURNS totuusarvon sen mukaan onko n alkuluku vai eikö
  IF n = 2 THEN RETURN true
  ELSE (* n > 2 *)
    t := 2
    DO
      jakojäännös := jakolaskun n/t jakojäännös
      t := t + 1
    WHILE (jakojäännös <> 0) AND (t < n)
    IF jakojäännös = 0 THEN
      RETURN false
    ELSE
      RETURN true
    ENDIF
  ENDIF (* IF n=2 ... *)
ENDMODULE
```

Koska funktion arvo on false heti, kun jakojäännös on nolla, niin voimme kirjoittaa lauseen RETURN false myös silmukan sisälle. Jos taas silmukka suoritetaan loppuun saakka, niin funktion arvo on true, jolloin jako ei ole mennyt tasan millään arvoista  $2, \dots, n-1$ .

```
(* Alkuehto: n on vähintään 2 *)
MODULE alkuluku(n) RETURNS totuusarvon sen mukaan onko n alkuluku vai eikö
  IF n = 2 THEN RETURN true
  ELSE (* n>2 *)
    t := 2
    DO
      jakojäännös := jakolaskun n/t jakojäännös
      IF jakojäännös = 0 THEN RETURN false ENDIF
      t := t + 1
    WHILE t < n
  ENDIF
  RETURN true
ENDMODULE
```

Tätä moduulia voidaan käyttää jossain muualla esim. seuraavasti:

- IF alkuluku(k) = true THEN ...
- IF alkuluku(k) = false THEN ...

tai mielummin:

- IF alkuluku(k) THEN ...
- IF NOT alkuluku(k) THEN ...

Huomaa, että totuusarvoisilla lausekkeilla 'alkuluku(k)' ja 'alkuluku(k) = true' on sama totuusarvo.

**Esimerkki.** Kilpikonnagrafiikkaa. Eräs grafiikan (ja geometrian) laji on ns. kilpikonnagrafiikka (kilpikonnageometria). Siinä tarkastellaan tasossa liikkuvaa subjektia, kilpikonaa, joka osaa: 1) liikkua askelittain eteen ja taakse, 2) paikalla ollessaan kääntyä asteittain () vasemmalle ja oikealle, sekä 3) nostaa ja laskea piirtovälineenä toimivan kynän. Jos kynä on alhaalla, jää piirtoalustaan konnan liikkueessa jälki, muutoin ei. Nimensä kilpikonnagrafiikka on saanut lattialla liikkuvasta robotista, jonka elektroniikkaa peittävä kuori sai sen muistuttamaan kilpikonaa. Kilpikonnageometriaa on käytetty sangen menestyksekkäästi algoritmisen ongelmanratkaisun opetuksessa kaiken ikäisten ihmisten kanssa etenkin Yhdysvalloissa, mutta myös Suomessa. Kilpikonna on suorittava subjekti, jonka ominaisuudet ja rajoitukset on helppo ymmärtää. Lisäksi tietokonegrafiikka keskittää opiskelijoiden huomion tehokkaasti sekä tarjoaa monen tasoisia opiskelijoita aidosti motivoivia, mutta silti kompakteja ongelmia.

Kilpikonnagrafiikka poikkeaa tavallisesta tasogeometriasta, jossa jokaisella pisteellä on sijainti xy-tasossa (x- ja y-koordinaatit). Kilpikonnagrafiikka on suhteellista siinä mielessä, että konnaa siirretään aina nykyisen sijainnin suhteen. Kilpikonnagrafiikalla voidaan harjoitella modulaarista ohjelmointia selvällä ja hausalla tavalla. Konnan ohjaamiseen käytettävissä oleva käskyjoukko on pieni ja selkeä, mutta siitä huolimatta saadaan helposti piirrettyjä monimutkaisia kuvioita. Tässä käytetyt komennot ovat *etene(x)*, *käännny(y)*, *laske kynä* ja *nosta kynä*. Lisäksi konnalla on aina suunta (sinne minne 'nokka' osoittaa). Komennolla *etene(x)*, kuljetaan x:n verran (eteenpäin, jos x>0 ja taaksepäin, jos x<0) nykyisestä sijainnista konnan osoittamaan suuntaan. Komennolla *käännny(y)* käännetään konnaa y asteen verran (myötäpäivään, jos y>0 ja vastapäivään, jos y<0) konnan nykyisestä suunnasta nähden. Ohjelmointikielet Logo ja Turbo Pascal sisältävät tällaisia (ja monia muitakin) komentoja konnan (kursorin) liikuttamiseen kuvaruudulla. On tietenkin selvää, että erilaiset kuvioiden piirtämiseen kirjoitetut moduulit ovat proseduureja, sillä konnan liikkuminen saadaan aikaan käskyllä, eikä tarkoituksena ole laskea mitään arvoa. Tällöin moduulin suoritus ei muuta muistipaikkojen arvoja, vaan saa aikaan (*sivu*)*vaikutuksen* kuvaruudulla.

Seuraavassa on moduuli, joka saa konnan piirtämään neliön, kun sivunpituus annetaan. Olkoon konnan kynän alkutilanteessa ylhäällä.

```
MODULE neliö(sivu)
  laske kynä
  REPEAT 4 TIMES
    etene(sivu)
    käännny(90)
  ENDREPEAT
  nosta kynä
ENDMODULE
```



Huomattakoon, että huolellisesti laaditun proseduurin ainoa vaikutus on piirtoalustaan ilmestyvä kuvio. Erityisesti proseduurilla ei ole mitään sivuvaikutuksia konnan *tilaan* (jonka määräävät konnan sijainti, suuntautuneisuus ja kynän asento): se on ns. *tilainvariantti*. Tämä on tärkeä ominaisuus, koska se tekee moduuleista helposti yhdisteltäviä. Moduulia voidaan huoletta kutsua missä tahansa tilanteessa ja jatkaa kutsun jälkeen kesken jääneen kuvion piirtämistä, koska konna on samassa tilassa kuin ennen moduulin kutsuakin. Kaiken lisäksi tilainvarianttien alimoduulien avulla konstruoidusta moduulista tulee itsekin automaattisesti tilainvariantti!

Kolmion piirto sujuu helposti samaan tapaan kuin neliönkin:

```
MODULE kolmio(sivu)
  laske kynä
  REPEAT 3 TIMES
    etene(sivu)
    käänny(120)
  ENDREPEAT
  nosta kynä
ENDMODULE
```

Moduulit ovat kovin samankaltaisia. Yleiskäyttöisempi ratkaisu saadaan, kun abstrahoidaan tehtävää edelleen. Huomataan, että mitä tahansa säännöllistä monikulmiota piirrettäessä kulmissa tehtävien käännösten summa on aina 360:

```
MODULE monikulmio(kulmienMäärä, sivu)
  laske kynä
  REPEAT kulmienMäärä TIMES
    etene(sivu)
    käänny(360 / kulmienMäärä )
  ENDREPEAT
  nosta kynä
ENDMODULE
```

Nyt on helppo määritellä:

```
MODULE neliö(sivu)
  monikulmio(4,sivu)
ENDMODULE

MODULE kolmio(sivu)
  monikulmio(3,sivu)
ENDMODULE
```

**Esimerkki.** Tarkastellaan seuraavaksi toista tasogeometriaan liittyvää esimerkkiä. Olkoon annettu kolme tason pistettä (koordinaattiparia). Onko niistä muodostuva kolmio suorakulmainen? Tehtävän vastaus on väitteen "muodostuva kolmio on suorakulmainen" totuusarvo: kyllä tai ei, tosi tai epätosi. Tämän arvon laskemista ei pidä sekoittaa arvon myöhempään hyväksikäyttämiseen, esimerkiksi sen tulostamiseen tai sen hyödyntämiseen muissa moduuleissa. Konstruoitavan moduulin tulee palauttaa arvo, totuusarvo, joten siitä on syytä tehdä funktio. Vastaus voidaan selvittää Pythagoraan lauseen avulla: suorakulmaisessa kolmiossa pisimmän sivun (hypotenuusan) pituuden neliö on yhtä suuri kuin muiden sivujen (kateettien) pituuksien neliöiden summa.

```
MODULE suorakulmainenko(a, b, c) RETURNS totuusarvo
  sivu1 := etäisyys(a,b)
  sivu2 := etäisyys(a,c)
  sivu3 := etäisyys(b,c)
  järjestä(sivu1, sivu2, sivu3)
  RETURN neliö(sivu1) + neliö(sivu2) = neliö(sivu3)
ENDMODULE
```

Tässä moduuli 'neliö' on tietenkin eri moduuli kuin edellä oleva neliön piirtävä moduuli ja pitäisi nimetä eriksi, jos näitä kahta käytettäisiin samassa ohjelmakokonaisuudessa. Moduulissa lasketaan ensin sivujen pituudet alimoduulin 'etäisyys' avulla. Sitten sivut järjestetään nousevaan suuruusjärjestykseen alimoduulin 'järjestä' avulla. Lopuksi palautetaan vastauksena väitteen 'pisimmän sivun pituuden neliö on yhtä kuin muiden sivujen pituuksien neliöiden summa' saama totuusarvo. Meidän pitää vielä kirjoittaa alimoduulit 'etäisyys', 'järjestä' ja 'neliö'. Moduuli etäisyys on muotoa:

```

MODULE etäisyys(pisteet (x1, y1) ja (x2, y2)) RETURNS pisteiden välinen etäisyys
  RETURN neliöjuuri(neliö(x1 - x2) + neliö(y1 - y2))
ENDMODULE

```

Pisteiden välisen etäisyyden laskemiseksi pisteiden nimet eivät riitä, vaan niiden rakenne täytyy tuntea tarkemmin. Siksi moduuli tarvitsee parametreikseen pisteiden x- ja y-koordinaattien arvot. Tietenkin kyseisten pisteiden koordinaatit tunnetaan tehtävän alusta alkaen (ne pitää tuntea, jotta tehtävän ratkaiseminen olisi mahdollista), mutta suorakulmainenko-moduulissa koordinaattien arvoja ei vielä tarvita, joten siinä riittää viitata pisteisiin niiden nimillä. Toki pisteiden sisäinen rakenne olisi voitu kirjoittaa näkyviin sielläkin, mutta se ei ole kyseisellä abstraktiotasolla välttämättömyys. Yksittäiset ohjelmointikielet määräävät menettelytavan tämäläpöä asioissa yksityiskohtaisesti.

Etäisyyden laskemiseksi hyödynnetään Pythagoraan lausetta toistamiseen. Alimoduulia 'neliö' tarvitaan myös nyt, joten kirjoitetaan se seuraavaksi. Toki neliöönkorotus-operaatio on useissa ohjelmointiympäristöissä jo valmiina, mutta se on hyvin helppo tehdä itsekin:

```

MODULE neliö(x) RETURNS x:n neliön
  RETURN x*x
ENDMODULE

```

Moduuli 'järjestä' eroaa edellä laadituista moduuleista sikäli, että sen tarkoituksena ei ole tuottaa uutta arvoa, vaan ainoastaan muuttaa olemassa olevien arvojen järjestystä. Niinpä on luonnollista ajatella moduulin merkityksen perustuvan uuden arvon tuottamisen asemesta parametreihin kohdistuvaan vaikutukseen. Tehdään moduulista siksi proseduri. Valinta toki voitaisiin tehdä toisinkin (tulkitsemalla lukujen muodostama kokonaisuus, jolla on tietty järjestys, omaksi koosteiseksi arvokseen), mutta imperatiivisessa ajattelussa proseduraalinen ratkaisu lienee luontevin. Moduulin sisäinen rakenne voitaisiin lainata aiemmin tarkastelemastamme valintarakente-esimerkistä, jossa pantiin kolme lukua suuruusjärjestykseen – onhan tässäkin tarkoitus järjestää nimenomaan kolme lukua. Laaditaan kuitenkin yleiskäyttöisempi lajittelualgoritmi, joka noudattaa ns. vaihtolajittelun ideaa (vrt. myös kappaleessa 2.5.4 esitetty kuplalajittelualgoritmi). Vaihtolajittelu, jolla voidaan lajitella minkä kokoinen vektori tahansa, esitetään kappaleessa 2.8.2.2. Sekä kupla- että vaihtolajittelu käyttää apunaan moduulia vaihda(x,y), joka vaihtaa muuttujien x ja y sisällöt keskenään. Esimerkiksi jos x=2 ja y=3, niin kutsun vaihda(x,y) jälkeen x=3 ja y=2.

```

MODULE järjestä(x1,x2,x3)
  IF x1>x2 THEN vaihda(x1,x2) ENDIF (* nyt x1<=x2 *)
  IF x1>x3 THEN vaihda(x1,x3) ENDIF (* nyt x1:ssä on pienin luku *)
  IF x2>x3 THEN vaihda(x2,x3) ENDIF (* nyt x1<=x2<=x3 *)
ENDMODULE

```

```

MODULE vaihda(x, y)
  apu := x
  x := y
  y := apu
ENDMODULE

```

Tässä on hyvä huomata, että kaikissa ohjelmointikielissä (esim. Javassa) ei ole mahdollista kirjoittaa em. moduuleja vaihda ja järjestä johtuen parametrien välitystavasta (kopiointi). Tällöin moduulien lauseet tulee kirjoittaa suoraan tarkasteltavaan kohtaan. Nyt tehtävän ratkaisu on valmis – vain neliöjuuren laskeva algoritmi puuttuu. Palataan siihen seuraavassa kappaleessa, jolloin teemme moduulin, joka palauttaa parametrinsa neliöjuuren. Ohjelmointikielissä on kuitenkin tällainen funktio yleensä valmiina.

## 2.6.5 Yhteenveto

Lopuksi yhteenveto modulaarisuuden eduista:

- Moduulit soveltuvat luonnollisella tavalla asteittain tarkentavaan menetelmään ja osittavaan suunnitteluun (top-down design).
- Moduuli on minkä tahansa sitä kutsuvan suuremman algoritmin itsenäinen komponentti. Komponenttien suunnittelun erottaminen toisistaan yksinkertaistaa ja nopeuttaa suunnitteluprosessia ja lisää algoritmien luotettavuutta.

- Moduulit selkeyttävät algoritmeja ja helpottavat niiden ymmärtämistä. Lisäksi erityisesti muiden kuin algoritmin alkuperäisen kirjoittajan on helpompi muuttaa algoritmia myöhemmin. Modulaarisuus helpottaa myös virheiden etsintää ja oikeellisuuden toteamista, koska algoritmia voidaan tarkastella paloittain.
- Liitettäessä valmis moduuli uuden algoritmin osaksi on välttämätöntä tietää ainoastaan, *mitä* moduuli tekee, ei sitä, *miten* se sen tekee. Näin ollen moduulin käyttäjän tarvitse tutustua ainoastaan moduulin ulkoiseen kuvaukseen.
- Kun moduuli on kerran suunniteltu, sitä voidaan käyttää hyväksi missä tahansa algoritmissa, jossa vastaava (osa)tehtävä esiintyy. Yleiskäyttöiset moduulit voidaan koota erityiseksi moduulikirjastoksi. Hyvin organisoidut moduulikirjastot tehostavat suuresti samantyyppisten algoritmien laatimista vastaisuudessa. Erityisesti olio-ohjelmointikielet (esim. Java ja C++) sisältävät erittäin laajat kirjastot, jotka ovat kaikkien ohjelmoijien käytössä.

## 2.7 Rekursio ja iteraatio

Jotkin ongelmat ovat niin helppoja, että ne voidaan ratkaista triviaalisti, esimerkiksi kirjoittamalla proseduri, jonka runkona on muutaman rivin mittainen peräkkäisrakenne tai funktio, jossa yksinkertaisesti luetellaan vaihtoehdot palautusarvot eri tilanteissa. Kaikki ongelmat eivät kuitenkaan ratkea suoraan, vaan usein tarvitaan jokin varsinainen menettelytapa ongelman ratkaisemiseksi. Algoritmisen ongelmanratkaisun päästrategiat ovat *rekursio* (recursion) ja *iteraatio* (iteration). Muodollisesti voidaan määritellä, että rekursiivinen moduuli on sellainen, joka sisältää itsensä kutsun (toki alkuperäisistä poikkeavin parametrein), ja iteratiivinen moduuli puolestaan sellainen, joka sisältää (päättävän) toistorakenteen. Tämä on kuitenkin vain toinen, pinnallisempi puoli asiasta. Pohjimmiltaan rekursio ja iteraatio ovat tapoja etsiä ongelman ratkaisua. Tässä kappaleessa tarkastellaan rekursiivisen ja iteratiivisen ongelmaratkaisun edellytyksiä ja ominaisuuksia, eroja ja yhtäläisyyksiä. Iteraatiosta on ollut jo useita esimerkkejä, joten aloitamme rekursiosta. Tutustutaan kuitenkin näihin strategioihin ensin käytännön elämään liittyvän esimerkin avulla.

**Esimerkki.** ”Iteraatio ja rekursio ovat kuin saman kolikon kaksi puolta”: Aidan maalaus.

```

MODULE maalaa(aita)
    WHILE aita maalaamatta DO
        suorita laudan maalaus
        siirry seuraavaan lautaan
    ENDWHILE
ENDMODULE

MODULE maalaa(aita)
    IF aita maalaamatta THEN
        suorita laudan maalaus
        maalaa(lopun aidasta)
    ENDIF
ENDMODULE

```

## 2.7.1 Rekursio

Modulaarisuusperiaatteen mukaan moduuli koostuu moduuleista. Moduulissa esiintyvän osatehtävän ratkaisemiseen voidaan käyttää mitä tahansa kyseisen tehtävän ratkaisevaa alimoduulia. Erityisesti jos tehtävä sisältää alkuperäisen ongelman kaltaisen (mutta pienemmän) osaongelman, voidaan moduulia itseään käyttää ratkaisun osana. Tällöin moduulin määrittelyn tulee sisältää moduulin itsensä kutsu, joten moduulista tulee rekursiivinen. Rekursio on siis modulaarisuusperiaatteen suora seuraus, eikä suinkaan mikään erillinen ominaisuus tai sallittu temppu, vaikka se joissakin yhteyksissä näkyy sellaisena esitettävänkin. Matalan tason kielissä, kuten konekielissä, rekursio ei kuitenkaan yleensä ole sallittua. Myöskään jotkut lausekielet, kuten Basicin, Fortranin tai Cobolin eräät versiot, eivät kielen toteutukseen liittyvien rajoitusten vuoksi salli rekursiota. Tällaisia kieliä ei tietenkään voi pitää aidosti modulaarisinakaan.

Rekursio on yleisen ongelmanratkaisuperiaatteen, reduktioperiaatteen erikoistapaus. Reduktiivisessa eli osittavassa ongelmanratkaisussahan ongelma ratkaistaan jakamalla se osiin, ratkaisemalla osaongelmat ja yhdistämällä osaongelmien ratkaisut koko ongelman ratkaisuksi. Reduktiota sanotaan rekursioksi, jos ainakin yksi osaongelma on alkuperäisen ongelman kaltainen. Jotta menettely suppenisi eli johtaisi todella ratkaisuun, tulee rekursiivisen aliongelman luonnollisestikin olla ongelman alkuperäistä tapausta yksinkertaisempi.

Rekursiivisten moduulien konstruoimisessa on huomattava kaksi seikkaa:

- moduulissa pitää kuvata ainakin yksi ongelman triviaali tapaus, joka ratkeaa suoraan, ei-rekursiivisesti, ja
- jokaisen rekursiivisen kutsun tulee lähestyä jotakin tällaista triviaalia tapausta

Näistä ensimmäistä sanotaan usein *rekursion kannaksi* ja jälkimmäistä *rekursio-askeleeksi*, jossa lähestytään rekursion kantaa. Vaatimuksista seuraa, että rekursiivisen kutsun tulee olla ehdollinen. Rekursiivisessa moduulissa ongelman ei-triviaalit tapaukset redusoidaan alkuperäisen ongelman kaltaisiksi, mutta yksinkertaisemmiksi ongelman tapauksiksi, ja triviaalit tapaukset ratkaistaan suoraan.

Jos moduuli kutsuu itseään, sanotaan rekursiota *suoraksi*. Suoran rekursion lisäksi esiintyy myös *epäsuoraa* rekursiota: vaikka mikään ongelman osista ei olisikaan alkuperäisen ongelman kaltainen, voi ositusta jatkettaessa osoittautua, että jokin osaongelmista sisältää edelleen alkuperäisen ongelman kaltaisen osaongelman. Algoritmin kuvauksen tasolla tämä tarkoittaa sitä, että vaikka moduuli M1 ei suoraan kutsuisikaan itseään, se voi kutsua moduulia M2, joka kutsuu edelleen moduulia M3 jne., kunnes vihdoin moduuli Mn kutsuu moduulia M1. Epäsuora rekursio ei luonnollisestikaan ole hyväksi algoritmin selkeydelle, eikä sitä sen vuoksi tulisi käyttää ilman erityisen hyvää syytä.

**Esimerkki.** Kokonaisluvun kertoman määrääminen rekursiivisesti. Kertoma määritellään usein rekursiivisen palautuskaavan avulla:  $0! = 1$  ja  $n! = n \cdot (n-1)!$ , jos  $n > 0$ . Siis:

```
(* Alkuehto: n >= 0 *)
MODULE kertoma(n) RETURNS n!
  IF n = 0 THEN
    RETURN 1                (* rekursion kanta *)
  ELSE
    RETURN n * kertoma(n-1) (* rekursioaskel *)
  ENDIF
ENDMODULE
```

Tarkastellaan esimerkiksi miten luvun 4 kertoma lasketaan:

```
kertoma(4):
IF 4 = 0 THEN
...
ELSE RETURN 4 * kertoma(3)
  kertoma(3):
  IF 3 = 0 THEN
...
  ELSE RETURN 3 * kertoma(2)
    kertoma(2):
    IF 2 = 0 THEN
...
    ELSE RETURN 2 * kertoma(1)
      kertoma(1):
      IF 1 = 0 THEN
...
      ELSE RETURN 1 * kertoma(0)
        kertoma(0):
        IF 0 = 0 THEN RETURN 1      (* 0! valmis *)
        RETURN 1 * 1 = 1           (* 1! valmis *)
      RETURN 2 * 1 = 2             (* 2! valmis *)
    RETURN 3 * 2 = 6              (* 3! valmis *)
  RETURN 4 * 6 = 24              (* 4! valmis *)
```

Huomaa erityisesti, että jokainen rekursiivisen kutsun suoritus päättyy vasta, kun kaikki sen generoimat kutsut ovat päättyneet. Esimerkiksi  $\text{kertoma}(4):n$  lasku sisältää aidosti  $\text{kertoma}(3):n$  laskun jne. Tämä tarkoittaa sitä, että  $\text{kertoma}(4):n$  lasku ainoastaan keskeytyy  $\text{kertoma}(3):n$  laskun ajaksi. Kun  $\text{kertoma}(3)$  on laskettu, jatketaan  $\text{kertoma}(4):n$  kesken jäänyttä laskua kertolaskulla  $4 \cdot 6 = 24$ . Vasta nyt kutsu  $\text{kertoma}(4)$  saa arvonsa. Jokainen RETURN-lause antaa siis arvon juuri sille laskulle, jonka osana RETURN-lause suoritetaan, ei suinkaan alkuperäiselle kutsulle.

Modulaarisuusperiaatteen mukaan moduulin muuttujat pysyvät erillään muiden moduulien muuttujista. Myös muuttujien nimet voidaan valita muista moduuleista riippumatta. Niinpä rekursiivisen moduulin jokaisella kutsullakin on omat (joskin samannimiset) muuttujansa, joiden arvot pidetään laskun edetessä erillään toisistaan. Tarkastellaan kertoma-moduulia ja merkitään kutsun  $\text{kertoma}(i)$   $n$ -nimistä parametria  $n_i$ :lla. Esimerkiksi  $n_2$  syntyy  $\text{kertoma}(2):n$  laskun alkaessa ja se saa arvonsa kutsusta ( $n_2=2$ ).  $\text{kertoma}(2):n$  lasku keskeytyy ja  $n_2$  passivoituu kutsun  $\text{kertoma}(1)$  suorituksen ajaksi, jolloin puolestaan parametri  $n_1$  on käytössä.  $\text{kertoma}(1):n$  laskun loputtua muistipaikka  $n_1$  häviää muistista, ja  $\text{kertoma}(2):n$  lasku jatkuu operaatiolla  $n_2 \cdot 1$ , missä 1 on kutsun  $\text{kertoma}(1)$  saama arvo. Operaation tulos 2 palautuu kutsun  $\text{kertoma}(2)$  arvoksi ja  $n_2$  häviää. Kullakin ajanhetkellä on siis vain yksi aktiivinen parametrin n esiintymä, johon kyseiset muuttujaviittaukset kohdistuvat.

**Esimerkki.** Myös  $\text{synt}(x,y)$ , missä kokonaisluvut  $x$  ja  $y > 0$ , voidaan määrätä rekursiivisella palautuskaavalla. Nyt täytyy vain erottaa kolme tapausta, joissa kussakin palautetaan eri arvo. Tapaus  $x=y$  on triviaali, mutta muut ovat rekursiivisia. Kummassakin rekursiivisessa kutsussa parametrit kuvaavat ongelman alkuperäistä tapausta yksinkertaisemmän tapauksen, koska jokaisessa kutsussa joko  $x:n$  tai  $y:n$  arvo pienenee, ja arvot yhtyvät viimeistään ykkösessä. Kun arvot yhtyvät, rekursio päättyy ja kutsupino purkautuu. Moduulin kolme lausetta voitaisiin kirjoittaa mihin tahansa järjestykseen, koska jokainen haara on ehdollinen ja toisensa pois sulkeva.

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

Esimerkiksi lasku `syt(24, 15)` etenee seuraavasti:

```
syt(24,15)
  = syt(9,15)
    = syt(9,6)
      = syt(3,6)
        = syt(3,3)
          = 3
            = 3
              = 3
                = 3
```

**Esimerkki.** Annetun sanan kääntäminen takaperin. Ongelman ratkaisu voisi olla seuraavanlainen:

```
Ota erilleen sanan ensimmäinen kirjain
Käännä sanan loppuosa takaperin
Lisää ensimmäinen kirjain käännetyin osan loppuun
```

Algoritmi on selvästi rekursiivinen. Mutta triviaali tapaus puuttuu. Sellaiseksi voidaan valita sana, jossa on korkeintaan yksi kirjain – sehän on käännettynä sama kuin kääntämättäkin. Jokaisessa kutsussa käännettävä sana lyhenee yhdellä, kun sanan ensimmäinen kirjain poistetaan. Kutsu siis todella yksinkertaistaa ongelmaa, koska ongelman tapaus lähenee triviaalia tapausta. Rekursio päättyy, kun triviaali tapaus saavutetaan. Tehtävässä käsitellään sanoja, joka ovat kirjaimista koostuvia rakenteita. Koska tietorakenteita tarkastellaan monisteessa vasta tuonnempana, tehdään moduulista proseduurin, joka tulostaa sanan kirjaimet takaperin yksi kerrallaan.

```
MODULE takaperin (sana)
  eka := sanan ensimmäinen kirjain
  IF sanan pituus > 1 THEN takaperin(sanan loppuosa) ENDIF
  tulosta(eka)
ENDMODULE
```

Ratkaisussa on se vika, että fraasin 'sanon ensimmäinen kirjain' korrekti käyttö vaatii, että sanan on oltava vähintään yhden kirjaimen mittainen. Ratkaisu ei siis sovellu tyhjielle sanoille. Tämä formaalinen ongelma poistuu helposti, kun triviaaliksi tapaukseksi valitaan tyhjä sana, jonka kirjainten tulostamiseksi (missä tahansa järjestyksessä) ei tarvitse tehdä mitään.

```
MODULE takaperin (sana)
  IF sana ei ole tyhjä THEN
    eka := sanan ensimmäinen kirjain
    takaperin(sanan loppuosa)
    tulosta(eka)
  ENDIF
ENDMODULE
```

## 2.7.2 Iteraatio

Imperatiivisessa ajattelussa iteraatio on varsin tavallinen menettelytapa, nojaahan koko imperatiivinen lähestymistapa suoraan tietokoneen työskentelytapaan, joka on luonnostaan iteratiivinen.

Iteraatioperiaate tarkoittaa sitä, että jotakin toimenpidettä toistamalla päästään yhä lähemmäksi ratkaisua. Tarkoitus on tuottaa yhä parempia ja parempia välitiloja,

likiarvoja annetun ongelman ratkaisulle. Prosessi päättyy, kun saadaan joko tarkka ratkaisu tai riittävän hyvä likiarvo. Iteraation soveltamiselle asetetaan kolme vaatimusta:

- väliarvojen olemassaolo
- keino edetä kohti ratkaisua
- kyky tunnistaa ratkaisu

Esimerkiksi aiemmin esitettyssä iteratiivisessa lajittelualgoritmissa (vaihtolajittelu) näiden vaatimusten täyttyminen voidaan todeta seuraavasti: Vaadittavat väliarvot ovat lukujonon eri järjestykset. Aluksi lukujono on järjestyksessä alusta lukien 0. alkioon saakka. Kullakin ulomman silmukan suorituskerralla päästään yhä lähemmäksi ratkaisua: jono on järjestyksessä alusta lukien  $i$ . alkioon saakka,  $i=1,2,\dots,n-1$ . Tarkka ratkaisu on saavutettu, kun  $i=n-1$ ; viimeinen eli  $n$ . alkio ei voi yksinään olla epäjärjestyksessä, koska kaikki muut alkioit ovat jo omalla paikallaan.

Erityisen luonteva iteratiivinen lähestymistapa on tehtävissä, joissa ei välttämättä saada tarkkaa ratkaisua, vaan on tyydyttävä likiarvoon. Silloin on tietysti määriteltävä miten hyvää likiarvoa pidetään riittävänä. Tällaisia *numeerisen analyysin* menetelmiä käytetään erityisesti sovelletussa matematiikassa.

**Esimerkki.** Neliöjuuren likiarvon laskeminen. Olkoon  $a$  jokin positiivinen reaalliluku. Iteratiivinen menetelmä reaali-luvun  $a$  neliöjuuren likiarvon laskemiseksi voidaan johtaa seuraavasti:

$$x = \sqrt{a}$$

$$\Rightarrow x^2 = a \Rightarrow 2x^2 = x^2 + a \Rightarrow x = \frac{1}{2} \left( x + \frac{a}{x} \right)$$

Jos  $x$ :lle on jo laskettu likiarvo  $x_1$ , saadaan viimeisellä yhtälöllä uusi, parempi likiarvo sijoittamalla vanha likiarvo yhtälön oikealle puolelle:

$$x_2 = \frac{1}{2} \left( x_1 + \frac{a}{x_1} \right)$$

Menetelmä suppenee niin nopeasti, että ensimmäinen likiarvo  $x_1$  voi olla pelkkä arvaus. Arvataan, että  $x_1 = 1$ . Voidaan osoittaa, että alla olevan rekursiivisen palautuskaavan avulla määritelty lukujono suppenee kohti  $a$ :n neliöjuurta:

$$x_{n+1} = \begin{cases} 1, & n = 0 \\ \frac{1}{2} \left( x_n + \frac{a}{x_n} \right), & n > 0 \end{cases}$$

Tarkastellaan esimerkkinä neliöjuuri kahden määrittämistä, jolloin  $a=2$ . Kun  $n=0$ , saadaan eo. palautuskaavaan (ylempi rivi) nojalla  $x_1=1$ . Sijoittamalla nyt  $n=1$  eo. palautuskaavaan (alempi rivi), saadaan

$$x_2 = 1/2(x_1 + 2/x_1) = 3/2 = 1.5,$$

ja samalla tavalla sijoittamalla tämä palautuskaavaan saadaan

$$x_3 = 1/2(x_2 + 2/x_2) = 1/2(3/2 + 4/3) = 17/12 \approx 1.42$$

jne. Siis uusi tarkempi likiarvo saadaan sijoittamalla vanha likiarvo palautuskaavan oikealle puolelle  $x_n$ :n paikalle. Nähdään, että laskutoimitukset käyvät hyvin monimutkaisiksi, mutta tietokone on omiaan laskemaan juuri tällaisia laskutoimituksia. Lisäksi havaitaan, että menetelmä suppenee erittäin nopeasti (eli ei tarvitse laskea montakaan lukujonon termiä, kun jo saadaan hyvä likiarvo neliöjuuri kahdelle).

Edellinen palautuskaava on suoraviivaista kirjoittaa algoritmiksi. Koska lukujono vähenee vasta  $n$ :n arvosta 2 lähtien, niin asetamme algoritmin lähtökohdaksi termin  $x_2$ , jonka arvo saadaan sijoittamalla  $x_1=1$  palautuskaavaan.

(\* Alkuehto:  $a > 0$  ja  $\varepsilon > 0$  \*)

```

MODULE neliöjuuri(a, ε) RETURNS √a :n likiarvo
  n := 2
  x2 := (1+a)/2
  REPEAT
    xn+1 :=  $\frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$ 
    n := n+1
  UNTIL xn-1 - xn < ε (* ei tarvita itseisarvoa, koska lukujono on vähenevä *)
  RETURN xn
ENDMODULE

```

Likiarvosta saadaan mielivaltaisen tarkka iterointia jatkamalla. Iterointi lopetetaan, kun virhe on riittävän pieni. Eräs yleinen tapa virheen arvioimiseksi on laskea kahden peräkkäisen termin ero. Algoritmista lopetetaan uusien termien laskeminen kun tämä ero on riittävän pieni, pienempi kuin sallittu virhe  $\varepsilon$ . Menetelmä on yksinkertaisuudestaan huolimatta niin tehokas, että kahdeksan numeron tarkkuus saavutetaan jo kolmen kierroksen jälkeen, jos  $a=4$ ; kuuden kierroksen jälkeen, jos  $a=100$ ; ja yhdeksän kierroksen jälkeen, jos  $a=10\,000$ .

Edellä oleva muuttujien indeksointi ei ole ohjelmointikielissä mahdollista, vaan tällöin väliarvot tulisi tallentaa taulukkoon x eli edellä tulisi kirjoittaa x[2] jne. Iteraation väliarvoja ei tietenkään tarvitse eikä kannata tallentaa, vaan kannattaa käyttää vain kahta muuttujaa: nykyinen ja edellinen, joissa ylläpidetään kahta viimeisintä likiarvoa. Tällöin likiarvon laskeva algoritmi saa näinkin yksinkertaisen muodon:

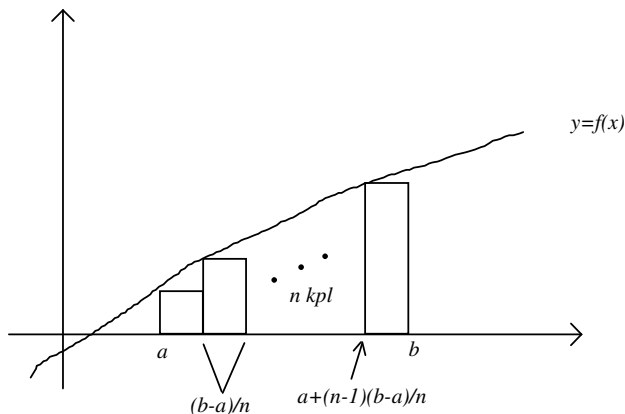
(\* Alkuehto:  $a > 0$  ja  $\varepsilon > 0$  \*)

```

MODULE neliöjuuri(a, ε) RETURNS √a :n likiarvo
  nykyinen := (1+a)/2
  REPEAT
    edellinen := nykyinen
    nykyinen :=  $\frac{1}{2} \left( edellinen + \frac{a}{edellinen} \right)$ 
  UNTIL edellinen - nykyinen < ε
  RETURN nykyinen
ENDMODULE

```

**Esimerkki.** Numeerinen integrointi. Iteratiivisia menetelmiä käytetään yleisesti myös numeeristen integraalien laskennassa, esimerkiksi pinta-alojen tai tilavuuksien määrittämiseksi. Lasketaan jatkuvan käyrän  $y=f(x)$  sekä suorien  $y=0$ ,  $x=a$  ja  $x=b$  rajoittaman alueen pinta-ala. Alaa voidaan approksimoida n:llä suorakulmiolla:



Pinta-ala saadaan kaavasta

$$I = \int_a^b f(x) dx \approx \left( \sum_{x=0}^{n-1} f \left( a + x \cdot \frac{b-a}{n} \right) \right) \cdot \frac{b-a}{n}$$

Likiarvo on sitä parempi, mitä suurempi n on. Kirjoitetaan moduuli, joka saa parametreinaan funktion f arvon laskevan funktiomoduulin f sekä luvut a ja b:



```
(* Alkuehto: a < b *)
MODULE integroi (f, a, b) RETURNS luku
  arvo := 0
  n := suuri luku (* esim. 1000 *)
  dx := (b-a)/n
  FOR x := 0,1,..., n-1 DO arvo := arvo + f(a+x*dx) ENDFOR
  RETURN arvo*dx
ENDMODULE
```

**Esimerkki.** Eratostheneen seula. Numeerisen analyysin lisäksi iteratiivisia menetelmiä käytetään myös muunlaisissa tehtävissä. Kreikkalainen matemaatikko ja tähtitieteilijä Eratosthenes (276–195 eKr.) esitti seuraavan, Eratostheneen seulana tunnetun menetelmän annettua lukua  $n$  pienempien alkulukujen määräämiseksi. Tämä erittäin näppärä algoritmi operoi positiivisten kokonaislukujen joukoilla. Huomaa kuitenkin, että useimmissa kielissä ei voi joukkoja käsitellä suoraan kuten alla on tehty, vaan ensin tulee kirjoittaa sopivat moduulit joukkojen ja joukko-operaatioiden toteutukseen (vrt. kappale 2.8).

```
MODULE seula(n) RETURNS alkulukujen joukko
  S := {2,3,4,5,6,...,n}
  A := ∅ (* tyhjä joukko *)
  REPEAT
    Etsi joukon S pienin luku b
    A := A ∪ {b} (* lisätään alkio b joukkoon A *)
    m := b
    REPEAT
      S := S - {m} (* poistetaan alkio m joukosta S *)
      m := m+b
    UNTIL m > n
  UNTIL b > neliöjuuri(n)
  RETURN A ∪ S
ENDMODULE
```

Algoritmi on paljon aikaisemmin esitetyn alkulukutestin varaan rakennettua algoritmia tehokkaampi, jos tarvitaan jonkin joukon kaikki alkuluvut. Se myös todella tuottaa kaikki etsityt alkuluvut.

### 2.7.3 Rekursio vai iteraatio?

Edellisistä esimerkeistä nähtiin miten toistorakenteeseen perustuvat algoritmit kertoman ja suurimman yhteisen tekijän määräämiseksi voitiin laatia myös rekursiivisesti. Itse ratkaisun idea oli sama, vain muoto oli erilainen. Muodollisesti rekursio ja iteraatio ovatkin useasti 'saman kolikon kaksi puolta', vaihtoehtoisia strategioita. Koska tietokone toimii iteratiivisesti, on selvää, että jokainen rekursiivinen algoritmi voidaan muuntaa iteratiiviseksi. Käytännössä rekursion käyttö saattaa olla muistia tuhlaavaa, mutta tämäkin on riippuvainen tietenkin ohjelmointiympäristöstä.

Rekursio perustuu modulaarisuuteen ja reduktiiviseen ajatteluun, joka on mitä luontevin yleinen ongelmanratkaisutapa ja vallitseva strategia deklarativisessa ohjelmoinnissa. Usein tehtävä ratkeaa helpommin rekursiivisesti kuin iteratiivisesti – ja joskus, esim. rekursiivisten tietorakenteiden yhteydessä – rekursio on ainoa luonnollinen ratkaisumetodi.

Jotkin tehtävät ovat sellaisia, että niille on kohtuullisen helposti löydettävissä joko rekursiivinen tai iteratiivinen ratkaisu, mutta ei molempia. On myös tehtäviä, joille ei ole lainkaan olemassa kunnollista ei-rekursiivista ratkaisua. Tällaisia tehtäviä sanotaan **luonnostaan rekursiiviksi** (inherently recursive). Ratkaisun 'kunnollisuus' tarkoittaa tässä sitä, että vaikka jokainen rekursiivinen moduuli voidaankin muuntaa muodollisesti iteratiiviseksi – ja konekielelle käännettäessä näin viimeistään aina tapahtuukin – ei

kaikkiin ongelmiin ole mahdollista löytää muunlaista iteratiivista ratkaisua kuin sellainen, joka 'matkii' alkuperäistä rekursiivista ratkaisua. Kysymys on myös ongelmanratkaisusta: yritetäänkö tehtävää ratkaista iteratiivisesti vai rekursiivisesti. Esimerkiksi luvussa 3.2.3 esitetään ns. Hanoin tornien ongelma (tutustu siihen jo nyt!), joka ratkeaa luonnollisesti rekursiivisesti. Samoin rekursiivisten tietorakenteiden käsittely sujuu helpoiten käyttäen rekursiota (ks. esim. luku 2.8.3.4).

Peruslähtökohdiltaan rekursio ja iteraatio eroavatkin toisistaan merkittävästi.

Yhteenvedona rekursiosta ja iteraatiosta voidaan todeta seuraavaa:

- rekursio ja iteraatio ovat yleensä vaihtoehtoisia lähestymistapoja ja ratkaisut voidaan muuntaa toisikseen
- rekursio on yleisen reduktioperiaatteen erikoistapaus ja kuuluu modulaarisuus-ajatteluun
- iteraatio puolestaan perustuu ratkaisun asteittaiseen lähestymiseen ja kuuluu imperatiiviseen ohjelmointiajatteluun
- rekursio on luonnollinen ratkaisustrategia funktionaalisessa ja logiikkaohjelmoinnissa.

## 2.8 Tieto- ja tallennusrakenteet

### 2.8.1 Tyyppi, abstrakti tietotyyppi ja sen implementointi

Tähän saakka on tarkasteltu pääasiassa algoritmien ohjausrakenteita, joilla ohjataan algoritmin askelten suoritusjärjestystä ja jätetty vähemmälle huomiolle algoritmien käsittelemä tieto (data). Käsitelty tieto on ollut yksinkertaista, jolla ei ole ollut varsinaisesti sisäistä rakennetta (esim. luku, merkki, totuusarvo). Yleensä algoritmin käsittelemä tieto ei ole irrallista ja mielivaltaista, vaan se muodostuu useista tiedoista, jotka liittyvät jotenkin toisiinsa. Kuvataanhan algoritmeilla jotain reaali maailman ongelman ratkaisua, jolloin tilannetta mallinnetaan joukolla loogisesti yhteenkuuluvia tietoja. Nämä tiedot halutaan usein koota yhteen ja esittää ne yhtenä *tietorakenteena* (data structure).

Imperatiiviset ohjelmointikieliet ovat yleensä tyyppitettyjä, mikä tarkoittaa sitä, että jokaisella ohjelman käsittelemällä tietoalkiolla (muuttuja tai lauseke) on *tyyppi*, joka kertoo millaisia arvoja tarkasteltava alkio voi saada ja millaisia operaatiota alkioon voidaan soveltaa. Vahvasti tyyppitetyissä kielissä (esim. Java, C++ ja Pascal) ohjelmoijan tulee aina esitellä muuttuja ja ilmoittaa sen tyyppi. Esim. Javassa voidaan määrittellä `int x`, joka tarkoittaa sitä, että `x` on muuttuja, johon voidaan tallentaa vain kokonaislukuarvoja. Sen sijaan esim. Python-kieli on dynaamisesti tyyppittävä kieli, jolloin muuttujaa ei tarvitse esitellä erikseen eikä sen tyyppiä tarvitse ilmoittaa, vaan tyyppi määräytyy automaattisesti riippuen sen käyttötavasta; esim. jos muuttujan arvoksi annetaan `2`, niin muuttujan tyyppiä tulee automaattisesti kokonaisluku (`int`). Näin voidaan ajatella asian olevan myös tämän monisteen pseudokielessä. Kielissä on valmiina nimetyt tyypit yksinkertaiselle tiedolle ja välineet erilaisten tietorakenteiden konstruomiseksi. Yksinkertaisiksi tyypeiksi käsitetään yleensä kokonaisluvut (tyypin nimi Javassa: `int`), reaaliluvut eli desimaaliluvut (`double`), totuusarvot `true` ja `false` (boolean) ja merkit (`char`).

Tyyppin voidaan siis ymmärtää muodostuvan kahdesta osasta:

- kaikista mahdollisista arvoista
- niistä operaatioista, joita kyseisen tyyppisiin alkioihin voidaan kohdistaa.

Esimerkiksi tyyppi `int` tulee täysin määriteltyä, kun kerrotaan mikä on suurin ja pienin kokonaisluku ja mitä operaatiota kokonaislukuihin voidaan kohdistaa (esim. `+`, `-`, `*`, `/`, jakojäännös) ja miten operaatiot toimivat. Kielen perustyypeille nämä operaatiot ovat 'sisäänrakennettuja', kuten esim. Javan tyyppin `int` tapauksessa, ja niiden toiminta vastaa oletettua toimintaa. Mutta jos rakennetaan omia tietorakenteita, tulee operaatiot ja niiden toiminta implementoida itse. Tietorakenteiden konstruoimiseksi on usein valmiina taulukot (array), tietueet (record) tai luokat (class). Tietorakenteita voidaan rakentaa käyttäen myös dynaamisia linkitettyjä rakenteita.

Puhuttaessa tietorakenteista tulee erottaa tietorakenteen abstrakti malli ja sen mahdolliset *tallennusrakenteet* (storage structure). Tallennusrakenne kertoo miten tietorakenne on toteutettu ja tallennettu tietokoneen muistiin. Abstraktissa mallissa kuvataan vain tietorakenteeseen kohdistettavat operaatiot ja niiden toiminta, jotka siis määrittelevät mallin mukaisten objektien käytöksen. Tällaista mallia kutsutaan *abstraktiksi tietotyyppiksi* (abstract data type, ADT). Termi abstrakti tarkoittaa tässä sitä, että 1) tarkastellaan mallin mukaisten alkioiden oleellisia piirteitä 2) malli on riippumaton käytettävästä ohjelmointikielystä 3) alkiot kuvataan niiden käytöksen avulla (so. operaatioiden ja niiden vaikutuksen avulla) eikä sen mukaan miltä ne näyttävät (mikä on niiden tallennusrakenne). Tietoalkiota mallinnetaan siis kaikista ohjelmointikielistä ja tallennusrakenteista riippumattomilla abstrakteilla tietotyypeillä.

Seuraavassa esitetään abstraktit tietotyypit lista, jono, pino, puu ja graafi. Sitä ennen esitetään kuitenkin konkreettiset työkalut, joilla nämä abstraktit tietorakenteet ja niiden toiminta voidaan toteuttaa. Näitä työkaluja ovat tietue, taulukko sekä erilaiset linkitetty rakenteet, jotka ovat käytettävissä useimmissa ohjelmointikielissä. Abstrakteihin tietotyyppeihin palataan vielä kappaleessa 2.8.5, jossa puhutaan lyhyesti oliokeskeisen ohjelmoinnin luokista, joiden avulla kaikki tarvittavat abstraktit tietotyypit toteutetaan.

Myös olio-ohjelmoinnin käsite *luokka* on tallennusrakenne, mutta se on paljon enemmän: luokalla implementoidaan koko ADT sisällyttämällä luokkaan käsiteltävä data sekä myös dataa käsittelevät metodit (algoritmimoduulit). Sen vuoksi seuraavassa luokan käsite tarkastellaan lopuksi omana kappaleenaan Oliokeskeinen ohjelmointi (Olio-ohjelmointi).

## 2.8.2 Tallennusrakenteet

### 2.8.2.1 Tietue

Tietuerakenteella (esim. RECORD-rakenne Pascalissa ja struct-rakenne C:ssä) voidaan koota loogisesti yhteenkuuluvat, usein erityyppiset tiedot yhdeksi tietorakenteeksi. Tietueen komponentteja (rakenneosia) kutsutaan tietueen *kentiksi*, jotka voivat puolestaan olla yleensä mitä rakenteita tahansa (esim. tietueita). Tietueen käsite kuuluu useimpiin ohjelmointikieliin, joskin eri kielissä tietueet ovat jonkin verran erilaisia. Esimerkiksi oliokielistä käytetään tietueen sijasta luokkaa, joka on tietuekäsitteen

laajennus: tietojen lisäksi yhteen sidotaan tietoja käsittelevät operaatiot eli metodit. Oliokeskeiseen ohjelmointiin ja luokkakäsitteeseen palataan kappaleessa 2.8.4.

**Esimerkki.** Henkilötiedot on luonnollinen tietuerakenne, joka voidaan koostaa halutuista tiedoista (esim. nimi, osoite, syntymäaika, ...). Esimerkiksi Pascalissa määrittely voidaan tehdä seuraavasti:

```
henkilotiedot=RECORD
    nimi: String;
    osoite: String;
    (* ... muita tietoja tarpeen mukaan *)
END;
```

Tietuetta käsitellään kentittäin ja tietueen kenttiin viitataan (usein) ns. **pistenotaatiolla**:

*muuttujan nimi.kentän nimi,*

missä muuttujan nimi on tietuetyyppinen muuttuja.

Tietueen kenttiä voidaan käyttää samalla tavalla kuin samantyyppisiä muuttujia. Tietueelle sallittavat operaatiot ovat:

- arvon sijoittaminen tietueen kenttään ja
- tietueen yksittäisen kentän arvon käyttäminen.

### 2.8.2.2 Taulukko

**Taulukko** (Array) on staattinen, kiinteän kokoinen rakenne, mutta taulukon koon voi kuitenkin määrittellä vasta ajon aikana. Tämän jälkeen taulukon kokoa ei saa enää muuttaa, mutta sen sisältöä eli komponenttien arvoja voidaan muuttaa. Taulukko koostuu useista samantyyppisistä tietoalkioista, jotka identifioidaan sijainnin mukaan indeksoimalla taulukon alkiot tarkoituksenmukaisella tavalla (alkiot indeksoidaan järjestysnumerolla 1,2,... tai 0,1, .. kuten esim. Javassa). Taulukon alkiot sijaitsevat muistissa fyysisesti peräkkäin, joten niiden käsittely on tehokasta ja helppoa. Tämän vuoksi taulukko on erittäin hyvä tallennusrakenne, jos käsiteltävien alkioiden lukumäärä on kiinteä. Taulukko voi olla yksi- tai useampiulotteinen. Yksiulotteinen taulukko indeksoidaan yhdellä indeksillä ja rakennetta nimitetään **vektoriksi** (vector). Kaksiulotteinen taulukko indeksoidaan kahdella indeksillä (rivi- ja sarakeindeksi) ja rakennetta kutsutaan **matriisiksi** (matrix). Ohjelmointikielissä taulukko voi olla useampiulotteinenkin, mutta seuraavassa käsitellään vain vektoreita ja matriiseja.

Olkoon  $V$  1-ulotteinen taulukko eli vektori, jossa on  $k$  alkioita eli jonka pituus on  $k$ . Olkoon lisäksi  $M$  2-ulotteinen taulukko eli matriisi, jonka ensimmäisessä ulottuvuudessa (rivit) on  $n$  alkioita ja toisessa (sarakkeet)  $m$  alkioita. Matriisia  $M$  sanotaan tällöin  $n \times m$ -taulukoksi. Taulukon yksittäisiin alkioihin (komponentteihin) viitataan notaatioilla  $V[i]$ , missä  $1 \leq i \leq k$ , ja  $M[i,j]$ , missä  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ . Taulukon komponentteja voidaan käyttää samalla tavalla kuin samantyyppisiä muuttujia. Taulukolle sallitut operaatiot ovat:

- tietoalkion tallennus taulukkoon indeksien osoittamaan kohtaan (esimerkiksi  $V[i]:=11$ ,  $M[2,3] := x$ ),
- taulukon yksittäisen tietoalkion käyttö (esim.  $y:=V[i]$ ,  $y := M[2,3]$  )

**Esimerkki.** Seuraavassa on moduuli, joka palauttaa parametrina annetun vektorin alkioiden keskiarvon. Useissa ohjelmointikielissä olemassa olevan vektorin T alkioiden lukumäärän saa selville lausekkeella T.length jota allakin käytetään.

```
(* Alkuehto: T.length>0 *)
MODULE keskiarvo(T) RETURNS T:n komponenttien keskiarvon
  s := 0
  k := 1
  WHILE k <= T.length DO
    s := s + T[ k ]
    k := k + 1
  ENDWHILE
  RETURN s / T.length
ENDMODULE
```

**Esimerkki.** Matriiseissa rivi-indeksi kirjoitetaan yleensä ennen sarake-indeksiä. Esimerkiksi juoksevasti ykkösestä alkaen indeksoitu 4x6-matriisi T voidaan kirjoittaa taulukkomuodossa seuraavasti:

M[1,1]	M[1,2]	M[1,3]	M[1,4]	M[1,5]	M[1,6]
M[2,1]	M[2,2]	M[2,3]	M[2,4]	M[2,5]	M[2,6]
M[3,1]	M[3,2]	M[3,3]	M[3,4]	M[3,5]	M[3,6]
M[4,1]	M[4,2]	M[4,3]	M[4,4]	M[4,5]	M[4,6]

**Esimerkki.** Vaihtolajittelu. Lista on tallennettu yksilotteiseen taulukkoon (vektoriin) T, joka sisältää esim. lukuja. Vaihtolajittelu perustuu kahden sisäkkäisen FOR-silmukan käyttöön. Ensin ensimmäistä alkioita T[1] (kun i=1) verrataan kaikkiin muihin alkioihin T[j] (j saa arvot 2:sta n:ään) ja alkioiden sisällöt vaihdetaan, jos löydetään väärä järjestys. Tämän jälkeen vektorin ensimmäisessä alkiossa (=T[1]) on vektorin pienin alkio. Sen jälkeen verrataan toista alkioita T[2] (kun i=2) verrataan sen perässä oleviin alkioihin T[j] (j=3,...,n) ja alkioiden sisällöt vaihdetaan, jos löydetään väärä järjestys. Tämän vaiheen jälkeen vektorin T toisessa alkiossa (=T[2]) on vektorin toiseksi pienin alkio. Kyseistä menettelyä jatketaan arvolla i=3, jolloin j saa arvot 4,...,n jne.

```
MODULE vaihtolajittelu(T)
  FOR i := 1, 2, ..., T.length-1 DO
    FOR k:= i+1, ..., T.length DO
      IF T[i] > T[k] THEN vaihda(T[i], T[k]) ENDIF
    ENDFOR
  ENDFOR
ENDMODULE
```

**Esimerkki.** Kahden kaupungin välisen etäisyyden määrittäminen. Kaupunkien väliset etäisyydet on tallennettu kaksilotteiseen taulukkoon Etä. Etäisyystaulukko voisi olla esimerkiksi seuraavanlainen 6x6-matriisi:

	Iisalmi	Jyväskylä	Kuopio	Pieksämäki	Suonenjoki	Varkaus
Iisalmi	0	254	85	174	136	223
Jyväskylä	254	0	169	80	118	129
Kuopio	85	169	0	89	51	138
Pieksämäki	174	80	89	0	38	49
Suonenjoki	136	118	51	38	0	87
Varkaus	223	129	138	49	87	0

Kirjoitetaan moduuli, joka määrittää kaupunkia a lähinnä olevan kaupungin:

```

MODULE LähinKaupunki(Etä, a) RETURNS a:ta lähinnä olevan kaupungin nimen
  lähinmatka := 100000 (* jokin 'iso' luku alkuarvoksi *)
  FOR j := Iisalmi, ..., Varkaus DO
    IF a <> j THEN (* ei käsitellä etäisyyttä itseensä *)
      IF Etä[a,j] < lähinmatka THEN
        lähinmatka := Etä[a,j]
        lähin := j
      ENDIF
    ENDIF
  ENDFOR
  RETURN lähin
ENDMODULE

```

Esimerkiksi LähinKaupunki(Etä, Jyväskylä) = Pieksämäki. Muuttujan lähinmatka alkuarvoksi olisi parempi antaa joku matriisin Etä rivillä a oleva luku eikä tällaista keksittyä suurta lukua. Useissa ohjelmointikielissä taulukkoa ei voida indeksoida kaupunkien nimillä, joten tällöin joudutaan kaupungit koodaamaan esimerkiksi kokonaisluvuilla: 1=Iisalmi, 2=Jyväskylä, jne.

**Esimerkki.**  $n \times m$ -lukutaulukon Taulu kaikkien alkioiden summan laskeminen.

```

MODULE taulusumma(Taulu, n, m) RETURNS lukujen summa
  summa := 0
  FOR i := 1, ..., n DO
    FOR k := 1, ..., m DO
      summa := summa + Taulu[i, k]
    ENDFOR
  ENDFOR
  RETURN summa
ENDMODULE

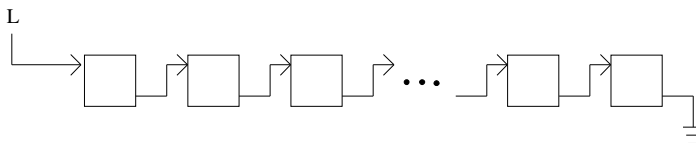
```

### 2.8.2.3 Linkitetty rakenne

Linkitettyssä rakenteessa jokainen tietoalkio sisältää varsinaisen datan lisäksi tiedon seuraajastaan, ns. osoittimen rakenteen seuraavaan alkioon. Tällöin rakenteen yksittäinen alkio on tietue, jossa on yksi tai useampia datakenttiä ja yksi (kuten ao. kuvassa) tai useampia osoittimia rakenteen muihin alkioihin:

data 1
...
data n
osoitin seuraajaan

Usein koko rakenne samastetaan ensimmäiseen alkioon osoittavan osoittimen kanssa. Linkitetty *yhteen suuntaan ketjutettu* (lineaarinen) rakenne L näyttää seuraavalta:

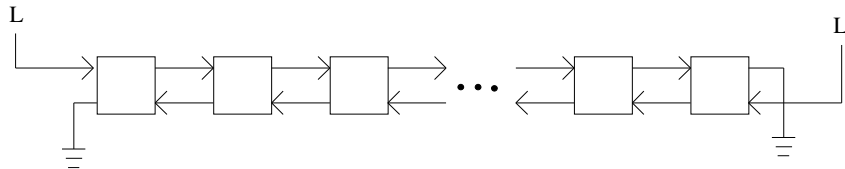


Linkitetty rakenne on rekursiivinen: jokaista alkioita seuraa rakenne, joka on alkuperäisen rakenteen kaltainen, joskin pienempi. Poikkeuksen tekee rakenteen viimeinen alkio, jolla ei ole seuraajaa. Voidaan myös sanoa, että viimeisen alkion seuraaja on tyhjä. Tätä

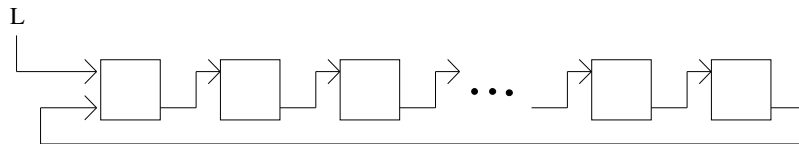
merkitään 'maadoittamalla' viimeisen alkion seuraaja-osoitin kuten yllä olevasta kuvasta näkyy.

Linkitetystä rakenteesta esiintyy useita muunnelmia käyttötarpeen mukaan. Esimerkiksi jos halutaan kulkea rakenteesta sekä eteen- että taaksepäin, kannattaa rakenne ketjuttaa kumpaankin suuntaan. Yleisimpiä ovat seuraavat variaatiot:

- **Kahteen suuntaan ketjutettu rakenne** on nimensä mukaisesti rakenne, jonka alkioilla on seuraajan lisäksi myös eksplisiittisesti osoitettu edeltäjä. Rakenteen etuna on se, että käsittelyn ei tarvitse edetä alusta loppuun, vaan voidaan palata myös taaksepäin. Haittana on tietysti osoittimien ylläpidon työlästyminen.



- **Rengasrakenne** on tavallinen linkitetty rakenne, jossa ensimmäinen alkio on asetettu viimeisen alkion seuraajaksi. Rengasmäinen rakenne mahdollistaa käsittelemisen sykklisesti.



- **Useita seuraajia salliva rakenne** on sellainen linkitetty rakenne, jossa alkiolla voi olla enemmän kuin yksi seuraaja.

### 2.8.3 Abstraktit tietotyypit

Edellä käsiteltiin konkreettisia tietojen tallennusrakenteita. Tietorakenteen abstraktissa mallissa kuvataan vain ne ominaisuudet ja operaatiot (miten rakennetta käsitellään) sekä niiden toiminta (esimerkiksi seuraavassa esitettävään tietorakenteeseen jono voidaan kohdistaa vain poisto- ja lisäysoperaatio, jotka toimivat ns. FIFO-periaatteella), joiden avulla tietorakenne tulee ulkoisesti täysin määritellyksi. Tätä mallia kutsutaan siis abstraktiksi tietotyyppiä. Itse tietorakenne voidaan tallentaa muistiin halutulla tavalla (esimerkiksi jono voidaan toteuttaa käyttäen vektoria tai linkitettyä rakennetta) samoin kuin operaatiot voidaan implementoida (koodata) ohjelmoijan mieltymyksen mukaan.

#### 2.8.3.1 Lista

**Lista** (list) on dynaaminen rakenne, joka koostuu alkiosta, jotka on järjestetty peräkkäin siten, että jokaisella alkiolla, paitsi viimeisellä, on yksikäsitteinen seuraaja ja jokaisella alkiolla, paitsi ensimmäisellä, on yksikäsitteinen edeltäjä. Peräkkäisen rakenteensa vuoksi listarakennetta kutsutaan toisinaan myös **peräkkäisrakenteeksi** (sequence). Tarvittavia listaan kohdistettavia operaatioita ovat esimerkiksi: alkion poisto ja lisäys listan mielivaltaiseen kohtaan, listan tyhjyyden tarkistus, listan ensimmäisen alkion ja loppuosan palauttavat operaatiot. Lista voidaan myös määrittellä rekursiivisesti seuraavalla tavalla.

**Määritelmä.** Lista on joko

- tyhjä lista tai
- alkio, jota seuraa lista

**Esimerkki.** Listarakenteita.

Sana on kirjainten muodostama lista.

Lause on sanojen muodostama lista.

Luvun esitys on numeroiden muodostama lista.

Juna on tietue, jonka kaksi komponenttia ovat veturi ja vaunujen muodostama lista.

Puhelinluettelo on henkilötietueiden muodostama lista, jossa kukin tietue sisältää esimerkiksi nimen, osoitteen ja puhelinnumeron.

Lista soveltuu tietorakenteeksi silloin, kun tietoalkioita käsitellään peräkkäin. Tämä voidaan kuvata seuraavalla algoritmilla:

```
Ota käsiteltäväksi listan ensimmäinen alkio
WHILE ei olla listan lopussa DO
    käsittele listan alkio
    siirry seuraavaan alkioon
ENDWHILE
```

Kun rajoitetaan listalle suoritettavia toimenpiteitä (esim. lisäys tai poisto sallitaankin vain tiettyyn kohtaan listaa) saadaan seuraavat tietorakenteet (puhutaan myös jono- ja pinoperiaatteista), joilla on tärkeät sovellusalueet:

- **Jono** (queue) on rakenne, jolle on määritelty vain operaatiot alkion lisäämiseksi ja poistamiseksi sekä tyhjyyden testaus. Alkiot poistetaan samassa järjestyksessä, jossa ne on lisätty (vrt. kassajono). Jonorakenne noudattaa ns. FIFO-periaatetta (First In First Out). Tämä voidaan implementoida listalla siten, että alkion lisäys tapahtuu aina listan toiseen päähän ja poisto (ellei lista ole tyhjä) toisesta päästä.
- **Pino** (stack) on rakenne, jolle on määritelty vain operaatiot alkion lisäämiseksi ja poistamiseksi sekä tyhjyyden testaus. Pinosta poistetaan aina viimeksi lisätty alkio (vrt. lautaspino). Pinorakenne noudattaa ns. LIFO-periaatetta (Last In First Out). Tämä voidaan implementoida listalla siten, että alkion lisäys ja poisto kohdistuvat aina listan samaan päähän.

### 2.8.3.2 Listan toteutus

Lista (niin yleinen listarakenne kuin esim. pino- tai jonorakenne) voidaan toteuttaa käyttämällä tallennusrakenteina vektoreita tai dynaamisia linkitettyjä rakenteita, jotka ovat käytettävissä useimmissa imperatiivisissa ohjelmointikielissä. Vektorin käyttöä listan implementoinnissa rajoittaa sen staattisuus. Tästä on kuitenkin myös etunsa, esimerkiksi taulukon alkioon on helppoa viitata järjestysnumeron perusteella, ja sen arvoa voidaan myös helposti muuttaa. Abstrakti tietotyyppi pino toteutetaan vektorilla kappaleessa 2.8.4.

Jos käsiteltävä lista on luonteeltaan vakiomittainen (esim. alkio kutakin vuoden kuukautta kohti), kannattaa lista implementoida käyttäen vektoria. Sen sijaan jos lista on aidosti dynaaminen, kannattaa käyttää ohjelmointikielen valmiita listarakenteita tai linkitettyjä rakenteita, joskin vektoriakin voi käyttää, kuten seuraavasta esimerkistä näemme.



Funktionaalisissa ja logiikkakielissä listan käsite on keskeinen, niinpä niissä on paremmat välineet listojen käsittelyyn, mutta niihin palataan luvussa 6.

**Esimerkki:** Dynaamisen listan toteutus vektorilla. Tietoabstraktiot eli abstraktit tietotyypit kuvaavat tiedon loogisen rakenteen lisäksi sen käyttäytymisen määrittelemällä tarkoin sille sovellettavissa olevat operaatiot. Esimerkiksi vaikka linkitettyjä rakenteita ei olisi käytettävissä, voidaan dynaaminen lista (dynaamisesti käyttäytyvä lista) toteuttaa staattisesti määrittelemällä riittävän iso vektori, ja käyttämällä sen alusta kulloinkin tarvittava määrä alkioita. Taulukon lisäksi tarvitaan tieto listan koosta. Taulukko ja listan kulloisenkin pituuden kertova alkio kootaan usein yhteen tietueeksi tai oliokieliä luokaksi.

Kun tälle rakenteelle vielä kirjoitetaan dynaamiset lisäys- ja poisto-operaatiot, on määritelty abstrakti dynaaminen listatyyppi, jolla on staattinen toteutus. Oletetaan, että taulukolle on varattu  $n$  ( $n$  on tarpeeksi suuri luku, jonka arvioidaan riittävän) komponenttia ja listan todellinen pituus on  $k$  alkioita,  $k < n$ . Ellei lisäys- tai poistopaikkaa ole määrätty (kuten esim. jonolla ja pinolla), on operaatiot helpointa kohdistaa listan loppuun. Jos lisäys suoritetaan listan alkuun tai keskelle, on kaikkia lisättävän alkion jälkeen tulevia alkioita siirrettävä eteenpäin. Seuraava algoritmi lisää uuden alkion  $a$  listan  $L$ :nneksi ( $1 \leq i \leq k+1$ ) alkioiksi, jolloin tietenkin myös  $k$  kasvaa yhdellä:

```

k := k+1
IF i = k THEN (* lisäys listan loppuun *)
    L[i] := a
ELSE (* lisäys listan alkuun tai keskelle *)
    (* Tehdään tilaa kohtaan i eli siirretään alkioita kohdasta i alkaen yhden verran eteenpäin.
       Tämä täytyy tehdä lopusta alkaen kohtaa i+1 saakka, jotta i:s komponentti ei tuhoutu. *)
    FOR x := k, k-1, ..., i+1 DO L[x] := L[x-1] ENDFOR
    L[i] := a (* asetetaan alkio a kohtaan i *)
ENDIF

```

### 2.8.3.3 Puu

**Puu** (tree) on lineaarisen listan yleistys. Puussa kaikilla alkioilla (paitsi ensimmäisellä) on yksikäsitteinen edeltäjä kuten listassakin, mutta alkioilla voi olla monta seuraajaa ja eri solmuilla voi olla eri määrä seuraajia. Näin rakenteesta tulee hierarkia. Tieto esitetään rakenteen haarautumiskohdissa eli **solmuissa** (nodes). Puun haarat eli oksat (branches) edustavat loogisia suhteita kahdella peräkkäisellä tasolla olevien solmujen välille. Solmun seuraajia sanotaan sen **pojiksi**, ja solmun edeltäjää sanotaan solmun **isäksi**. Joskus käytetään myös termiparia vanhempi-lapsi<sup>1</sup>. Puu piirretään yleensä niin, että isä-poika -suhteet kulkevat ylhäältä alaspäin ja suhde piirretään viivana (kaarena) solmujen välillä. Hierarkian ylimmän tason solmua nimitetään puun **juureksi** (root) ja alimman tason solmuja nimitetään **lehdiksi** (leaves). Muita solmuja sanotaan **sisäsolmuiksi**. Puun haaroja, jotka muodostavat solmun ja sen välittömän tai välillisen seuraajan yhdistävän reitin, sanotaan näiden solmujen väliseksi **poluksi** (path). Polun **pituus** on polulla olevien haarojen (kaarien) lukumäärä. Puun **korkeus** on taas pisimmän polun pituus juuresta lehtisolmuun. Juurisolmulla ei ole edeltäjää, muilla solmuilla on yksi edeltäjä. Lehtisolmuilla ei ole seuraajia, muilla solmuilla on yksi tai useampia seuraajia. Solmun **aste** on solmun seuraajien lukumäärä.

Listojen tavoin puu samastetaan usein juuren, hierarkiassa ylimmän alkionsa kanssa. Puu on rekursiivinen rakenne: jokainen puun solmu on itse yhden pienemmän puun eli alipuun (subtree) juurisolmu. Lehtisolmukin voidaan ajatella juurisolmuksi: se on oman alipuunsa ainoa solmu. Puun jokainen solmu edustaa siis omaa (ali)puutaan. Solmun seuraajan edustamaa alipuuta sanotaan solmun **poikapuuksi**.

<sup>1</sup> isä-sanan valinta perustuu sen parempaan taipuvuuteen verrattuna vanhempi-sanaan, joten kyseessä ei siis ole missään tapauksessa sukupuolinen syrjintä!

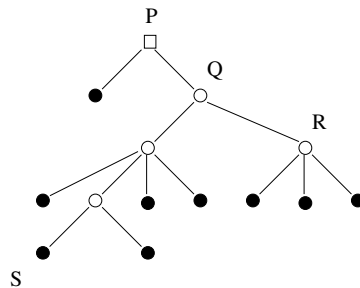
Usein sovitaan, että puun jokaisella solmulla on tarkalleen  $k$  poikapuuta, joista osa voi olla tyhjiä (puu, jossa ei ole yhtään solmua). Tällaista puuta kutsutaan ***k*-haaraiseksi** tai ***k*-ariseksi** (puun ariteetti on  $k$ ) puuksi.  $k$ -haarainen puu voidaan siis määritellä rekursiivisesti myös seuraavasti:

**Määritelmä.** *k*-haarainen puu (*k*-ary tree) on joko

- tyhjä puu, jossa ei ole lainkaan solmuja, tai
- se koostuu juurisolmusta, jota seuraa  $k$  poikapuuta, jotka ovat  $k$ -haaraisia puita.

Solmun aste on siis sen ei-tyhjiä poikapuiden lukumäärä. Jos puun jokaisen solmun, paitsi lehtisolmujen, kaikki  $k$  poikapuuta ovat ei-tyhjiä, sanotaan puuta ***täydelliseksi***  $k$ -haaraiseksi puuksi. Tällöin polun pituus juuresta jokaiseen lehtisolmuun on yhtä pitkä.

**Esimerkki.** Seuraavassa kuvassa puun  $P$  juuri on merkitty neliöllä, sisäsolmut täyttämättömällä ympyröillä, lehdet täytetyillä ympyröillä, mutta tyhjiä alipuita ei ole merkitty näkyviin. Puun ariteetti on 4, mutta se ei ole täydellinen, koska esim.  $Q$ :n aste on 2. Solmun  $R$  edustama puu on  $Q$ :n poikapuu. Polun pituus juurisolmusta  $P$  solmuun  $S$  on 4, joka on myös puun  $P$  korkeus (pisin polku). Huomaa, että kuvasta ei näy esim. se, mitkä juuren neljästä poikapuusta ovat tyhjiä. Tyhjät alipuut voidaan tarvittaessa piirtää kuvaan jollakin erikoissymbolilla.



Puiden rekursiivisen rakenteen vuoksi puita käsittelevät algoritmitkin ovat yleensä rekursiivisia. Myös useimmat tarkasteltavat puiden ominaisuudet ovat rekursiivisia. Esimerkiksi  $k$ -haaraisen puun korkeus<sup>2</sup> voidaan määritellä (ja sen määräävä algoritmi voidaan kirjoittaa) rekursiivisesti seuraavalla tavalla:

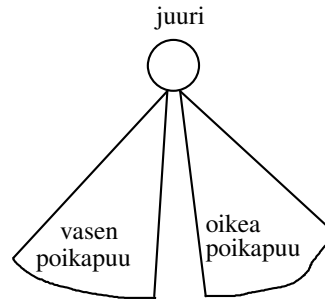
**Määritelmä.** Olkoon  $p$   $k$ -haarainen puu.

- Jos  $p$  on tyhjä tai koostuu vai juurisolmusta,  $p$ :n korkeus on 0.
- Jos  $p$ :llä on vähintään yksi ei-tyhjä poikapuu,  $p$ :n korkeus on  $1 + \max(h_1, h_2, \dots, h_k)$ , missä  $h_i$  on puun  $p$   $i$ :nnessä poijan korkeus.

### 2.8.3.4 Binääripuut

Lista on 1-haarainen puu (ariteetti on 1, ns. unaarinen puu). 2-haaraisista puuta sanotaan ***binääripuiksi***. Binääripuussa solmun seuraajia sanotaan solmun ***vasemmaksi*** ja ***oikeaksi*** pojaksi:

<sup>2</sup> joissakin oppikirjoissa korkeus määritellään yhtä isommaksi, jolloin se tarkoittaa puun tasojen lukumäärää (tyhjän puun korkeus on 0 ja vain yhden solmun sisältävä puun korkeus on 1).



Vaikka binääripuut ovat puista kaikkein yksinkertaisimpia, ne ovat kuitenkin erittäin ilmaisuvoimaisia konstruktioita. Siksi binääripuut ovatkin hyvin paljon käytettyjä tietorakenteita. Puista esiintyy lukemattomia erilaisia variantteja, jotka soveltuvat mitä moninaisimpiin tehtäviin.

Vaikka puu ei ole lineaarinen rakenne, voidaan puu esittää lineaarisena listana luettelomalla puun solmut tiettyssä (systemaattisessa) järjestyksessä. Järjestys ei viittaa tässä solmujen arvojen suuruusjärjestykseen, vaan luettelointi- eli läpikäyntijärjestykseen. On myös huomattava, että useimmiten lineaarisen esityksen perusteella ei voida päätellä alkuperäistä puurakennetta. Kun oletetaan, että binääripuun jokaisen solmun kohdalla vierailaan vasemmassa pojassa aina ennen oikeata poikaa, jäävät jäljelle seuraavat kolme binääripuun läpikäyntijärjestystä:

- **esijärjestys** (*preorder tree walk*): solmussa käydään aina ennen solmun poikia
- **välijärjestys** (*inorder*): solmussa käydään aina poikien välissä
- **jälkijärjestys** (*postorder*): solmussa käydään aina poikien jälkeen

Puille voidaan määritellä useita operaatioita. Seuraavissa esimerkeissä käytetään binääripuiden käsittelyssä tarvittavia operaatioita: puun tyhjyyden testaus (merkitään:  $p = \text{tyhjä}$  tai  $p \neq \text{tyhjä}$ ), puun  $p$  oikean ja vasemman alipuun määrääminen (merkitään:  $p.\text{oikea}$  ja  $p.\text{vasen}$ ), alkion lisääminen puuhun (kyseinen moduuli esitetään listan lajittelun yhteydessä edempänä). Muita tarvittavia operaatioita ovat esimerkiksi: alkion poistaminen puusta, onko alkio puussa, jne.

**Esimerkki.** Algoritmi puun alkioden tulostukseen välijärjestyksessä. Merkitään solmun  $p$  tietokentän arvoa merkinällä  $p.\text{arvo}$ .

```

MODULE välijärjestys(p)
  IF p <> tyhjä THEN
    välijärjestys(p.vasen)
    tulosta(p.arvo)
    välijärjestys(p.oikea)
  ENDIF
ENDMODULE

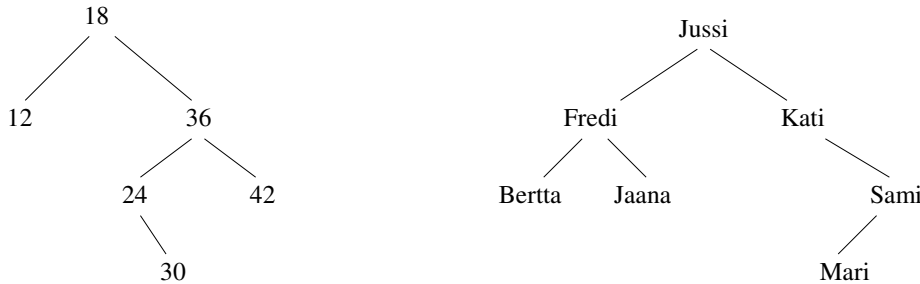
```

Yleisimmin käytetty binääripuun läpikäyntijärjestys on välijärjestys. Seuraavaksi määritellään, milloin binääripuun sanotaan olevan **järjestetty** (tai **lajiteltu**). Nyt järjestys viittaa puun solmujen arvojen järjestykseen. Merkitään tätä järjestystä symbolilla  $<$  ja oletetaan, että *puun solmujen arvot ovat keskenään erisuuria*.

**Määritelmä.** Puu  $p$  on *järjestetty binääripuu*, jos

- $p$  on tyhjä tai
- $p$  on ei tyhjä ja seuraavat ehdot ovat voimassa:
  1.  $p$ :n juuren arvo  $>$   $p$ :n vasemman poikapuun jokaisen solmun arvo
  2.  $p$ :n juuren arvo  $<$   $p$ :n oikean poikapuun jokaisen solmun arvo
  3.  $p$ :n vasen ja oikea poikapuu ovat järjestettyjä binääripuita.

**Esimerkki.** Järjestettyjä binääripuita.



Esimerkiksi puun Jussi vasen poikapuu on alipuu Fredi, joka sisältää (välijärjestyksessä) solmut Bertta, Fredi ja Jaana. Solmun Kati edustaman puun vasen alipuu on tyhjä. Kaikki puun Fredi solmut ovat aakkosissa ennen solmua Jussi, joka puolestaan edeltää kaikkia puun Kati solmuja (välijärjestyksessä Kati, Mari ja Sami).

Vasemman puoleisen puun välijärjestys voidaan kehittää seuraavasti: ensin saadaan 12, 18, 36 ja tässä edelleen 36 korvataan puun 36 välijärjestyksellä, jolloin saadaan 12, 18, 24, 36, 42, josta edelleen saadaan 12, 18, 24, 30, 36, 42.

**Huom.** Järjestettyä binääripuuta kutsutaan usein myös *binääriseksi hakupuuksi*, koska tällaisesta puusta löydetään haluttu tieto nopeasti. Lisäksi järjestetyllä binääripuulla on seuraava tärkeä ominaisuus: Jos se luetaan välijärjestyksessä, on saatu lineaarinen lista järjestyksessä. Tämä tosiasia todistetaan induktiolla kappaleessa 3.3.2. Tätä ominaisuutta käytetään hyväksi jo seuraavassa lajittelualgoritmissa.

**Esimerkki.** Järjestetyn binääripuun käyttö listan järjestämisessä.

Tehtävänä on lajitella annettu nimilista aakkosjärjestykseen. Kaksivaiheisen lajittelualgoritmin idea on seuraava:

```

järjestämätön lista → järjestetty binääripuu → järjestetty lista
MODULE lajittele(lista list) RETURNS lajitellun listan
  IF list on tyhjä THEN RETURN tyhjä lista ENDIF
  p := ListaPuuksi(list) (* konstruoi listan list alkoista järjestetyn binääripuun p *)
  uusilist := PuuListaksi(p) (* konstruoi järjestetyn binääripuun p alkoista uuden järjestetyn listan *)
  RETURN uusilist
ENDMODULE
    
```

*Moduuli ListaPuuksi:* Parametri list viittaa aluksi listan alkuun, eli listan ensimmäiseen alkioon. Listan alkoista muodostetaan järjestetty binääripuu lisäämällä ne puuhun yksitellen. Aluksi puuhun laitetaan listan ensimmäinen alkio, joka asetetaan puun juureksi. Jokaisen lisäyksen jälkeen list siirtyy viittaamaan aina listan seuraavaan alkioon, joka lisätään puuhun toistorakenteessa. Huomaa tässä, että p osoittaa koko ajan puun juureen, ja näin listan jokaisen alkion lisäystalkastelu puuhun aloitetaan aina puun juuresta.

```
(* Alkuehto: lista list ei ole tyhjä *)
MODULE ListaPuuksi(lista list) RETURNS järjestetyn binääripuun
  p.arvo := list.arvo (* listan ensimmäinen alkio asetetaan puun juureksi *)
  p.vasen := tyhjä
  p.oikea := tyhjä
  list := list.seuraava
  WHILE list <> tyhjä DO
    LisääPuuhun(list.arvo, p) (* suorituksen jälkeen p viittaa aina p:n juureen *)
    list := list.seuraava
  ENDWHILE
ENDMODULE

MODULE LisääPuuhun(alkio a, järjestetty binääripuu p)
  IF p = tyhjä THEN
    p.arvo := a
    p.vasen := tyhjä
    p.oikea := tyhjä
  ELSE IF a < p.arvo THEN
    LisääPuuhun(a, p.vasen)
  ELSE
    LisääPuuhun(a, p.oikea)
  ENDIF
ENDIF
ENDMODULE
```

*Moduuli LisääPuuhun:* Lisättäessä alkio järjestettyyn binääripuuhun on huolehdittava siitä, että puu pysyy järjestettynä. Tyhjään puuhun lisääminen on helppoa: lisättävästä alkioista tulee puun ainoa solmu, siis juuri. Lisättäessä alkio ei-tyhjään puuhun täytyy sen oikea paikka määrätä. Paikka määrätään välijärjestyksen mukaan: jos lisättävä alkio on puun juurta pienempi, se lisätään rekursiivisesti vasempaan poikapuuhun, ja jos se on juurta suurempi, se lisätään rekursiivisesti oikeaan poikapuuhun. Itse lisäys tapahtuu lopulta aina tyhjään alipuuhun – näin puussa jo olevia alkioita ei jouduta siirtämään. Uuden solmun arvoksi tulee lisättävä alkio, ja uuden solmun kummaksikin seuraajaksi asetetaan tyhjä puu. Muodostettavan puun juureksi p tulee ensimmäiseksi lisättävä alkio.

*Moduuli PuuListaksi:* Muodostettaessa järjestetyn binääripuun alkioista järjestettyä lineaarista listaa, määräytyy kunkin alkion paikka välijärjestyksen mukaan: kutakin alkioita ennen tulevat listaan kaikki sitä välijärjestyksessä edeltävät alkioita ja sen jälkeen tulevat kaikki sitä välijärjestyksessä seuraavat alkioita. Ensin muodostetaan lista puun p vasemman poikapuun alkioista. Sen perään lisätään juuri p, ja sen perään lisätään puun p oikean poikapuun alkioista muodostettu lista. Alkio lisätään aina listan loppuun, siis tyhjään alilistaan – näin listassa jo olevia alkioita ei jouduta siirtämään. Uusi solmu tulee listan viimeisen alkion seuraajaksi, ja sen arvoksi tulee lisättävä alkio. Muodostettavan listan ensimmäiseksi alkioiksi tulee näin puun kaikkein vasemmanpuoleisin – eli arvoltaan pienin – alkio.

```
MODULE PuuListaksi(järjestetty binääripuu p) RETURNS järjestetyn listan
  list := tyhjä
  IF p <> tyhjä THEN
    PuuListaksi(p.vasen)
    Lisää juuren arvo listan loppuun
    PuuListaksi(p.oikea)
  ENDIF
  RETURN list
ENDMODULE
```

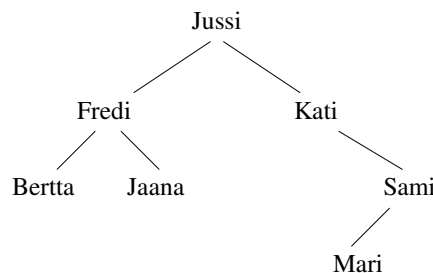
Algoritmin toteutuksen yksityiskohdat riippuvat käytettävän ohjelmointikielen mekanismeista, etenkin osoittimien (viittausten) toteutuksesta ja parametrinvälityksen yksityiskohdista. Nämä seikat sivuutetaan tässä. Edellä oleva versio kuitenkin toimii, jos käytetään arvoparametrivälitystapaa (parametrien kopiointi uusiksi muuttujiksi) kuten esim. Javassa.

### 2.8.3.5 Binääripuun toteutus

a) Binääripuu voidaan toteuttaa linkitettyinä rakenteena, jossa jokaiseen solmuun tallennetaan tietokenttien arvojen (tai arvon) lisäksi osoittimet vasempaan ja oikeaan poikapuuhun:

data 1
...
data n
osoitin vasempaan poikaan
osoitin oikeaan poikaan

b) Linkitetyn toteutuksen lisäksi puu voidaan toteuttaa muullakin tavoin. Usein on tarpeen esittää loogisesti hierarkkinen rakenne fyysisesti lineaarisena. Puu voidaan esittää merkkijonona ns. **sulkumerkki-** eli **termiesitystä** käyttäen. Puun sulkumerkkiesityksessä juuren pojat kirjoitetaan sulkeisiin juuren jälkeen. Tyhjää puuta merkitään tarvittaessa jollain erikoissymbolilla, vaikkapa symbolilla - . Tyhjän pojan merkitseminen on tarpeellista, jos eri pojilla on eri roolit. Lehtien tyhjiä poikia ei merkitä näkyviin. Näin esimerkiksi puun



termiesitys on

Jussi (Fredri (Bertta, Jaana), Kati (-, Sami (Mari,))).

Termiesityksessä puun solmut esitetään siis lineaarisesti käyttäen esijärjestystä.

c) Binääripuu voidaan esittää myös listana seuraavasti:

- Puun juuri on listarakenteen ensimmäinen alkio.
- Listan  $i$ . paikassa sijaitsevan solmun vasen poika sijaitsee paikassa  $2i$  ja oikea poika paikassa  $2i+1$ .

Esimerkiksi edellä tarkasteltu puu esitettäisiin listana seuraavasti: [Jussi, Fredri, Kati, Bertta, Jaana, -, Sami, Mari, -]. Puuttuvia poikia ei tietenkään voi jättää esittämättä, jos puun rakenne halutaan säilyttää. Tällaisia tietorakenteita sanotaan **implisiittisiksi**, koska on asetettu jokin yleinen sääntö, jonka avulla voidaan määrätä isä-poika-suhteet. Tällainen rakenne mahdollistaa puun toteuttamisen ja tallentamisen staattisesti, vektorina  $V$ .

Jos tässä rakenteessa  $V[i] > V[2i]$  ja  $V[i] > V[2i+1]$  kaikilla mahdollisilla  $i$ :n arvoilla, niin rakennetta kutsutaan **keoksi**. Kun tämä rakenne tulkitaan puuksi, tarkoittavat ehdot sitä, että puun jokaisen solmun vasen ja oikea poikasolmu ovat arvoltaan pienempiä kuin itse solmu, mutta poikasolmut voivat olla missä järjestyksessä tahansa. Näin ollen puun juuressa ja vektorin ensimmäisenä alkiona on suurin alkio (tai vastaavasti pienin jos varustetaan eo. ehdot  $<$  -vertailuilla). Kekoa voidaan käyttää erittäin tehokkaasti esimerkiksi lukujoukon maksimin (tai minimin) haun toteuttamiseen: suurin alkio löytyy aina nopeasti puun juuresta. Keon ylläpito on huomattavasti helpompaa kuin täydellisesti lajitellun binääripuun; eihän esimerkiksi maksimin etsimiseksi tarvitse tietää kaikkien alkioiden täydellistä keskinäistä järjestystä.

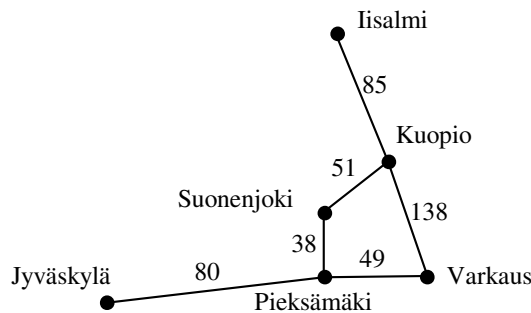
### 2.8.3.6 Graafi

**Graafi** (graph) eli **verkko** (net) on puurakenteen yleistys samalla tavalla kuin puu on listan yleistys. Graafi muodostuu joukosta solmuja (nodes) eli kärkiä (vertices) ja joukosta solmuja yhdistäviä **kaaria** (edges). Kahden solmun välillä on kaari, jos solmujen välillä vallitsee jokin tietty suhde (binäärinen relaatio). Kaaret eivät välttämättä muodosta hierarkiaa solmujen välille kuten puiden yhteydessä eikä solmuilla ole välttämättä seuraajaa tai edeltäjää. Graafin solmun **aste**<sup>3</sup> on niiden kaarien lukumäärä, joiden toisena päätepisteenä ko. solmu on.

**Esimerkki.** Edellä kaupunkien väliset etäisyydet tallennettiin staattiseen taulukkoon (matriisiin):

	Iisalmi	Jyväskylä	Kuopio	Pieksämäki	Suonenjoki	Varkaus
Iisalmi	0	254	85	174	136	223
Jyväskylä	254	0	169	80	118	129
Kuopio	85	169	0	89	51	138
Pieksämäki	174	80	89	0	38	49
Suonenjoki	136	118	51	38	0	87
Varkaus	223	129	138	49	87	0

Sama tieto voidaan myös esittää graafisesti karttana, johon on piirretty kaupunkeja yhdistävät reitit ja niiden pituudet. Tällainen kartta muodostaa graafin, jossa solmuja ovat kaupungit ja kaaria reitit. Solmun Kuopio aste on 3.



Kumpi esitysmuoto sitten on parempi? Esitystavan valinta riippuu ensisijaisesti tehtävästä, joskin valintaan vaikuttaa myös käytettävä ohjelmointikieli. Rakenne on pyrittävä valitsemaan niin, että se

- kuvaa mahdollisimman hyvin tiedon luonnollista rakennetta, ja
- mahdollistaa tarvittavien operaatioiden tehokkaan toteutuksen.

Esimerkissämme staattisen taulukon etuna on se, että kahden kaupungin välinen lyhin etäisyys on helposti ja nopeasti löydettävissä, kun taas graafista etäisyyden laskeminen vie aikaa ja voi vaatia vertailuja vaihtoehtoisten reittien välillä. Toisaalta dynaamisen graafin käyttöä puoltaa se, että tieyhteyksien muuttuessa tai parantuessa riittää lisätä tai muuttaa niitä kaaria, joita muutos välittömästi koskee. Esimerkiksi jos Kuopion ja Pieksämäen välille rakennettaisiin suora moottoritie, muuttuu Jyväskylän ja Iisalmen välinen etäisyys vastaavasti. Taulukkoesityksessä tämä aiheuttaisi muutostarpeen myös Jyväskylän ja Iisalmen kohdalle, kun taas graafissa riittää muuttaa Kuopion ja Pieksämäen välistä kaarta.

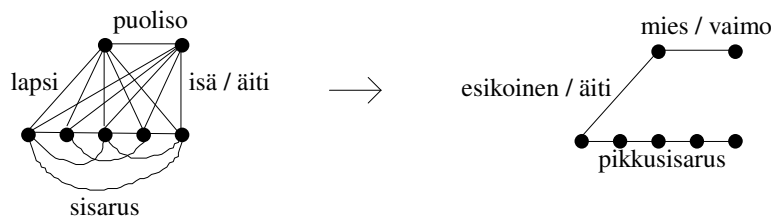
<sup>3</sup> puiden yhteydessä aste määriteltiin hiukan eri tavalla: puun solmun aste on sen ei-tyhjien poikapuiden lukumäärä

Myös graafit ovat yleisesti käytettyjä tietorakenteita, joista esiintyy useita variantteja. Yleisimpiä ovat:

- **Suunnattu ja suuntaamaton graafi.** Suunnatussa graafissa kaaret ovat yksisuuntaisia, ja suuntaamattomassa graafissa kaksisuuntaisia. Esimerkiksi maantiekarttaa kuvaava graafi on suuntaamaton, mutta kaupungin katuverkkoa kuvaava kartta, jossa solmuina ovat katujen risteykset, muodostaa suunnatun graafin, koska osa kaduista on yksisuuntaisia. Tällöin kaksisuuntaista katua kuvaamaan tarvitaan kaksi vastakkaisuuntaista kaarta.
- **Painotettu graafi.** Kahden solmun välillä voi olla myös useampia kaaria, tai kaaret voivat olla eri pituisia tai eri vahvuisia. Tällöin kuhunkin kaareen liitetään erityinen paino. Esimerkiksi kaupungin katukartassa kadut voitaisiin painottaa kuhunkin suuntaan menevien kaistojen lukumäärällä. Kaupunkien välisiä teitä kuvaava maantiekartta toteutetaan painotettuna graafina, jossa painoina ovat kaupunkien väliset etäisyydet.
- **Suunnattu syklitön graafi,** DAG (Directed Acyclic Graph) on yleisesti käytetty rakenne, jonka erityispiirteet käyvät ilmi sen nimestä. Syklillä tarkoitetaan kaarien muodostamaa polkua, joka päättyy samaan solmuun mistä se alkaa.

Edellä tarkasteltiin graafeja, joissa jokaisella kaarella oli sama merkitys (esim. kaupungista A on tieyhteys kaupunkiin B). Voimme varustaa graafin myös erityyppisillä kaarilla, jolloin solmusta voi lähteä useita kaaria, jotka kuvaavat eri asioita. Esimerkiksi sukupuuta kuvaava graafi, jossa solmuina ovat yksittäiset ihmiset, muodostuu DAG-rakenteeksi, jossa kaarilla on oma roolinsa (esimerkiksi 'vaimo' tai 'lapsi') ja näin myös suuntansa. Kun kaaren 'vaimo' suunta käännetään, saadaan kaari 'mies', ja käännettäessä kaari 'lapsi' saadaan 'vanhempi'. Graafi ei ole puu, koska jokaisella lapsella on kaksi vanhempaa ja vanhemmilla voi olla useita lapsia. Graafi on syklitön, mutta jos sukugraafiin lisätään symmetrisiä tai välillisiä sukulaisuussuhteita – kuten 'puoliso' tai 'serkku' – voi graafiin tulla syklejä.

Tietojen esittämiseen valitut relaatiot (eli kaarien merkitykset) tulee valita huolella. Kaiken tarpeellisen on oltava mukana, mutta turhia kaaria on usein syytä välttää jo selkeydenkin vuoksi. Esimerkiksi sukugraafissa lasten lukumäärälle on vaikea ja turhaa asettaa sopivaa ylärajaa. Näin ollen 'lapsi'-relaatio ei ole käytännöllinen, ja niinpä on parempi korvata se unaarisilla relaatioilla 'esikoinen' ja 'pikkusisar'. Lisäksi sukugraafissa hyvä ratkaisu on käyttää symmetrisen relaation 'puoliso' sijasta relaatioita 'mies' ja 'vaimo':



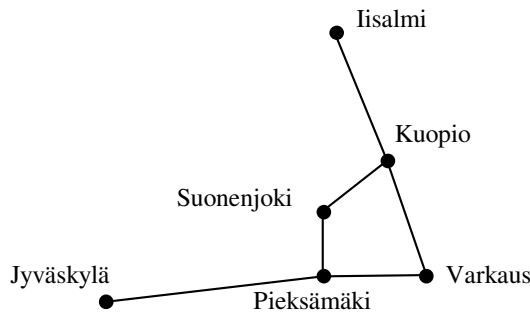
### 2.8.3.7 Graafin toteutus

- Graafit voidaan toteuttaa esimerkiksi linkitetyillä rakenteilla, joissa kustakin solmusta asetetaan linkit kaikkiin niihin solmuihin, joihin tarkasteltavasta solmusta on kaari.



b) Graafit voidaan toteuttaa myös staattisesti käyttämällä **bittikarttaesitystä (vierekkäisyysmatriisia)**. Bittikartta on matriisi, jossa graafin solmut ovat sekä rivi-että sarakeindeksinä, ja  $i$ . rivin  $j$ . sarakkeen alkioiksi asetetaan 1, jos solmujen  $i$  ja  $j$  välillä on kaari, ja 0 muutoin. Bittikarttaesitys on helppo yleistää painotetuille graafeille tallentamalla matriisiin ykkösen sijasta tarkasteltavan kaaren paino. Jos graafissa on useammantyyppisiä kaaria, tulee kullekin kaarityypille muodostaa oma bittikarttansa. Tällöin linkitettyjen rakenteiden käyttö on kuitenkin luontevampaa.

**Esimerkki.** Graafia



vastaa bittikartta

	Iisalmi	Jyväskylä	Kuopio	Pieksämäki	Suonenjoki	Varkaus
Iisalmi	0	0	1	0	0	0
Jyväskylä	0	0	0	1	0	0
Kuopio	1	0	0	0	1	1
Pieksämäki	0	1	0	0	1	1
Suonenjoki	0	0	1	1	0	0
Varkaus	0	0	1	1	0	0

### 2.8.4 Abstraktien tietotyyppien implementoinnista

Useissa ohjelmointikielissä ei ole juuri abstraktien tietotyyppien implementointiin tarkoitettuja välineitä, ja niinpä tiedon rakenne määritellään tyyppien määrittelyosassa ja operaatiot tavallisina aliohjelmoina, jolloin usein ei ajatellakaan, että kyseessä on abstraktin tietotyypin implementointi. Tässä on se huono puoli, että itse tieto ja operaatiot ovat erillään ja toisaalta se, että tieto ei ole suojattu vääriltä operaatioilta. Tällöin sallittujen operaatioiden joukkoa ei ole edes mietitty, vaan operaatioita lisätään 'tarpeen mukaan' vailla kokonaisvaltaista suunnittelua.

Monissa kielissä on kuitenkin mahdollista sitoa tietorakenne ja siihen kohdistettavat operaatiot yhdeksi kokonaisuudeksi esittämällä niiden määrittely- ja implementointiosia yhdessä. Näissäkin rakenteissa on itse tallennusrakenteisiin mahdollisuus viitata suoraan käyttämättä pelkästään tyyppille määriteltyjä operaatioita.

Kaikista turvallisin tapa on ns. **kapselointi** (encapsulation), jossa tietorakenteen ja operaatioiden implementointi on täysin kätkeyty käyttäjältä, jolloin tietorakenteesta näkyvät ulospäin vain rakenteeseen kohdistettavat operaatiot. Tällöin siis abstrakti malli ja implementointi ovat täysin erotetut toisistaan ja itse tallennusrakenteeseen ei voi suoraan viitata, vaan tietorakennetta voi käsitellä ainoastaan sille määriteltyjen operaatioiden kautta. Kapseloinnissa määritellään tietorakenteen

- **julkinen** (public) osa: tietotyypin nimi ja operaatioiden nimet, ja
- **yksityinen** (private) osa: tietorakenteen tallennusrakenne ja operaatioiden implementoinnit.

Kapselointi ei ole mahdollista kaikissa ohjelmointikielissä, mutta siihen liittyviä suunnitteluperiaatteita voi luonnollisesti soveltaa, vaikka ohjelmointikielessä ei olisikaan valmiina kapseloinnin toteuttamiseen tarvittavia työkaluja.

**Esimerkki.** Esitetään abstraktin tietotyypin pino määrittely (kuvitteellisella ohjelmointikielellä) a) ilman kapselointia ja b) kapseloituna. Pinohan oli sellainen lista, jossa alkion lisäys (push) ja poisto (pop) tapahtuu aina listan samaan päähän (eli tarkoittaa sitä, että pinosta otetaan pois aina sinne viimeksi lisätty alkio). Määrittely on vieläpä geneerinen, jolloin pinoon tallennettavien alkioiden tyyppi (T) annetaan määrittelyn parametrina. Tällöin voidaan tällaisen pinomuuttujan määrittelyn yhteydessä kertoa pinon alkioiden tyyppi; esim. määrittely p: Pino[Integer] tarkoittaa sitä, että tunniste p on liitetty pinoon, jonka alkiot ovat kokonaislukuja.

```
a) DEFINE DATATYPE Pino[T] (* kapseloimaton *)
    IMPLEMENTATION
        pinon_max_koko = 100
        Pino=RECORD
            pinon_koko: Integer
            alkiot: ARRAY[1... pinon_max_koko] OF T
        ENDRECORD
    OPERATIONS
        MODULE push(p: Pino, a: T) (* asettaa alkion a pinon päällimmäiseksi alkiksi *)
            IF p.pinson_koko = pinon_max_koko THEN 'virhe'
            ELSE
                p.pinson_koko:=p.pinson_koko+1
                p.alkiot[p.pinson_koko]:=a
            ENDIF
        ENDMODULE
        MODULE pop(p: Pino, a: T)
            (* ottaa pinon p päällimmäisen alkion ja sijoittaa sen a:n arvoksi *)
            IF p.pinson_koko=0 THEN 'virhe'
            ELSE
                a := p.alkiot[p.pinson_koko];
                p.pinson_koko := p.pinson_koko - 1
            ENDIF
        ENDMODULE
        MODULE make(p: Pino)
            (* alustaa ja luo tyhjän pinon *)
            p.pinson_koko := 0
        ENDMODULE
    ENDDDEFINE
```

```

b) DEFINE DATATYPE Pino[T] (* kapseloitu *)
    PUBLIC
        pinon_max_koko=100
        MODULE push(p: Pino, a: T)
        MODULE pop(p: Pino, a: T)
        MODULE make(p:Pino) (* pinon luonti *)
    PRIVATE
        Pino=RECORD
            pinon_koko: Integer
            alkiot: ARRAY[1... pinon_max_koko] OF T
        ENDRECORD
        MODULE push(a,p)
            ...
        ENDMODULE
        MODULE pop(p,a)
            ...
        ENDMODULE
        MODULE make(p: Pino)
            ...
        ENDMODULE
    ENDDDEFINE

```

Kun rakenne on kapseloimaton, niin silloin itse tallennusrakenteeseen voidaan suoraan viitata käyttämättä pino-operaatioita push ja pop; voitaisiin esim. kirjoittaa p.alkiot[1]:=2 eli asetetaan pinon ensimmäiseksi alkiksi luku 2. Tämä ei tietenkään ole toivottavaa ja niinpä kapseloidussa rakenteessa em. lause on kielletty, joten tulee kirjoittaa push(p,2). Operaatioiden push ja pop implementoinnissa tulee myös varmistaa, että ei yritetä ottaa alkia tyhjästä pinosta (p.pinson\_koko=0) ja että ei viitata vektorin 'yli' (p.pinson\_koko>pinon\_max\_koko).

## 2.8.5 Oliokeskeinen ohjelmointi

Tässä monisteessa modularisoinnin käsite on liitetty algoritmiseen modularisointiin, jossa alkuperäinen tehtävä jaetaan osatehtäviin, moduuleihin. Yleisesti ottaen modularisointiin kuuluu yhtä hyvin käsiteltävän tiedon modularisointi. **Oliokeskeisessä ohjelmoinnissa** (object oriented programming) eli **olio-ohjelmoinnissa** modularisointi tarkoittaa lähinnä käsiteltävän tiedon modularisointia tai oikeastaan käsiteltävien tietotyyppien modularisointia. Oliokeskeisen ohjelmoinnin yhtenä lähtökohtana on ollut myös edellä esitetty hyväksi havaittu kapselointiperiaate, jolla voidaan estää suorat viittaukset käytettyjen tietotyyppien attribuuttien arvoihin. Seuraavassa esitetään lyhyesti oliokeskeisen suunnittelun ja ohjelmoinnin peruskäsitteet.

**Oliokeskeisessä analyysissä** tulee löytää tarkasteltavan maailman sellaiset tieto-objektit, joilla on merkitystä tarkasteltavan tehtävän kannalta. Tällöin tarkasteltava tehtävä abstrahoidaan ja tuloksena saadaan tarkasteltavien tieto-objektien abstraktit mallit. Abstraktissa mallissa kuvataan objektien relevantit ominaisuudet (attribuutit) ja ne operaatiot (metodit), joita kuhunkin objektiin voidaan soveltaa. Abstrakti malli riippuu siitä mihin tarkoitukseen malli tehdään. Esimerkiksi jos tarkastellaan autoja, niin auton korjaajan ja auton ajajan abstrakti malli autosta on hyvin erilainen.

Analyysin tavoitteena on siis löytää abstraktit tietotyypit (abstraktien tietotyyppien yhteydessä mainittiin, että ADT on ohjelmointikielistä ja näin ollen myös tyyppi-konstruktoreista riippumaton käsite).

Oliokeskeisen ohjelmoinnin keskeiset käsitteet ovat **luokka** (class) ja **olio** eli **objekti** (object). Luokka on staattinen käännoaikainen käsite, jolla ADT voidaan suoraan

implementoida eli luokan avulla määritellään uusi **tyyppi**. Olio on taas dynaaminen ohjelman suorituksen aikana luotava rakenne, luokan alkion **esiintymä** eli **instanssi**. Luokka on siis myös tallennusrakenne. Luokassa määritellään luokkaan kuuluvien olioiden abstrakti malli esittämällä ne attribuutit (ominaisuudet, tietokentät) ja operaatiot (menetelmät eli moduulit), joiden avulla jokainen luokan alkio, olio, voidaan karakterisoida. Luokan käsite vastaa siis jo aiemmin käsitellyä tyyppin käsitettä. Jokainen käsiteltävä olio kuuluu siis johonkin luokkaan eli siis on jotain tyyppiä. Olio ei ole kuitenkaan puhtaasti tietoa säilyttävä (kuten esim. tietue), vaan myös toiminnallinen, koska olioon voidaan kohdentaa olion luokassa määriteltyjä operaatioita. Olemassa olevan olion tila määritellään attribuuttiensa (ominaisuuksien arvoja, jotka kuvaavat olion tilaa kullakin hetkellä) avulla. Olion **tilaa** voidaan muuttaa muuttamalla sen attribuuttien arvoja tai soveltamalla siihen luokassa määriteltyjä operaatioita, **metodeja**.

Oliokieli on yleensä valmiina mekanismit, joiden avulla voidaan helposti noudattaa – ja tulee noudattaa – edellä esitettyä kapselointiperiaatetta. Tällöin tietorakenteen ja siihen kohdistuvien operaatioiden implementointi kätkeytyy käyttäjältä, jolloin tietorakenteesta näkyvät ulospäin vain ne operaatiot, joilla rakennetta voidaan käsitellä. Tällöin siis itse tallennusrakenteeseen ei voida suoraan viitata luokan ulkopuolelta, vaan tietorakennetta voidaan käsitellä ainoastaan sille määriteltyjen operaatioiden kautta.

Seuraavassa esitetään ensin hyvin yksinkertainen esimerkki luokasta, jolla karakterisoidaan henkilöitä. Tässä karakterisointi tapahtuu attribuuttien nimi, osoite ja syntymävuosi perusteella. Kun olio luodaan, tulee näille attribuuteille antaa arvot. Alla olevassa luokassa ei ole kuitenkaan yhtään muutosmetodia, joten rakenne vastaa aiemmin esitettyä tietuerakennetta. Sen jälkeen käsitellään pinorakennetta, joka sisältää pinon luonnolliset muutosmenetelmät: push (laita alkio pinoon) ja pop (ota alkio pinosta).

**Esimerkki.** Määritellään Eiffel-kielillä luokka, jolla voidaan karakterisoida henkilöitä (kommentti alkaa merkkiparilla -- ja jatkuu aina rivin loppuun). Tässä kaikki attribuutit ovat julkisia, joten tämä ei noudata edellä esitettyä kapselointiperiaatetta (ei hyvä!). Luokka ei myöskään sisällä metodeja olioiden käsittelyyn, joten tämä määrittely vastaa edellä esitettyä tietueen käsitettä.

```

class PERSON      -- määriteltävän luokan nimi
creation          -- ilmoitetaan luontirutiinin nimi; tässä make
  make            -- uusi luokan instanssi luodaan operaatiolla make
feature          -- attribuuttien (piirteiden) määrittely alkaa
  make(n: STRING; a: STRING; y: INTEGER) is
  -- proseduuri, joka alustaa henkilön parametreina annetuilla tiedoilla oliota luotaessa
  do
    name:=n
    address:=a
    year_of_birth:=y
  end
  name: STRING
  address: STRING
  year_of_birth: INTEGER
end -- class PERSON

```

Nyt voitaisiin määrittellä p: PERSON ja kirjoittaa lause !!p.make("Brown", "New York", 1960), jolla luodaan (!! tarkoittaa aina olion luontia) uusi PERSON-tyyppinen olio p eli luodaan luokan PERSON instanssi kohdistamalla p:hen operaatio make, ja alustetaan kyseinen olio parametreina annetuilla tiedoilla.

**Esimerkki.** Määritellään (ohjelmakoodia vaille) luokka pino Eiffelillä. Edellä olevan pino-määrittelyn lisäksi tässä määrittelyssä on mukana myös operaatiot (funktiot), joilla voidaan testata onko pino tyhjä tai onko pino täynnä. Määrittelyssä on edellisen pinoesimerkin tapaan julkinen (kirjoitetaan Eiffelissä: feature {ANY}) ja yksityinen (feature {NONE}) osa.

```

class PINO[ElementType]
creation
  make
feature {NONE}      -- vastaa määrittelyä PRIVATE, eli seuraaviin kahteen määrittelyyn (alkiot ja
                    -- pinon_max_koko) ei saa viitata luokan ulkopuolelta
  alkiot: ARRAY[ElementType] -- tallennusrakenteena vektori, jonka alkiot ovat tyyppiä
                    -- ElementType.
                    -- kukaan ei voi suoraan viitata tallennusrakenteeseen
  pinon_max_koko: INTEGER
feature {ANY}      -- vastaa edellä ollutta määrittelyä PUBLIC, eli seuraavia operaatiota saavat
                    -- kaikki käyttää
  make(n: INTEGER) is -- pinon alustus
    do               -- pinoon voidaan tallentaa enintään n alkioita
      ...
      pinon_max_koko:=n
    end -- make
  push(alkio: ElementType) is -- proseduuri, joka tallentaa alkion pinoon
    do
      ...
    end -- push
  pop: ElementType is      -- funktio, joka palauttaa pinon päällimmäisen alkion
    do
      ...
    end -- pop
  is_empty: BOOLEAN is    -- funktio, joka palauttaa arvon true, jos jono on tyhjä
    do
      Result:=(pinon_koko=0) -- funktion tulos (tallennetaan. aina 'muuttujaan' Result)
    end -- is_empty
  is_full: BOOLEAN is     -- funktio, joka palauttaa arvon true, jos jono on täynnä
    do
      Result:=(pinon_koko=pinon_max_koko)
    end -- is_full
  pinon_koko: INTEGER     -- pinossa olevien alkioiden määrä
end -- class Pino

```

Kun tätä määrittelyä verrataan aiempaan tietotyypin pino määrittelyyn, niin suurin ero on pino-operaatioiden parametreissa. Nyt itse pino ei ole lainkaan parametrina, miksi? Tämä johtuu siitä, että nämä operaatiot kohdistetaan johonkin luotuun pino-olioon. Esimerkiksi määritellään p: Pino[Integer] (pino, jonka alkuina on kokonaislukuja) ja luodaan tyhjä pino, johon voidaan tallentaa enintään 100 kokonaislukua: !!p.make(100). Silloin luvun 2 lisäys pinoon tapahtuu käskyllä p.push(2) eli pinoon p kohdistetaan operaatio push, jonka parametrina on luku 2.

Luokille on ominaista, että ne voivat muodostaa hierarkkisia rakenteita *periytyminen* avulla. Periytymisellä tarkoitetaan yliluokan (superclass) olioiden käytöksen (so. metodien ja attribuuttien) siirtymistä osaksi aliluokan (subclass) olioiden käytöstä. Aliluokassa voidaan uudelleen määrittellä (eli ylikirjoittaa) yliluokan attribuutteja ja metodeja ja toisaalta aliluokassa voidaan määrittellä aliluokalle spesifisiä uusia attribuutteja ja metodeja. Esimerkiksi meillä voisi olla luokka Eläin ja sen aliluokka Lintu, joka perisi kaikki luokan Eläin attribuutit ja metodit, mutta sisältäisi myös uusia ominaisuuksia, esim. attribuutin siipien\_välin\_pituus ja metodin lennä.

Miten sitten oliokeskeinen ohjelmointi eroaa perinteellisestä algoritmikeskeisestä ohjelmoinnista? Perinteisen ohjelmointiajattelun lähtökohtana on algoritmi, jonka

tarvitsemat tietorakenteet on usein otettu mukaan 'tarpeen mukaan' pyrittäessä kohti haluttua päämäärää. Voidaan siis sanoa, että lähtökohtana ovat passiiviset tiedot, joita käsittelevät aktiiviset toimenpiteet (algoritmit). Oliokeskeisessä ajattelussa tietobjekteja ajatellaan aina myös niihin kohdistettavien metodien kautta. Hiukan liioitellusti voidaan sanoa, että oliot ovat aktiivisia, jolloin kokonaisuutta 'tarkkaileva' algoritmi ei ole enää keskeinen. Oliokeskeinen ohjelmointikieli ei ole pelkästään ohjelmointiväline, vaan mitä suurimmassa määrin myös suunnitteluväline. Nimittäin annettu tehtävä ja varsinkin ohjelman käsittelemät tiedot tulee ennen ohjelmoinnin alkamista tarkasti jäsentää. Oliiohjelmoinnin keskeisin – ja myös vaativin tehtävä – onkin luokkarakenteen suunnittelu ennen ensimmäisensä algoritmin toteutusta. Työ aloitetaan suorittamalla edellä esitetty oliokeskeinen analyysi. Oliopohjaiset järjestelmät tarjoavat useita valmiita luokkia (esim. listojen, pinojen, jonojen, puiden, ...) käsittelyyn. Niinpä ensin onkin syytä tutustua huolellisesti järjestelmän tarjoamiin luokkiin, *palveluihin*.

## 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 hyvyttä 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 algoritmisen 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*<sup>4</sup> (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 algoritmisen 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 tehtävälle 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

<sup>4</sup> 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!<sup>5</sup>

### 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.  $\lambda$ -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 määritellyillä algoritmeilla, se voidaan ratkaista jokaisella muulla tavalla määritellyillä algoritmeilla. 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 hyväksytyt, 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<sup>6</sup>.

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

---

<sup>5</sup> Tietokoneiden historian lasketaan yleensä alkaneen vasta toisen maailmansodan loppuvaiheissa.

<sup>6</sup> 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 ikuisen 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 vasta oletus 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 syöteenä P:n ohjelmakoodi (so. tekstitiedosto). Tässä ei ole mitään kummallista, onhan olemassa lukuisia ohjelmia, jotka saavat syöteenä 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 tähän 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 esitellään kolme ongelmaa, jotka eivät ole laskettavissa. Todistukset perustuvat siihen, että pysähtymisongelma ei ole laskettavissa. Yleinen todistusstrategia on seuraava: tehdään vasta oletus, 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 Yhteenvedo 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 todistamaan, että tällaista algoritmia ei ole olemassa.

## 3.2 Kompleksisuus

Parhaimmallaan 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).

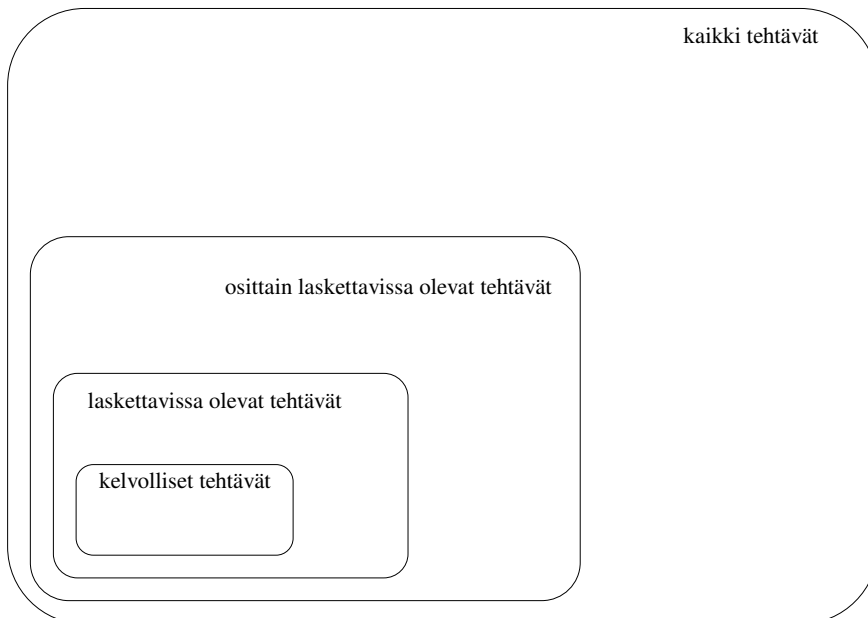
**Esimerkki.** Suurimman yhteisen tekijän määräämiseen esitettiin iteratiivinen algoritmi kappaleessa 2.6 ja rekursiivinen kappaleessa 2.7. Esitystapaeroista huolimatta moduulit määräävät suurimman yhteisen tekijän laskennallisesti tarkalleen samalla tavalla – vähennyslaskun avulla – ja ovat näin muodoin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

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

```
(* 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  $\text{syt}(75,6)$  etenee seuraavasti:  $\text{syt}(75,6) = \text{syt}(6,3) = \text{syt}(3,0) = 3$ . 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 tehokkaasti (kuten yleensä on laita).

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



Tärkeimmät tietokoneressit ovat **aika**, **muisti (tila)** ja **laitteisto**. Aika tarkoittaa algoritmin suoritukseen kuluvaan aikaan, 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 resurssi-tarpeisiin. 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 ohjauksjärjestelmissä 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 alkioden 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  $n$ :n asteen polynomi on muotoa

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

Polynomi identifoidaan kertoimilla  $a_i, i=0,1,\dots,n$ , 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 xi 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 näytävät 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  $T_1(n) = 2+3+\dots+(n+1) = n(n+3)/2$  kertolaskua<sup>7</sup> ja  $T_2(n) = n$  yhteenlaskua.

<sup>7</sup> Lausekkeen arvo saadaan suoraan taulukkokirjasta – kyseessä on aritmeettisen sarjan summa.

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

```

MODULE P( $a_0, \dots, a_n, x$ ) RETURNS  $P_n(x)$ 
  summa :=  $a_0$ 
  xpot := 1
  FOR  $i := 1, \dots, n$  DO
    xpot := xpot *  $x$ 
    summa := summa +  $a_i$  * 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  $i := 2, \dots, A.length$  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 suoritusaikalausekkeitä tarkastellaan yleensä vain niitä lausekkeiden osia, jotka suurilla  $n$ :n arvoilla dominoivat lauseketta. Dominoiva tekijä määrää täysin algoritmin asymptoottisen käyttäytymisen. Näin lausekkeitä 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 kaksinkertaistuksessa toisen asteen polynomin arvo nelinkertaistuu, ja  $n$ :n kymmenkertaistuksessa 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  $\sim$ . Seuraavassa luetellaan tärkeimpiä nimityksiä algoritmien aikakompleksisuudelle (huonommuusjärjestyksessä). Olkoon vakio  $c > 1$ . Algoritmi on

<b>eksponentiaalinen,</b>	jos $T(n) \sim c^n$ ,
<b>polynomiaalinen,</b>	jos $T(n) \sim n^c$
<b>lineaarinen,</b>	jos $T(n) \sim n$ ,
<b>logaritminen,</b>	jos $T(n) \sim \log_c n$ ,
<b>vakioaikainen,</b>	jos $T(n) \sim 1$ (aikakompleksisuus ei riipu syöteen 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  $T(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n^1 + a_0$ , 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 alkoiden määrä. Lajittelutehtävässä keskeinen alkeisoperaatio on kahden alkion vertailu niiden keskinäisen järjestyksen määrittämiseksi. Parittaisten vertailujen lisäksi tässä algoritmissa tehdään alkoiden 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  $(n-1)+(n-2)+\dots+1 = n(n-1)/2$  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.

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

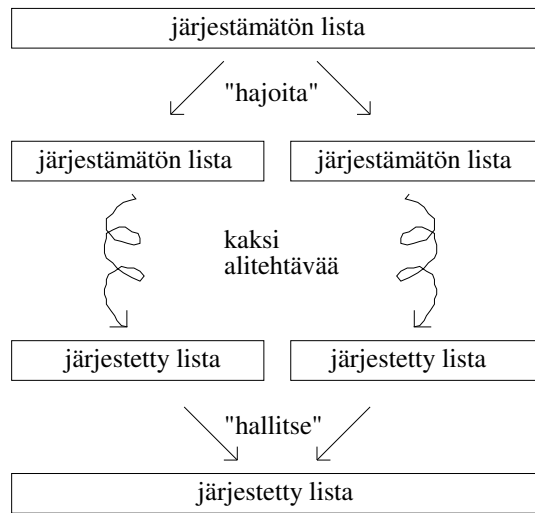
Polynomiaaliset algoritmit ovat enimmäkseen käyttökelpoisia, varsinkin jos eksponentti on pieni. Polynomiaalisilla algoritmeilla käyttökelpoisuuden raja tulee vastaan vasta suurilla  $n$ :n arvoilla. Lineaariset algoritmit ovat nopeita myös suurilla  $n$ :n arvoilla, logaritmisten algoritmien suoritusnopeuteen 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.

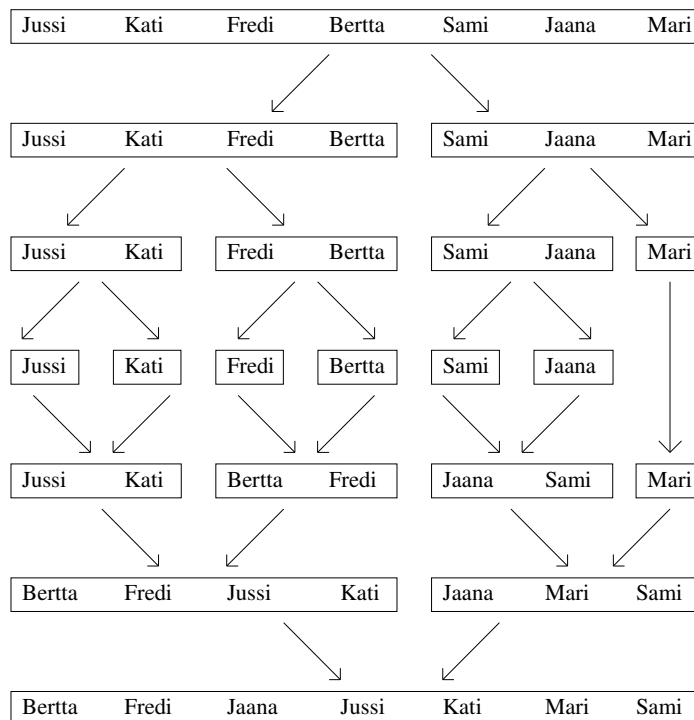
Edellä saatiin vaihtolajittelun asymptoottiseksi aikakompleksisuudeksi  $n^2$ , mutta lajittelutehtävän voi ratkaista tehokkaamminkin, reduktiivisesti hajota ja hallitse -periaatetta käyttäen. Nimilista voidaan lajitella jakamalla nimilista kahtia, lajittelemalla kumpikin puolisko jollakin tavalla, ja 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
  (* Limitsmoduulin 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 alkoiden kokonaismäärään nähden lineaarisessa ajassa. Merkitään  $n:n$  alkion lajittelussa tarvittavien parittaisten vertailujen määrää  $T(n)$ :llä. Silloin

$$\begin{aligned}
 T(n) &= T(n/2) && \text{listan alkuosan rekursiivinen lajittelu} \\
 &+ T(n/2) && \text{listan loppuosan rekursiivinen lajittelu} \\
 &+ n && \text{limitys} \\
 &= 2T(n/2) + n
 \end{aligned}$$

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

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

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

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

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 puolisoista 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 selvittää noin kymmenessä tunnissa! Limityslajittelu onkin asympotoottisesti 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:

$$\begin{aligned}
 T(n) &= T(n-1) + 1 \\
 &= T(n-2) + 1 + 1 \\
 &= T(n-3) + 1 + 1 + 1 \\
 &= \dots \\
 &= T(0) + n \\
 &= n,
 \end{aligned}$$

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

$$\begin{aligned}
 T(n) &= 1 && \text{jakolasku } n//2 \text{ (rinnastuu vaativuudeltaan kertolaskuun)} \\
 &+ T(n/2) && \text{rekursiivinen kutsu} \\
 &+ 1 && \text{kertolasku } m*m \\
 &= T(n/2) + 2 \\
 &= T(n/4) + 2 + 2 \\
 &= T(n/2^2) + 2*2 \\
 &= T(n/2^3) + 2*3 \\
 &= \dots \\
 &= T(n/2^m) + 2m
 \end{aligned}$$

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

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

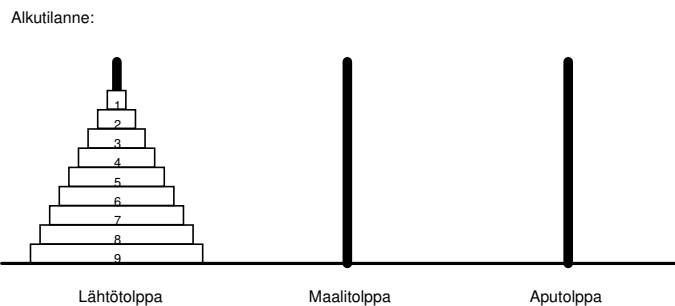
pohjalta kirjoitetusta algoritmista tulee noin kolme kertaa algoritmia  $\exp$  hitaampi!

### 3.2.3 Hanoin tornien ongelma

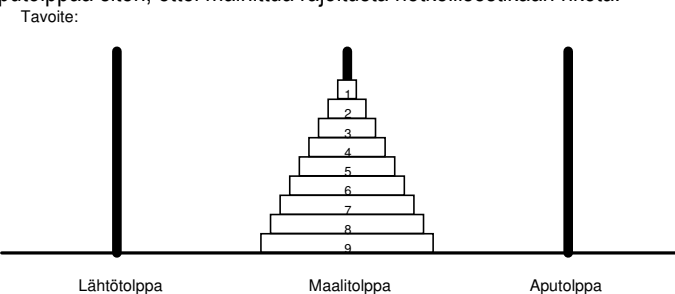
Seuraavaksi esitetään esimerkki, joka ratkeaa luonnollisesti käyttäen rekursiota ja johon tehtävän luonteen vuoksi on hankalaa keksiä iteratiivista algoritmia.

**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 erisuuria kivi-kiekkoa 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 apuolppaan 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 ulottuvuuksistaan 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:



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



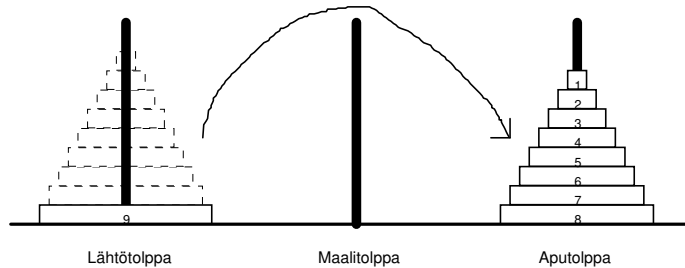
Tehtävä on vaikea. Se on vaikea siinäkin mielessä, että ratkaisualgoritmi saattaa aluksi tuntua vaikealta keksiä, mutta etenkin siinä mielessä, että ratkaisun tuottaminen on hyvin monimutkaista. Jos kiekkojen määrä on vähänkin suurehko, niin siirtojen määrä kasvaa niin suureksi, että yksittäisen siirron 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:

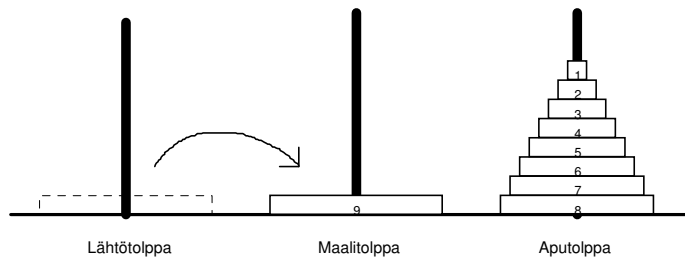
1. Siirrä tavalla tai toisella 63 päällimmäistä kiekkoa lähtötolpasta aputilppaan.
2. Siirrä alimmainen kiekko lähtötolpasta maalitolppaan.
3. Siirrä aputilppaan siirretyt 63 kiekkoa maalitolppaan.

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

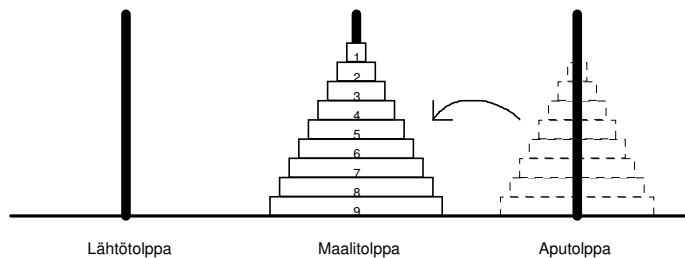
Osa 1:



Osa 2:



Osa 3:



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, vähenehän 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, siinä 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 tolppasta 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 tolppasta 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 tolppaa (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 tolppasta 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 tolppasta 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 keskimäinen lause eli yhden kiekon siirto (osa 2). Seuraavaksi suoritetaan jälleen 8 kiekon tehtävä: siirretään tolppaa 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 Tolppasta 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:

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

Koska yhden kiekon ongelman ratkaisemiseksi tarvitaan yksi siirto,  $T(1) = 1$ , joten saadaan edelleen<sup>8</sup>

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

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  $T(64) = 2^{64} - 1 \approx 585$  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 kelvollisuuskään 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.

<sup>8</sup> Geometrisen sarjan summa löytyy taulukkokirjasta. Se voidaan myös ajatella seuraavasti:  $2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0$  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.
- *Pakkausongelma* (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.
- *Lukujärjestysongelma* (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 rajoittamalla 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 lukujärjestysongelma 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 algoritmista 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.

Yhteenvedona 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, oikeellisuustyyppistä ja menetelmiä algoritmin toimivuuden ja oikeellisuuden osoittamiseen.

Useimmissa ohjelmissa on virheitä (bugs). Jopa sellaiset ohjelmat, jotka ovat olleet vuosikautia käytössä, voivat tietyissä olosuhteissa toimia väärin. Yleensä suurin osa ohjelmoijan työajasta kuluu 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 tehtävnmäärittely. Parannus: mahdollisimman luonnollinen ja tarkasti jäsenelty tehtävnmäärittely. 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 ensimmäistään 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 määriykset: 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:  $n \geq 0$ . 2) Taulukon  $T$  alkuiden keskiarvon palauttavan moduulin alkuehto on muotoa:  $T.n \text{ pituus} > 0$  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 vakuuttuneisuutta 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:  $1+2+\dots+n=n(n+1)/2$ ,  $n \geq 1$  (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  $P(0), P(1), \dots$

**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  $n=0,1,2,\dots$ . 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  $n=0, 1, 2, \dots$

*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 'pienien' 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 tiettyinä 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 toistorakenteiden 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.

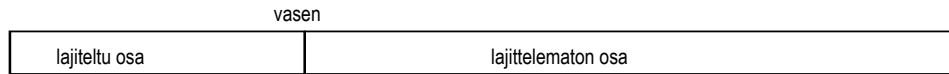
**Esimerkki.** Vaihtolajittelun (2.8.2.2) oikeellisuus. Kun vaihtolajittelualgoritmi kirjoitetaan oikeellisuustodistusta silmällä pitäen käyttäen havainnollisia muuttujan nimiä, käy menetelmän toimivuus ilmeiseksi. Näin oikeellisuuden todistamisen periaatteiden tuntemus myötävaikuttaa algoritmin laatuun 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 alkioita. Osoitetaan, että tämä väite on voimassa ulomman silmukan lopussa silmukkalaskurin vasen arvoilla 1, 2, ..., pituus-1. Olkoon tämä väite nimeltään  $P(\text{vasen})$ . Väitettä kutsutaan ns. silmukkainvariantiksi.



1. Alkutilassa vektorin alkioiden parittaisia vertailuja ei ole suoritettu lainkaan, joten väite on triviaalisti voimassa alkutilassa. Oletetaan, että väite pitää paikkansa lopputilassa. Tällöin  $\text{vasen} = \text{pituus} - 1$ , joten vektorin  $V$  alkioita  $V(1), V(2), \dots, V(\text{pituus} - 1)$  ovat järjestyksessä ja alkio  $V(\text{pituus})$  on vähintään kaikkien muiden alkioiden suurin. Täten vektori  $V$  on järjestetty ja algoritmi toimii oikein.
2. Osoitetaan väitteen voimassaolo induktiolla. Kun  $\text{vasen} = 1$ , niin vektorin ensimmäistä komponenttia verrataan vektorin komponentteihin 2, 3, ...,  $\text{pituus}$  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  $V(k) \geq V(k-1) \geq \dots \geq V(1)$  ja  $V(j) \geq V(i)$  kaikilla  $j = k+1, k+2, \dots, \text{pituus}$  ja  $i = 1, 2, \dots, k$ . Kun nyt ulompi silmukka suoritetaan silmukkalaskurin vasen arvolla  $k+1$ , tulee algoritmin nojalla paikkaan  $V(k+1)$  pienin alkioista  $V(k+1), \dots, V(\text{pituus})$ . Induktio-oletuksen nojalla taas  $V(k+1) \geq V(i)$ ,  $i = 1, 2, \dots, k$ , joten  $V(k+1) \geq V(k) \geq \dots \geq V(1)$ . Näin ollen vektorin  $k+1$  ensimmäistä alkioita 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. Ensimmäin 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 päätyminen 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 päätyminen 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  $g(x,y) = x^{x^{\dots^x}}$ , 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 määrittäminen 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 aritmeettisiä 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 liikenne-merkeissä, 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 kuin 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 kirjoitusmerkillä. 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 numero-merkeistä 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-kirjoitettavia 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.

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

**Määritelmä.**  $k$ -järjestelmän esitys

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

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

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

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

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

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

$$(a_{n-1}a_{n-2}\dots a_1a_0 \cdot a_{-1}a_{-2}\dots a_{-(m-1)}a_{-m})_k$$

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

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.**

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

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

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

$$3610_{10} = 451 \cdot 8 + 2 = (56 \cdot 8 + 3) \cdot 8 + 2 = ((7 \cdot 8 + 0) \cdot 8 + 3) \cdot 8 + 2 = 7032_8$$

$$374_{11} = 3 \cdot 11^2 + 7 \cdot 11 + 4 = (3 \cdot 11 + 7) \cdot 11 + 4 = 444_{10}$$

$$= 111 \cdot 4 + 0 = (27 \cdot 4 + 3) \cdot 4 + 0 = ((6 \cdot 4 + 3) \cdot 4 + 3) \cdot 4 + 0 = (((1 \cdot 4 + 2) \cdot 4 + 3) \cdot 4 + 3) \cdot 4 + 0$$

$$= 12330_4$$

Desimaaliluvun muunnos 10-järjestelmästä  $k$ -järjestelmään voidaan suorittaa seuraavaa menetelmää käyttäen: Olkoon luvun kokonaisosa  $x = (a_{n-1}a_{n-2}\dots a_1a_0)_{10}$  ja desimaaliosa  $y = (0.a_{-1}a_{-2}\dots a_{-(m-1)}a_{-m})_{10}$ , missä siis  $0 \leq y < 1$ . Jaetaan  $x$  toistuvasti  $k$ :lla (ottamalla osamäärän kokonaisosa aina uudeksi jaettavaksi), kunnes jaettava tulee nolllaksi. 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 nolllaksi.

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

Kokonaisosa:	$\lfloor 123 / 4 \rfloor = 30$ , jakojäännös = <b>3</b>
	$\lfloor 30 / 4 \rfloor = 7$ , jakojäännös = <b>2</b>
	$\lfloor 7 / 4 \rfloor = 1$ , jakojäännös = <b>3</b>
	$\lfloor 1 / 4 \rfloor = 0$ , jakojäännös = <b>1</b>
Desimaaliosa:	$0.703125 \cdot 4 = 2.8125$
	$0.8125 \cdot 4 = 3.25$
	$0.25 \cdot 4 = 1.00$
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'<sup>9</sup>. 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  $-2^{n-1}, \dots, 0, \dots, 2^{n-1} - 1$ .

<sup>9</sup> 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 puhtaana 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  $-5$ :

$$\begin{array}{r} -2 = 1010 \\ +3 = 0011 \\ \hline 1101 = -5 \end{array}$$

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 positiivisilläkin luvuilla. Tarkastellaan äskeitä esimerkkilaskua uudelleen, nyt yhden komplementtiesitystä käyttäen:

$$\begin{array}{r} -2 = 1101 \\ +3 = 0011 \\ \hline 0000 = (+)0 \end{array}$$

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

$$\begin{array}{r} -2 = 1110 \\ +3 = 0011 \\ \hline 0001 = +1 \end{array}$$

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  $(m_{n-1}m_{n-2}\dots m_0)_2$ . Silloin

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

ja

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

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:

$$\begin{array}{r} +4 = 0100 \\ +5 = 0101 \\ \hline 1001 = -7 \end{array}$$

Jos  $n = 16$ , voidaan esittää kokonaisluvut  $-32\ 768 \dots +32\ 767$ . Jos  $n = 32$ , voidaan esittää kokonaisluvut  $-2\ 147\ 483\ 648 \dots +2\ 147\ 483\ 647$ . 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  $x \equiv y \pmod{m}$ , jos  $x$  voidaan kirjoittaa muodossa  $x = y + km$ , missä  $k$  on kokonaisluku. Esimerkiksi  $-7 \equiv 9 \pmod{16}$ . Kahden komplementtiesityksessä lukusuora onkin siis lukusykli: Laskutoimitukset tapahtuvat esityksen pituuden  $n$  määräämän lukualueen puitteissa. Toisistaan  $k \cdot 2^n$  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  $-a\dots b$  kokonaan positiiviselle puolelle lisäämällä *kaikkiin* lukuihin  $a$ . Jos esimerkiksi  $n=4$  ja lukualue  $-8\dots 7$ , 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.

$$10.11 = 0.1011 \cdot 2^2, \text{ joten mantissa} = 1011_2 \text{ ja eksponentti} = 2_{10}$$

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

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  $-128 \dots 127$ )
  - mantissan itseisarvo: 23 bittiä
2. Pitkät liukuluvut: 8 tavua = 64 bittiä:
  - merkkibitti
  - eksponentti: 11 bittiä, koodaus excess-1024 (eksponentin arvoalue  $-1024 \dots 1023$ )
  - mantissan itseisarvo: 52 bittiä

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

### 4.1.4 Boolean algebra

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

ollen 'piirejä voidaan manipuloida matematiikan avulla'. Sen vuoksi seuraavaksi määritellään Boolean algebra ja sen laskulait.

Kahden alkion, yhteen- ja kertolaskun sekä ns. komplementtioperaation muodostamaa matemaattista struktuuria  $B = \langle \{0,1\}, +, \times, \bar{\phantom{x}} \rangle$  sanotaan Boolean algebraksi. Sen avulla mallinnetaan laskutoimituksia kaksijärjestelmässä. Boolean algebra eroaa tavallisesta lukujen algebrasta alkioden rajoitetun lukumäärän suhteen, mikä aiheuttaa eroja myös operaatioiden käyttäytymisessä. Määritelmän mukaan Boolean algebran ainoat alkiot ovat 0 ja 1. Alkioden välille on määritelty binääriset operaatiot, joita merkitään merkeillä + ja  $\times$ . 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  $\times$  presedenssi on korkeampi kuin operaation +. Esimerkiksi lausekkeessa  $x+x \times y$  (eli  $x+xy$  kuten seuraavassa lyhyesti merkitään) lasketaan ensin  $xy$ . Huomaa, että operaatioilla + ja  $\times$  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 Boolean algebran laskulait, joita innokas lukija voi verrata tavanomaisen algebran vastaaviin lakeihin. Jatkossa jätetään, kuten on tapana, operaattori  $\times$  merkitsemättä, paitsi numerosymbolin vieressä.

### Boolean algebran laskulait:

- $0+0 = 0, 0+1 = 1+0 = 1+1 = 1; \quad 0 \times 0 = 0 \times 1 = 1 \times 0 = 0, 1 \times 1 = 1; \quad \bar{0} = 1, \bar{1} = 0.$
- 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:  $x+0 = x, x+1 = 1; \quad x \times 0 = 0, x \times 1 = x; \quad x + \bar{x} = 1, x\bar{x} = 0$
- 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 Boolean kerto- ja yhteenlaskuoperaatioiden lisäksi kaksijärjestelmässä toimittaessa usein käytetään **modulo 2 -yhteenlaskua**, joka määritellään seuraavasti:  $0 \oplus 0 = 1 \oplus 1 = 0, 0 \oplus 1 = 1 \oplus 0 = 1$ . 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  $2 \equiv 12 \pmod{10}$ .



Kun 0 tulkitaan epätodeksi (false), 1 todeksi (true),  $\times$  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 Boolean algebran laskulakeja. Boolean 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  $\text{NOT}(X \text{ OR } Y) = (\text{NOT } X) \text{ AND } (\text{NOT } Y)$  ja  $\text{NOT}(X \text{ AND } Y) = (\text{NOT } X) \text{ OR } (\text{NOT } Y)$  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 Boolean algebran laskulakeja loogisten piirien sieventämisessä, jossa neljä viimeistä laskulakia ovat keskeisessä asemassa.

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

## 4.2 Fysikaaliset perusteet

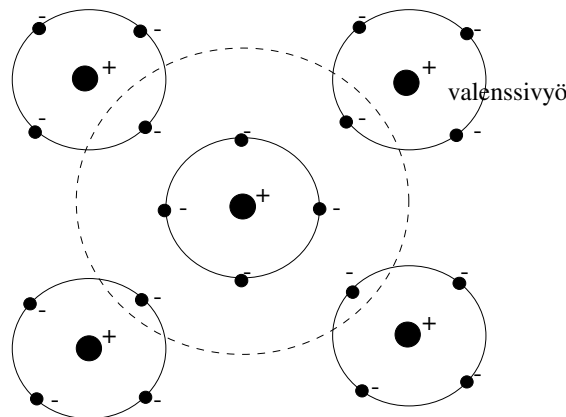
Ensimmäiset tietokoneet olivat täysin mekaanisia, myöhemmin elektromeekaanisia, ja nykyiset koneet ovat täysin elektronisia. Tietokoneiden fysikaaliset perusteet ovatkin puolijohteissa ja digitaalelektronikassa.

### 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. *valenssivöillä* sekä sitä lähempänä ydintä sijaitsevat elektronit ovat sidottuja ytimeen, kun taas ns. *johtavuusvöillä* tai ulompana olevat elektronit voivat siirtyä atomista toiseen. Sidoksissa aineen valenssielektroneita voi nousta johtavuusvölle ja siirtyä sieltä edelleen viereiseen atomiin. Tällöin aine *johtaa sähköä*. Kyseisiä elektroneja sanotaan *negatiiviseksi varauksenkuljettajiksi*. Tällaisia aineita ovat mm. useimmat metallit. Toisten aineiden sidoksissa taas valenssivölle jää vajaa (alle kahdeksan elektronin) miehitys: syntyviä elektroniaukkoja, puuttuvia elektroneja, sanotaan *positiiviseksi 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

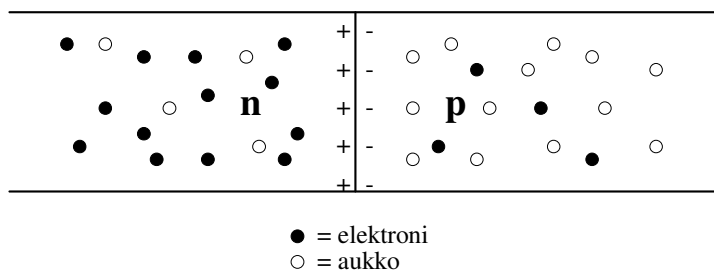
efektiivisesti kahdeksan valenssi-elektronia, joten vapaita elektroneja tai myöskään aukkoja ei ole. Pii ei siis johda sähköä.



Puoliyohteet voidaan **huumata** (dope) vierailta 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-aineksi**. Jos taas piihin lisätään indiumia, booria tai galliumia, joilla on kolme valenssielektronia, syntyy rakenteeseen vapaita aukkoja, jotka toimivat positiivisina varauksenkuljettajina. Tällaista ainetta sanotaan **p-aineksi**. Huomaa, että niin n- kuin p-tyypinkin puoliyohteet ovat kuitenkin kokonaisvarauksettomia.

## 4.2.2 Puoliyohdekomponentit

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



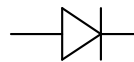
Diffuusion vuoksi vapaita elektroneja pyrkivät n-ainesta, missä niitä on enemmän, p-aineseen, missä niitä on vähemmän. Aukot pyrkivät vastaavasti p-ainesta n-aineseen. Diffuusion seurauksena liitoksen rajapinta varautuu sähköisesti siten, että alun perin varaukseton n-aine muuttuu saamiensa aukkojen 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? Kytetään jännitelähde ensin **myötäsuntaan**: plusnapa p-aineseen ja miinusnapa n-aineseen. Silloin varaus purkautuu, sähkökenttä häviää ja diffuusio vahvistuu. Plusnapa vetää puoleensa

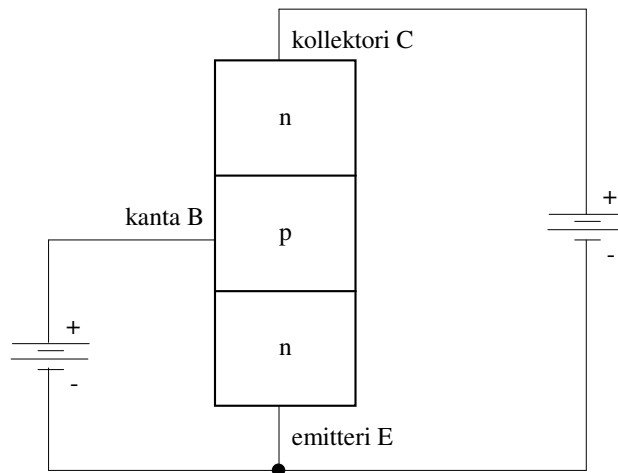
n-aineen ylimääräisiä elektroneja ja miinusnapa puolestaan imee p-aineen vapaat aukot. Vastaavasti plusnapa tuottaa p-aineeseen lisää aukkoja sieltä poistuneiden tilalle ja miinusnapa puolestaan tuottaa n-aineeseen lisää elektroneja sieltä poistuneiden tilalle. Diffuusio jatkuu niin kauan kuin ulkoinen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



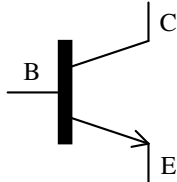
Kahdesta peräkkäisestä np-liitoksesta puolestaan muodostuu **transistori**, jolla on kolme kytkentäkohtaa: kollektori (C), emitteri (E) ja kanta (B = base). Ns. bipolaarisen npn-transistorin rakenne on seuraavanlainen:



Transistorin toiminnan idea on, että kollektorin ja emitterin välisessä virtapiirissä kulkevaa virtaa voidaan säädellä kannan ja emitterin välisellä virralla. Tietyillä signaali-tasoilla sopivasti rakennettu transistori toimii virtavahvistimena, mutta digitaalelektronikassa ollaan enemmän kiinnostuneita transistorin *kytkentäominaisuuksista*: mitä tapahtuu, kun signaali muuttuu nopeasti korkeasta (merkitään sitä ykkösellä) matalaksi (merkitään nolllalla) 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.

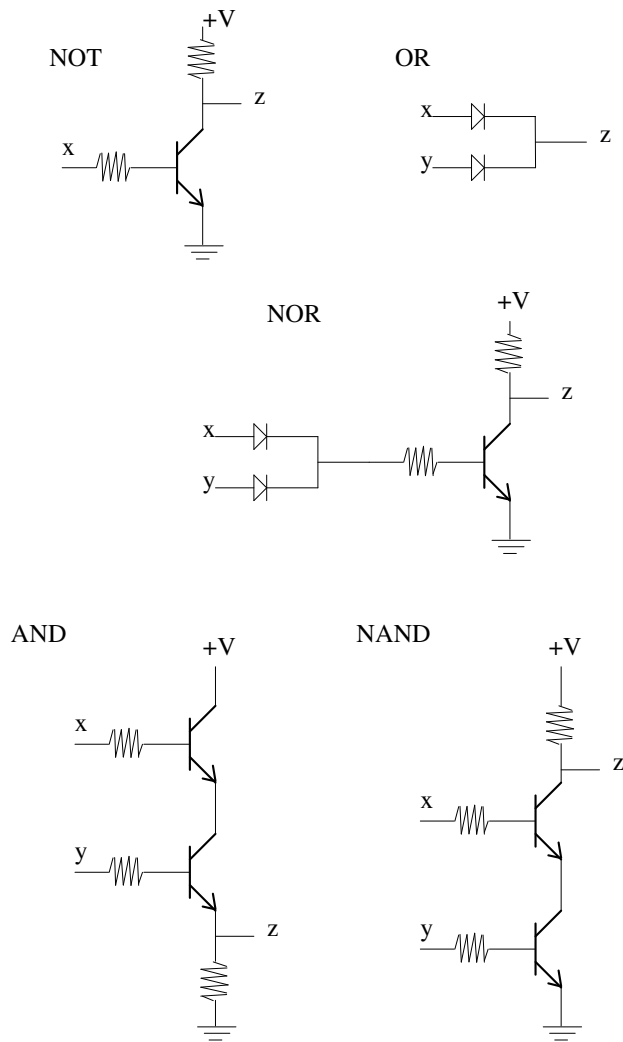
Transistorin toimintaa pyritään lisäksi edistämään erilaisilla rakenteellisilla ratkaisuilla. Esimerkiksi kanta tehdään käytännössä mahdollisimman ohueksi, jotta mahdollisimman suuri osa emitteriltä emittoituvista elektroneista päätyisi kollektorille. Näin kantavirta minimoituu. Aikaisemmin transistoria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

Nykyään transistoreita tehdään monilla tavoilla. Yleisiä ovat ns. kanavatransistorit (field effect transistor, FET), jotka jänniteohjattuina kuluttavat vähemmän tehoa kuin virtaohjatut 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:



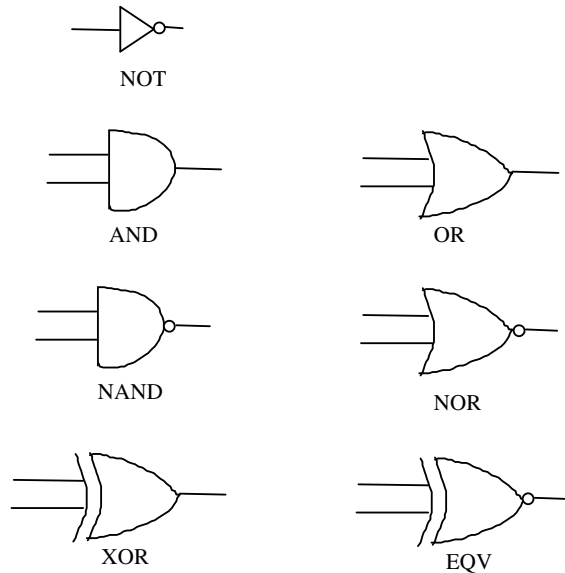
Todellisuudessa loogisten porttien toteutus on toki monimutkaisempi: niihin kuuluu mm. erilaisia puskuriasteita, joiden tarkoitus on saada portit toimimaan tarkoitetulla tavalla ympäristöstä riippumatta. Porttien toteutukseen voidaan myös käyttää erilaisia tekniikoita, logiikoita. Tässä yhteydessä tärkeintä 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 Boolean lausekkeiden syntaktisena varianttina, vaihtoehtoisena esitysmuotona. Oleellista on, että lausekkeet ovat toteutettavissa puolijohde-elektronikalla. Porttien toteuttamat totuusfunktiot voidaan määrittellä *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 Boolean lausekkeet:

x	NOT x $\bar{x}$
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

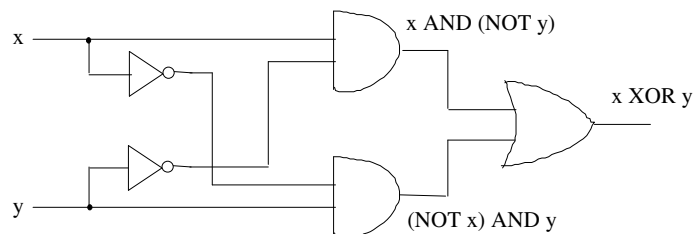
Loogisista veräjistä käytettävät piirrossymbolit ovat:



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 Boolean lausekkeina, joita voidaan sieventää käyttämällä Boolean algebran laskulakeja. Sieventämisen tarkoitus on ilmeinen: matematiikkaa käyttäen voidaan tarvittavan ‘raudan’ määrää vähentää. Sieventämisestä esitetään seuraavassa esimerkkejä.

Esitettyä vähemmälläkin määrällä perusportteja tullaan toimeen. Esimerkiksi pois-sulkeva 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.

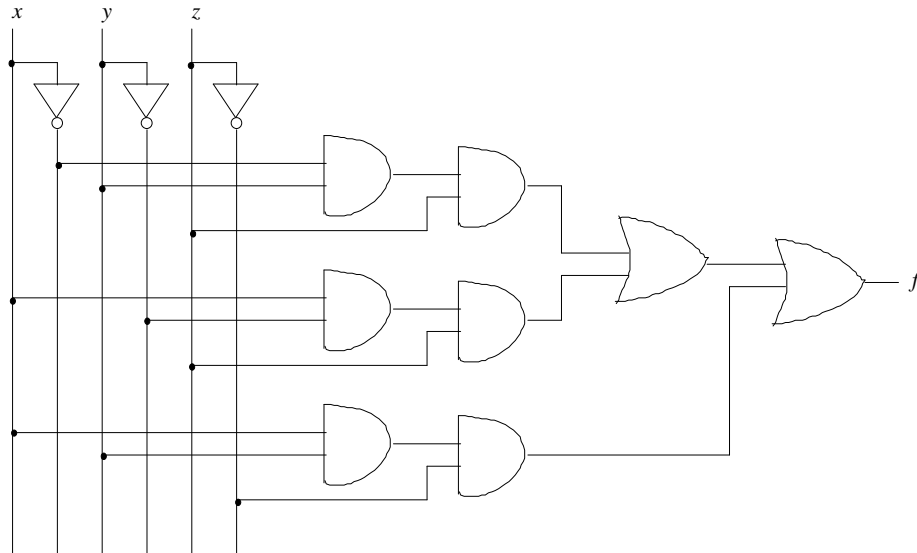
**Esimerkki.** Muodostettava looginen piiri, joka toteuttaa funktion  $f = f(x,y,z)$ , jonka totuustaulu on:

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:

$$(\text{NOT } x \text{ AND } y \text{ AND } z) \text{ OR } (x \text{ AND NOT } y \text{ AND } z) \text{ OR } (x \text{ AND } y \text{ AND NOT } z).$$

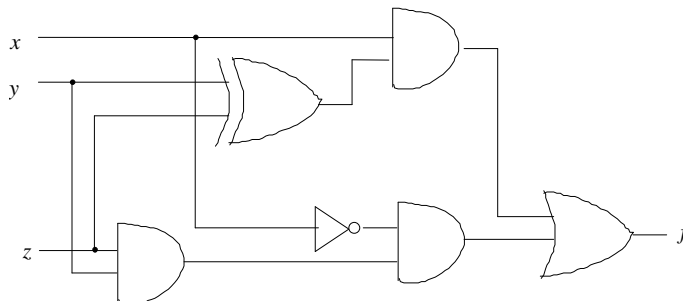
Siis  $f(x,y,z) = \bar{x}yz + x\bar{y}z + xy\bar{z}$ , ja voimme piirtää



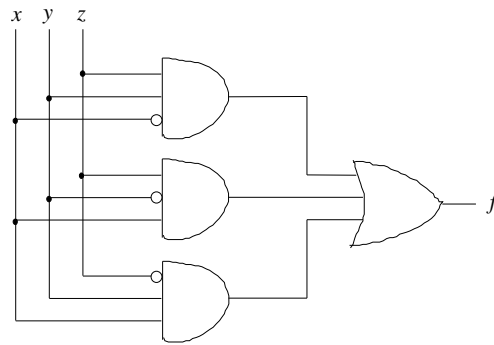
Toisaalta

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

jolloin



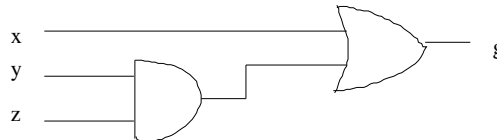
Assosiatiivisten operaatioiden osalta esitystä voidaan vielä yksinkertaistaa jättämällä turhat sulkeet pois ja sallimalla binääriselle portille useamman kuin kaksi syöttöä. Huomaa, että AND- ja OR-portille voidaan sallia useita syötteitä ja portin toiminta on ilmeinen. On myös tapana esittää käännetty syöttö erillisen NOT-veräjän asemesta pienellä ympyrällä vastaavassa syöttölinjassa. Tällaisiin 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  $z$ :stä riippumatta arvon 1, kun  $x = 1$ , ja  $x$ :stä riippumatta arvon 1, kun  $y = z = 1$ . Voidaan siis 'keksiä', että  $g(x, y, z) = x \text{ OR } (y \text{ AND } z) = x + yz$ , 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:

$$\bar{g} = \bar{x} \bar{y} \bar{z} + \bar{x} \bar{y} z + \bar{x} y \bar{z}$$

jolloin

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

ja edelleen sieventämällä

$$\begin{aligned} &= \bar{x} \bar{y} \bar{z} + \bar{x} \bar{y} z + \bar{x} y \bar{z} \\ &= \bar{x} \bar{y} \bar{z} \cdot \bar{x} \bar{y} z \cdot \bar{x} y \bar{z} \\ &= (x + y + z)(x + y + \bar{z})(x + \bar{y} + z) \\ &= x + (y + z)(y + \bar{z})(\bar{y} + z) \\ &= x + (y + z\bar{z})(\bar{y} + z) \\ &= x + y(\bar{y} + z) \\ &= x + y\bar{y} + yz \\ &= x + yz \end{aligned}$$

### 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:

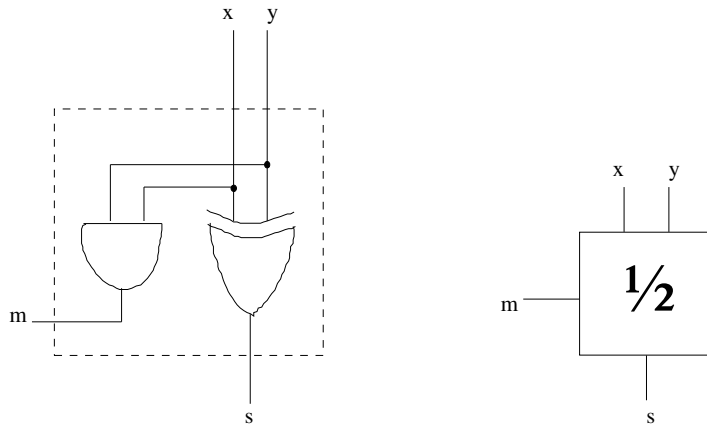
- **Aritmetiikasta** huolehtiva yksikkö. Monisteessa esitettävässä tietokoneessa toteutetaan laitteistollisesti kahden komplementtilukujen yhteen- ja vähennyslasku; kerto- ja jakolasku hoidetaan hyvin matalan tason ohjelmilla (firmware). Korkeamman tason laskenta toteutetaan konekielellä tai korkeamman 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 käsittelemään ohjaavaa tietoa.
- **Loogiset vertailut** mahdollistavat algoritmeissa välttämättömien valintarakenteiden ja dynaamisten toistorakenteiden toteuttamisen hyvin matalalla tasolla. Tarkasteltavat primitiiviset operaatiot ovat kahden komplementtilukujen nolisuuden 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

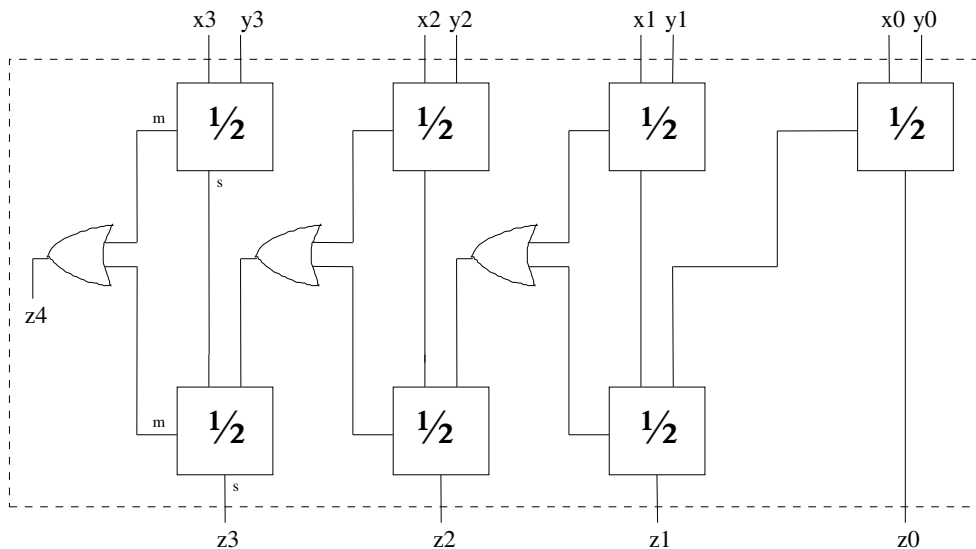
Kahden kahden komplementtiluvun yhteenlasku toteutetaan tavallisella 'kynällä ja paperilla allekkain' -algoritmillä. Kussakin sarakkeessa olevat numerot lasketaan yhteen, tulokseen lisätään mahdollinen muistinumero edellisestä sarakkeesta, ja tulos kirjoitetaan viivan alle. Samalla syntyy mahdollisesti 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:

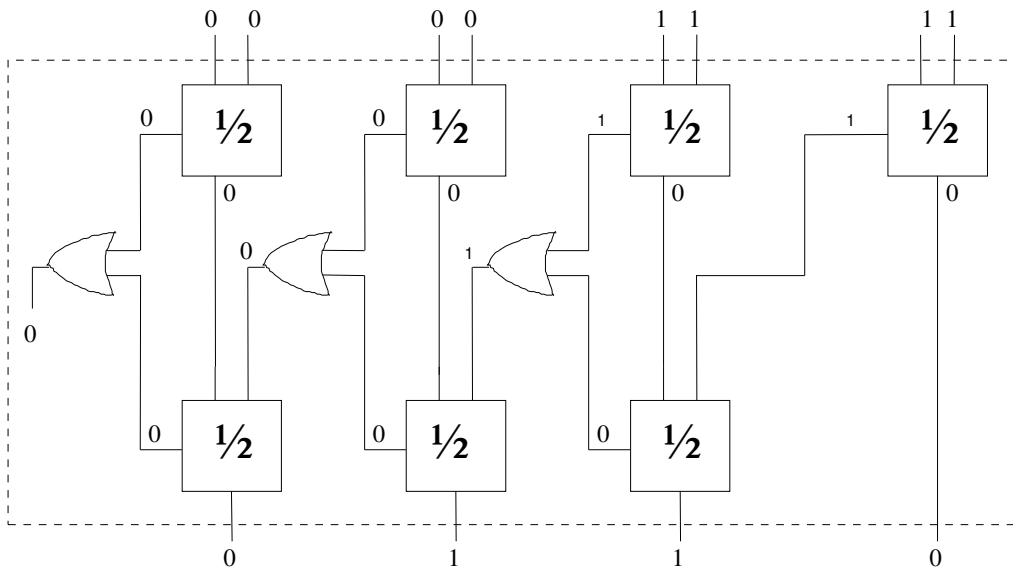


Pidempien lukujen yhteenlasku vaatii useita puolisummaimia. n-bittinen *kokosummain* (full adder) konstruoidaan modulaarisesti  $2n-1$ :stä puolisummaimesta. Esimerkiksi 4-bittinen kokosummain:



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  $-8\dots+7$  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?).

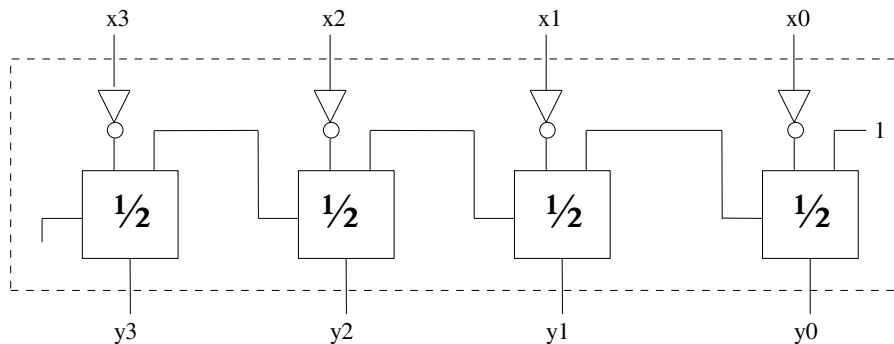
Havainnollistetaan 4-bittisen kokosummainen toimintaa esimerkkisummalla  $0011+0011$ , jossa ei tapahdu ylivuotoa ja summaksi tulee  $0110$  (eli  $3_{10}+3_{10}=6_{10}$ ). Kirjoitetaan arvot 0 ja 1 piirin johtoihin:



Itse asiassa kokosummaina 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 muistinumero 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 kokosummainen avulla, kunhan toinen yhteenlaskettava muunnetaan ensin vastaluvukseen. Vastalukuoperaatio sujuu kahden komplementtilukujen määritelmän mukaan muuttamalla bitit komplementeikseen ja lisäämällä tulokseen ykkösen:



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  $x = 0$ .

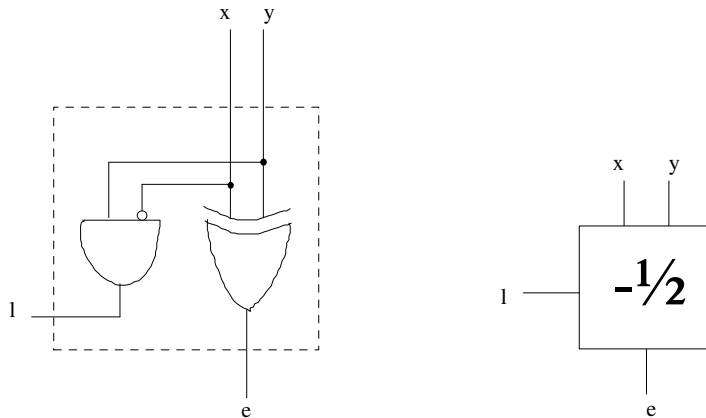
Vähennyslasku voidaan toteuttaa myös suoraan, samalla tavalla kuin yhteenlasku. Vähennyslasku suoritetaan tuttuun tapaan 'kynällä ja paperilla allekkain' -algoritmeilla. Kun nolasta vähennetään ykkönen, täytyy lainata  $10_2$  seuraavasta sarakkeesta. Esimerkiksi:

$$\begin{array}{r} \phantom{0}1010 \\ \phantom{0}1101 \\ - \phantom{0}0111 \\ \hline \phantom{0}0110 \end{array}$$

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

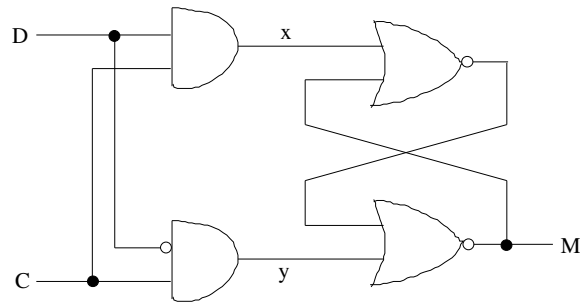
Taulukosta nähdään, että sarake-erotus  $e = x \oplus y$  (samoin kuin yhteenlaskussakin!) ja lainaus  $l = \bar{x}y$ . Puolivähennin ja sen symboli ovat:



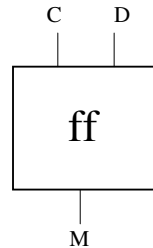
**Kokovähennin**  $n$ -bittisille luvuille konstruoidaan modulaarisesti puolivähentimistä, kuten yhteenlaskulaite. Tämä jätetään harjoitustehtäväksi. Huomaa lainauksen kulkeutuminen!

### 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 tulobitti 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 nolaksi. 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  $y = C\bar{D}$  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  $(x,y) = (D,\bar{D})$  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 johtaja 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:

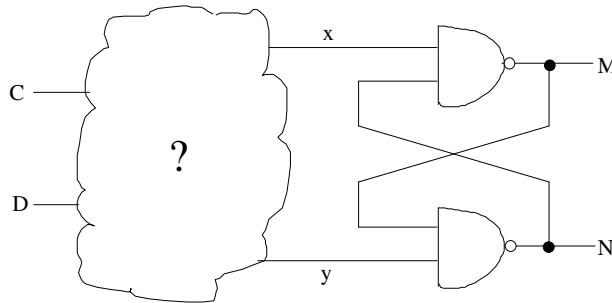
**Tapaus 1: Muista bitti M.** Tällöin tulee asettaa  $C=0$ . Silloin  $x=y=0$  D:stä riippumatta. Tällöin muistiosan ylempään NOR-porttiin menee 0 ja M ja näin ollen ylempään portin ulostulo on  $\overline{M}$ . Tällöin alempaan NOR-porttiin menee syötteenä  $\overline{M}$  ja 0, joten ulos tulee M.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



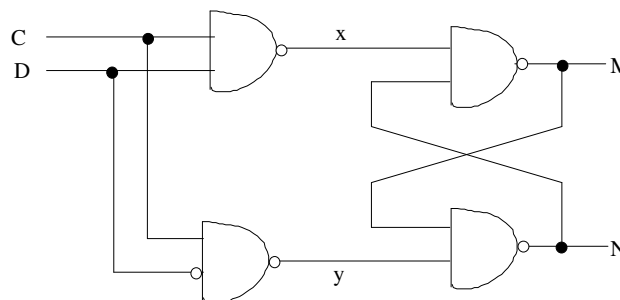
Miten ohjausosa on konstruoitava, jotta muistiosa ja koko piiri toimii halutulla tavalla? Analysoidaan piirin toiminta yksityiskohtaisesti. Ulostulon M rinnalle on merkitty apulinja N. Käydään ensin läpi kaikki mahdolliset  $(x,y,M,N)$ -yhdelmät ja katsotaan miten muistiosa käyttäytyy. Osa tapauksista 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		*	
<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	*		
0	1	0	0	1	1		*	
0	1	0	1	1	1		*	
<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	*		
0	1	1	1	1	0		*	
1	0	0	0	1	1		*	
<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	*		
1	0	1	0	1	1		*	
1	0	1	1	0	1		*	
1	1	0	0	1	1			*
<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	*		
<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	*		
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  $M = \bar{N}$ . Valitaan kiikun ykköstilaksi tila  $(M,N) = (1,0)$  ja nolatilaksi tila  $(M,N) = (0,1)$ , jolloin kiikun muistama bitti = M. Kiikun toiminnan tulee siis olla (merkintä - tarkoittaa 'ei väliä'):

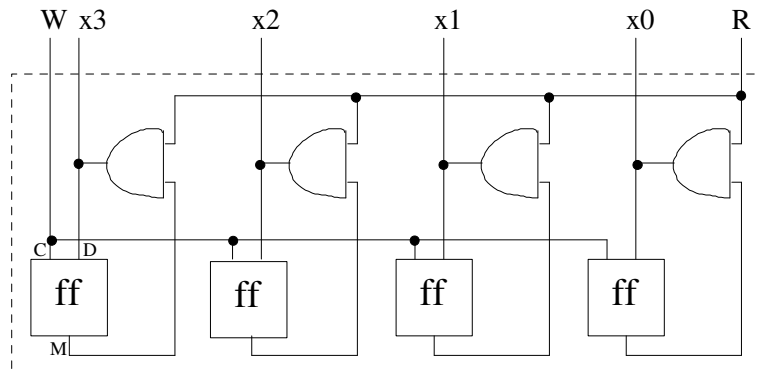
Lepotila	C = 0, D = -:	x = 1, y = 1	$M_t = 0$	$M_{t+1} = 0$
Lepotila	C = 0, D = -:	x = 1, y = 1	$M_t = 1$	$M_{t+1} = 1$
Asetus 0	C = 1, D = 0:	x = 1, y = 0	$M_t = -$	$M_{t+1} = 0$
Asetus 1	C = 1, D = 1:	x = 0, y = 1	$M_t = -$	$M_{t+1} = 1$

Täten  $x=1$ , kun  $C=0$  tai  $C=1$  ja  $D=0$ , siis  $x = \bar{C} + \bar{C} + CD = \bar{C}\bar{D}$ . Vastaavasti  $y=1$  aina paitsi kun  $C=1$  ja  $D=0$ , siis  $y = \bar{C}D$ . Tällainen ohjaus estää myös epästabiilin käytöksen, joka tapahtuu, jos x ja y tulevat yhtäaikaa 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 siinä 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

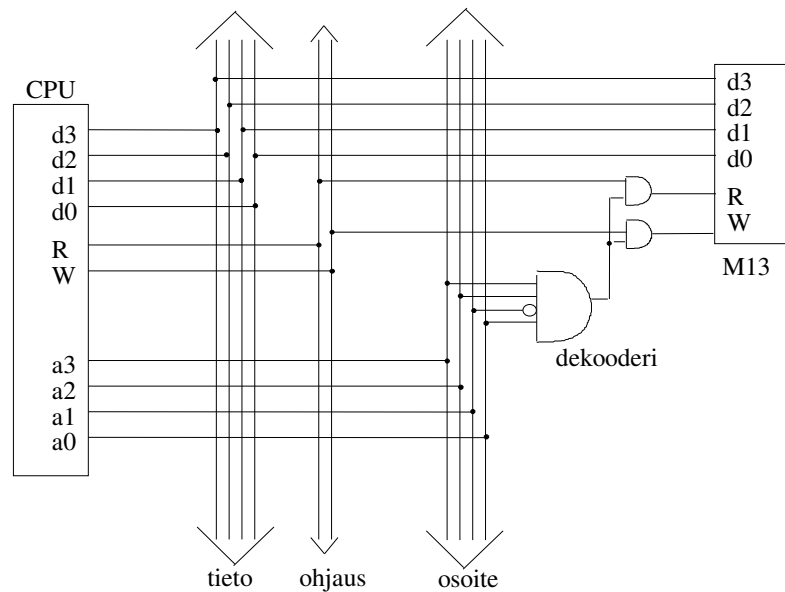
Tietokoneen eri komponentteja yhdistäviä tiedonsiirtoyhteyksiä kutsutaan **väyliksi** (bus). Väylä on kokoelma komponentit yhteenliittäviä johtoja. Väyliä on toiminnallisesti kolmenlaisia: tietoväylä, ohjausväylä ja osoiteväylä. Sama fyysinen väylä voi toisinaan toimia useammassa tehtävässä. Tietoväylä 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-, ohjau- 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 ohjaukset tunnistava **dekooderi**, joka aktivoituu osoitteesta  $13 = 1101_2$ . Tällöin  $a_3=1$ ,  $a_2=1$ ,  $a_1=0$ ,  $a_0=1$ , joka siis aktivoi muistipaikan M13. Ohjauksella R ja W ohjataan tiedon ('d niin kuin data' eli bitit d<sub>3</sub>, d<sub>2</sub>, d<sub>1</sub>, d<sub>0</sub>) 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).

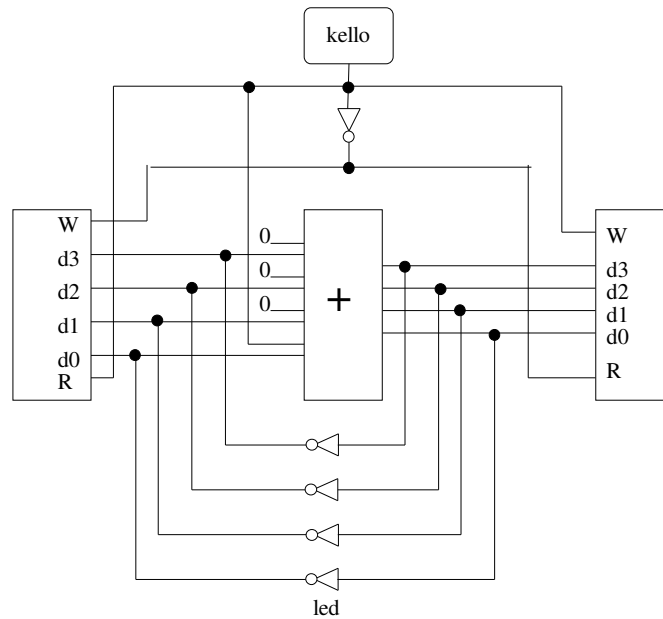




### 4.3.6 Ohjauslogiikka ja kello

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 kello-signaali on ykkönen, on vasemmanpuoleinen rekisteri aktivoitu lukemista ja oikeanpuoleinen kirjoittamista varten. Tällöin vasemmanpuoleisessa rekisterissä olevaan lukuun lisätään ykkönen ja tulos tallennetaan oikeanpuoleiseen rekisteriin. Kun kello-signaali vaihtuu, vaihtuvat myös rekisterien roolit, ja oikeanpuoleisen rekisterin sisältö kopioituu vasemmanpuoleiseen. Samalla ykkösbittien kohdalla olevat ledit palavat.



## 4.4 Mikro-ohjelmitava tietokone

### Ohjelmitavuus

Tietokoneen erottaa muista automaattisista koneista sen *ohjelmitavuus*. Tietokoneen voi ohjelmoida tekemään uusia tehtäviä muuttamatta sen rakennetta, kun taas laskukoneet, automaattipesukoneet jne. pystyvät suorittamaan vain niitä tehtäviä, joita varten ne on rakennettu. Ohjelmitavuuden perustana on se, että osa laitteesta 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-ohjelmitaviksi* (microprogrammed) tietokoneiksi, ja mikrokäskyillä kirjoitettuja ohjelmia kutsutaan mikro-ohjelmiksi (microprograms). Mikro-ohjelmista käytetään myös termiä laitemisto (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 tulkitaan 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-ohjelmanlaskuri MPC, 16-bittinen datarekisteri 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 mikrokäskyn. Käskyn bitit (c1 – c22) toimivat koneen muiden komponenttien ohjausbitteinä. Rekisterissä oleva käsky siis määrää, miten tietokoneen muodostava valtavan monimutkainen 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-ohjelmanlaskuri 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 käsittelemään 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 (shiftright-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 kytkeytyy MIR-rekisteri ohjaa tiedon siirtoa eli koneen toimintaa.

## Kello

Mikro-ohjelmoitavan tietokoneen toimintoja ohjaa viisivaiheinen kello, joka antaa pulssin syklistä 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 yhtäikaa. 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  $c_1, \dots, c_8$  siten, että kyseinen binääriluku vastaa sitä muistiosoitetta, johon halutaan hypätä ja lisäksi asetetaan  $c_{18}=1$ . 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  $c_{18}=1$  saa aikaan sen, että MIR:ssä oleva binääriluku  $c_1 \dots c_8$  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  $c_2=c_5=c_{10}=1$  ja muut bitit nolliä.

**Esimerkki.** Lisää rekisteriin A rekisterin B sisältö. Koska väylälle 1 ei voida kopioida kuin rekisterin MDR sisältö, tarvitaan toiminnon toteuttamiseen kaksi mikrokäskyä: ensimmäisessä  $c_2=c_{13}=1$  ja toisessa  $c_1=c_6=c_9=1$ .

**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 kaksois-pisteellä 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 kello sykli ja korvataan nimiöt todellisilla osoitteilla. Kirjoitetaan tässä symbolisessa esityksessä

- 1) ensimmäinen ja toinen kellonvaihe asetuslauseena (siirtona)  $x+y \rightarrow z$ ,
- 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  $x+y \rightarrow z$  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  $(x+y) \times 2$  tarkoittaa, että väylällä 3 olevaa tulosta siirretään bitin verran vasemmalle. Merkinnyt  $(A=0)$  ja  $(A<0)$  tarkoittavat vastaavien rekisteriä A tarkastelevien laitteiden tulostusta (c19, c20). Joka kellonvaiheen jälkeen väylät nollataan eli esim. siirrossa  $0+0 \rightarrow C$  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	1	1	1	1	1	1	1	1	1	2	2	2	
										0	1	2	3	4	5	6	7	8	9	0	1	2
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 ( $c_5=1$ ), muutetaan vastaluvukseen ( $c_7=1$ ), 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 ( $c_5=c_7=1$ ). 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  $A \geq 0$ ) viedään väylään 1 ja MPC ( $= 6$ ) väylään 2. Kello 5 noudetaan käskyrekisteriin seuraava käsky joko paikasta 7 ( $A < 0$ ) tai paikasta 8 ( $A \geq 0$ ).
- 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  $c_1, c_2, \dots, c_{22}$  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.  $d_{15}, d_{14}, \dots, d_0$ , jolloin ylin (merkitsevin, vasemmanpuoleisin) bitti on  $d_{15}$ . Sen vuoksi esim.  $c_1$  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  $c_1, \dots, c_{22}$  toiminnan kuvauksessa, kun siirretään dataa tietokoneessamme:

- $c_5=1$ : väylän 1 (DC1) alin bitti  $d_0$  saa arvon 1 ja muut ovat nollia.
- $c_{14}=1$  (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  $DC3_{11-0} \rightarrow MAR$ .
- $c_{18}=1$  (esim. käsky 5: yllä): tällöin  $c_1 \dots c_8$  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  $c_1 \dots c_8 \rightarrow DC1_{7-0}$
- $c_{21}=1$ :  $MDR_{15-12} \rightarrow DC1_{3-0}$ .

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  $A * C = C + C + \dots + C$ . 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.  $0+0 \rightarrow \text{MDR}$ ), 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 tärkeitä, koska ohjelma kirjoitetaan vain kerran, ja sen jälkeen kukaan ei sitä näe. Mikro-ohjelma:

osoite	mikrokäskyn selvennys	selitys
0	$0+0 \rightarrow \text{MDR}$ ; ; $1+\text{MPC} \rightarrow \text{MPC}$	MDR:lle alkuarvo; siirry seuraavaan käskyyn.
1	; ; $(A=0) + \text{MPC} \rightarrow \text{MPC}$	Haarautu sen mukaan, onko $A = 0$ vai ei.
2	; ; $5+0 \rightarrow \text{MPC}$	$A = 0$ : Poistu silmukasta (c1 ... c8 = 00000101).
3	$\text{MDR}+C \rightarrow \text{MDR}$ ; ; $1+\text{MPC} \rightarrow \text{MPC}$	$A \neq 0$ : Lisää C MDR:ään.
4	$-1+A \rightarrow A$ ; ; $1+0 \rightarrow \text{MPC}$	Pienennä A:ta, ja hyppää silmukan alkuun.

ja binäärisenä:

osoite	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	
									0	1	2	3	4	5	6	7	8	9	0	1	2		
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 kellopulsssia, joita on siis yhteensä  $5 * 200\,000 = 1\,000\,000$ . Jos oletetaan, että kelloaajuus on 10 MHz (todellisuudessa kelloaajuus on tietenkin paljon isompi), kertolaskun suoritusajaksi saadaan  $10^6 * 10^{-7} \text{ s} = 0.1 \text{ s}$ .

Kertolaskun voi suorittaa nopeamminkin. Käytetään lähtökohtana tavallista koulu-algoritmia, esimerkiksi:

$$\begin{array}{r}
 000101 \\
 000110 \\
 \hline
 000000 \\
 000101 \\
 000101 \\
 000000 \\
 000000 \\
 000000 \\
 + \hline
 00000011110
 \end{array}$$

Algoritmiin on koneemme rajoittuneisuuden vuoksi tehtävä seuraavat muutokset:

- Koska vain sanan ensimmäistä bittiä voidaan tarkastella erikseen, on osasumat "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 nolaksi. 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  $A * C = \text{MDR}$  modifioitu koulualgoritmi:

```

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

## Mikro-ohjelma:

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

Kertolaskun mikro-ohjelman kirjoittaminen binäärinä on suoraviivasta.

Tämä kertolaskuohjelma on huomattavasti aikaisempaa nopeampi. Yhden kertolaskusilmukan 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  $98 * (5 * 10^{-7} \text{ s}) \approx 50 \mu\text{s}$ . Algoritmi on siten noin 2000 kertaa nopeampi kuin naiivi kertolasku!

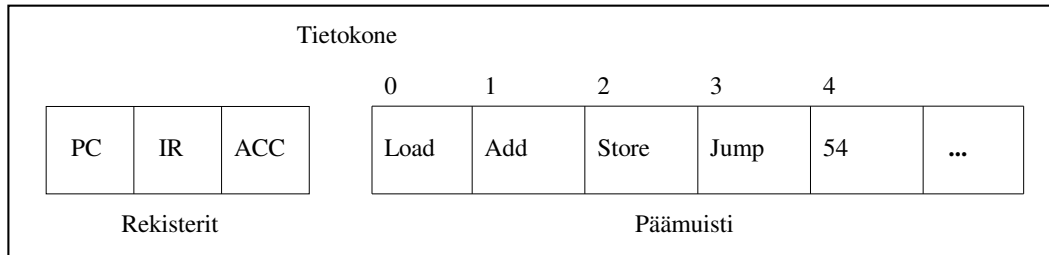
#### 4.4.5 Jakolasku

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:

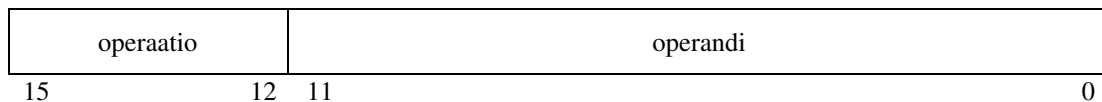


Koneessa on siis kolme erikoisrekisteriä ja päämuisti, johon on tallennettuina bittijonoina konekielen käskyistä koostuva konekielinen ohjelma ja sen käsittelemä data. Käytössä on kolme abstraktia erikoisrekisteriä:

- 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 kirjoittaessa meidän ei tarvitse miettiä näitä rekistereitä, mutta kun seuraavassa luvussa kirjoitamme mikro-ohjelmoidun tulkin konekielille, 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 suoritusjärjestyistä 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	$ACC \leftarrow (M)$
0010	m11...m0	STORE	M	$(M) \leftarrow ACC$
0011	m11...m0	ADD	M	$ACC \leftarrow ACC + (M)$
0100	m11...m0	SUBTRACT	M	$ACC \leftarrow ACC - (M)$
0101	m11...m0	MULTIPLY	M	$ACC \leftarrow ACC * (M)$
0110	m11...m0	DIVIDE	M	$ACC \leftarrow ACC / (M)$ (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  $\rightarrow$  ilmaisemaan sitä miten tietoa siirrettiin väyliltä rekistereihin:  $DC1+DC2 \rightarrow DC3$ , kun taas yllä  $\leftarrow$  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 tarvittavat 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 nollassi, tulee meillä olla nolla tallennettuna johonkin päämuistin muistipaikkaan, jonka sisältö ladataan akkuun LOAD-käskyllä.

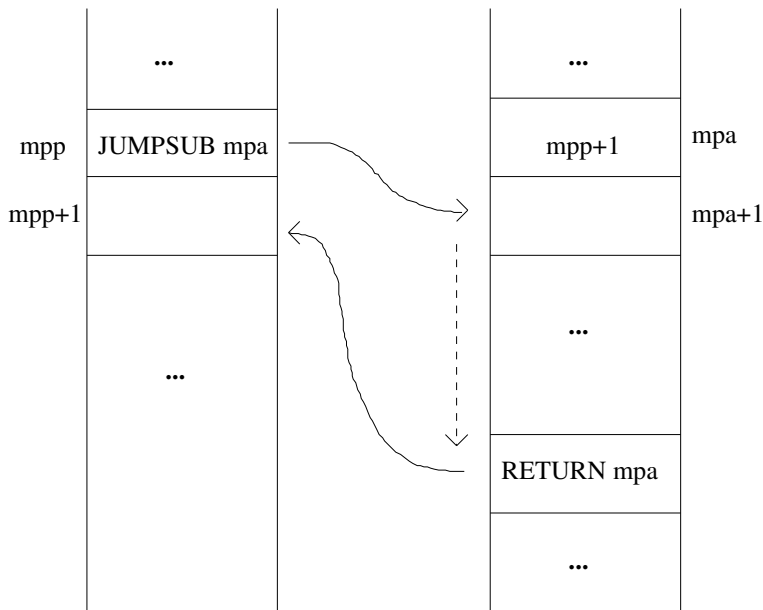
Tässä on hyvä huomata, että päämuistissa oleva data esitetään myös 16 bitillä ja se onko muistipaikan sisältö konekielinen komento vai ohjelman käsittelemää dataa (tietoa) määräytyy vasta ohjelman suoritusvaiheessa. Tarkkaavainen lukija voi tässä vaiheessa huomata myös seuraavan melko kummallisen ominaisuuden: 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. yksiosoittekieleen, jossa vain toinen binääristen operaatioiden operandi annetaan eksplisiittisesti itse käskyssä ja toinen on aina implisiittisesti akku. On olemassa myös kaksi- ja jopa kolmiosoittekieliä. Kaksiosoittekielissä toinenkaan operandi ei ole aina akku, vaan se

voidaan antaa käskyssä. Kolmiosoitekielissä myös operaation tulos voidaan viedä yhdellä käskyllä haluttuun muistipaikkaan. Yksioioitekieli 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  $(13) \leftarrow (13) + (10)$ , 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. Käsitteltävä data on muistipaikoissa 381-383.

```

MODULE exp(n) RETURNS 2n
  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	$ACC \leftarrow arvo$
378	ADD	382	$ACC \leftarrow arvo + arvo$
379	STORE	382	$arvo \leftarrow ACC$
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 2x + 2y
  RETURN exp(x) + exp(y)
ENDMODULE

```

Konekielinen ohjelma:

287	LOAD	314	Hae x akkuun.
288	STORE	311	Tallenna x aliohjelman syötteenksi.
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 syötteenksi.
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 konekielille

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 ohjelmanalaskurina 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. Ohjelmanalaskuriksi 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 <sub>15-12</sub> + MPC → MPC	Kasvata ohjelmalaskuria B, ja haarautu 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 <sub>11-0</sub> + 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 <sub>11-0</sub> + 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 <sub>11-0</sub> + 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 <sub>11-0</sub> + 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 <sub>11-0</sub> + 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 <sub>11-0</sub> + 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	$0 + C \rightarrow A; ; 0 + 0 \rightarrow MPC$	Osamäärä C:ssä: $ACC \leftarrow C$ ; tulkkaa seur. käsky.
61	$MDR + 0 \rightarrow B; ; 0 + 0 \rightarrow MPC$	JUMP-käsky. Tulkkaa seuraava.
62	$; ; (A = 0) + MPC \rightarrow MPC$	JUMPZERO-käsky alkaa
63	$MDR + 0 \rightarrow B; ; 0 + 0 \rightarrow MPC$	$ACC = 0$ : Hyppää annettuun muistipaikkaan.
64	$; ; 0 + 0 \rightarrow MPC$	$ACC < 0$ : Tulkkaa seuraava käsky.
65	$; ; (A < 0) + MPC \rightarrow MPC$	JUMPNEG-käsky alkaa
66	$MDR + 0 \rightarrow B; ; 0 + 0 \rightarrow MPC$	$ACC < 0$ : Hyppää annettuun muistipaikkaan.
67	$; ; 0 + 0 \rightarrow MPC$	$ACC \geq 0$ : Tulkkaa seuraava käsky.
68	$MDR + 0 \rightarrow C, MAR; ; 1 + MPC \rightarrow MPC$	JUMPSUB alkaa
69	$0 + B \rightarrow MDR; MDR \rightarrow (MAR);$ $1 + MPC \rightarrow MPC$	Tallenna paluusoite aliohjelman alkuun.
70	$1 + C \rightarrow B; ; 0 + 0 \rightarrow MPC$	Hyppy aliohjelman toiselle riville. Tulkkaa seuraava käsky.
71	$MDR_{11-0} + 0 \rightarrow MAR; (MAR) \rightarrow MDR;$ $1 + MPC \rightarrow MPC$	RETURN alkaa: Paluusoite MDR:ään
72	$MDR + 0 \rightarrow B; ; 0 + 0 \rightarrow MPC$	Paluusoite 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: mikro-käskyt 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 nolaksi, 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: mikro-käskyt 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 muisti-  
paikan sisältö ja indeksirekisterin sisältö

Osoitustapa	Latauskäsky	Merkitys
välitön (Immediate)	LOADI <i>oso</i>	$ACC \leftarrow oso$
suora	LOAD <i>oso</i>	$ACC \leftarrow (oso)$
epäsuora (InDirect)	LOADID <i>oso</i>	$ACC \leftarrow ((oso))$
indeksoitu (IndeXed)	LOADIX <i>oso</i>	$ACC \leftarrow (oso+X)$
epäsuora indeksoitu	LOADIDX <i>oso</i>	$ACC \leftarrow ((oso)+X)$

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

indeksirekisteri		muisti	
4			...
		280	282
		281	87
		282	13
		283	27
		284	16
		285	66
		286	77
			...

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 nolasta 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:

- 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 saakka 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 osoittama seuraavaksi suoritettava 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 magneettinauhvoja 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 siirräntässä** 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 rajoittaa 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 valmisteleva 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 (reititys algoritmit), 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 lähetysreitit 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. *kilpavarauusväylä*: 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, liittyy tiedon paketin perille tulosta paketin loppuun ja lähettää paketin eteenpäin, kunnes se saapuu takaisin lähettäjälleen, joka poistaa paketin renkaasta ja vapauttaa vuoromerkkin.
- 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 Systemiohjelmisto

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 tuhlavaista 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. systemiohjelmaa (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 systemiohjelmaa, joita ei käsitellä tässä yhteydessä tarkemmin, ovat:

### 3) Linkittäjät

Linkittäjien (linkers) tehtävänä on koota erillään käännetty 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. levyiltä) keskusmuistiin.

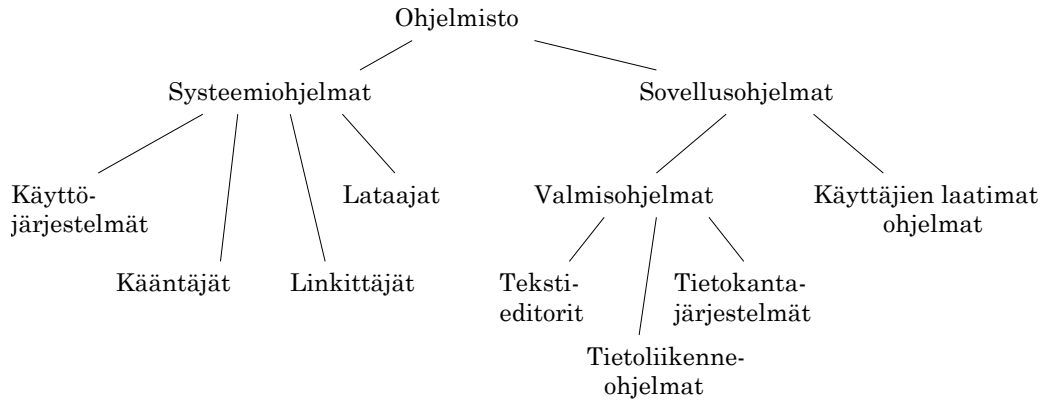
### 5) Editorit

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).

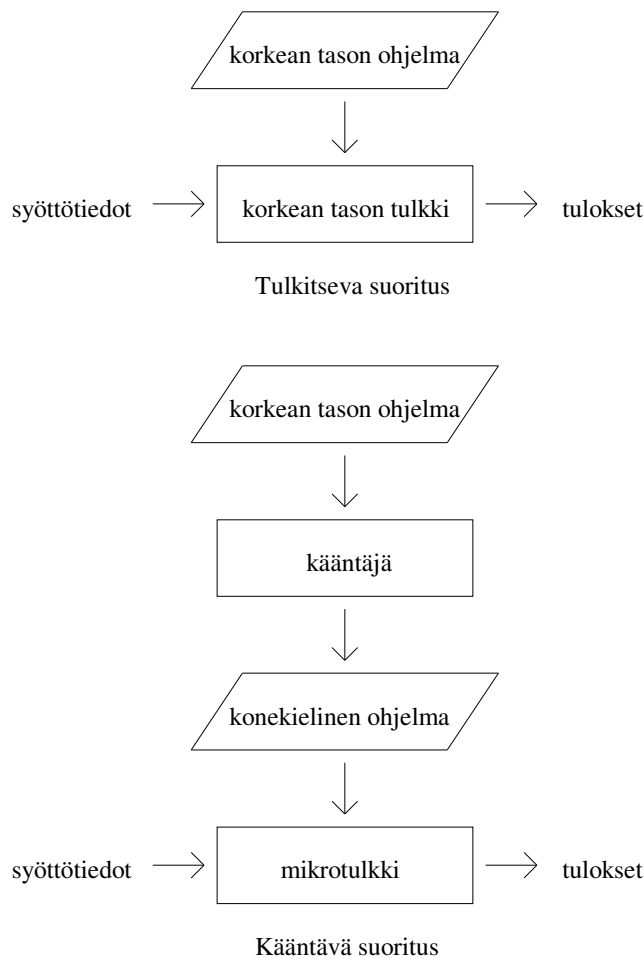
Korkean tason ohjelmointikielissä on luonnolliset ohjaus- ja tietorakenteet. Konekielissä ohjausrakenteet toteutetaan primitiivisillä operaatioilla (esim. JUMP-käsky) ja tietorakenteet primitiivisten, osoitteisiin kohdistettavien, operaatioiden avulla. Niinpä algoritmien suunnittelu on helpompaa ja ohjelmakoodi 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:



### 5.1.1 Tulkitseminen

Tulkitsemisessä 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 yksinkertaisten tulkittavien 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 syntyneitä 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 kehitysvaiheeseen 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 määrättyllä 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 määrittellä, 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  $A \rightarrow w$ , 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 päätesymboleiksi, eli kieliopissa on kullekin välisymbolille määriteltävä sellainen sääntö, että välisymbolit voivat laventua päätesymboleiksi.

Jokainen syntaktisesti kelvollinen (kieliopin mukainen) päätesymbolijono voidaan esittää *johto- eli jäsenyspuuna* (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 päätesymbolijonon 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>, <intransiiviverbi>, <transiiviverbi>

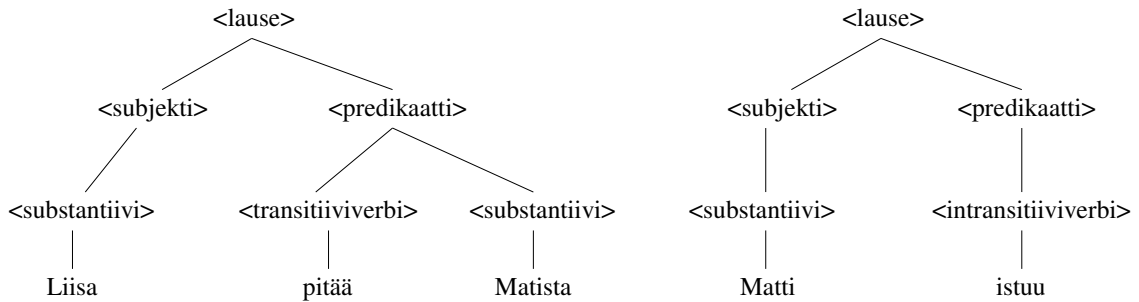
Terminaalit: Matti, Liisa, Matista, istuu, pitää

Produktiot:

```

<lause> → <subjekti><predikaatti>
<subjekti> → <substantiivi>
<predikaatti> → <intransiiviverbi>
<predikaatti> → <transiiviverbi> <substantiivi>
<substantiivi> → Matti
<substantiivi> → Liisa
<substantiivi> → Matista
<intransiiviverbi> → istuu
<transiiviverbi> → pitää
    
```

Johto- eli jäsenyspuita:



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:

$N \rightarrow w_1 \mid w_2 \mid \dots \mid w_k$  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.

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

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

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

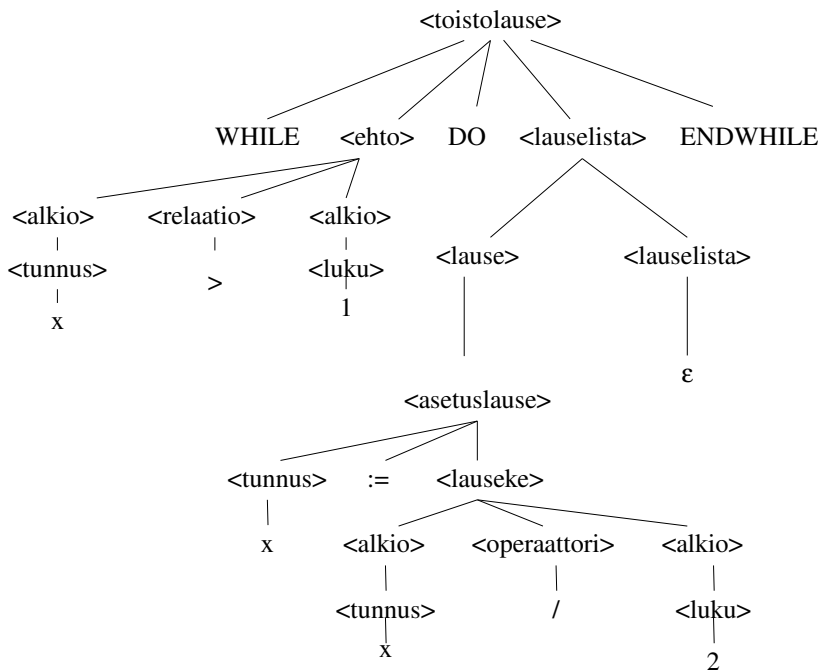
```

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



### 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 tunnustaja, ja selauksen tulos on selattujen tekstialkioiden lista.

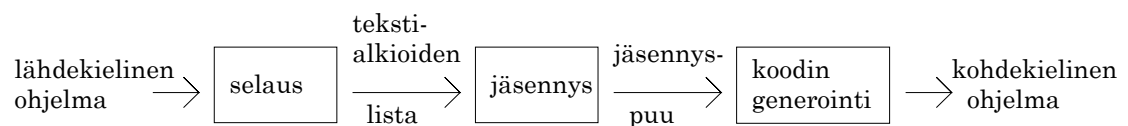
## 2) Syntaktinen analyysi (syntax analysis) eli jäsentäminen

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 samanaikaisesti.



### 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 tyyppiä:

- 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 tyyppiin. 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 jäsennyksessä, jossa päätesymbolit ovat lehtisolmuissa, ja syntaksiluokka, johon päätesymbolijono kuuluu, puun juuressa. Jäsennyksessä muodostamiseen on olemassa kaksi päästrategiaa: osittava (top-down) jäsennyksessä eli juuresta lehtiin -menetelmä ja kokoava (bottom-up) jäsennyksessä eli lehdistä juureen -menetelmä.

Osittavassa jäsennyksessä ohjelman jäsennyksessä 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 jäsennyksessä 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 epäonnistuneensa jatkossa.

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

#### Osittava jäsentäminen

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

$$S \rightarrow s_1 \dots s_m \quad (s_i\text{:t ovat nonterminaaleja tai terminaaleja)}$$

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

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

```

MODULE jäsenitys(nonterminaali N, terminaalijono w, johtopuu P) RETURNS onnistuu / ei
  REPEAT
    Kokeile produktiota N → u
    IF u = w THEN (* u oikea terminaali, jäsenitys onnistuu suoraan *)
      P := N(w)
      RETURN onnistuu
    ELSE (* u:ssa yksi tai useampia nonterminaaleja *)
      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 jäsenitys(Ni, wi, Pi) onnistuu (* rekursiivinen kutsu ! *) THEN
          P := N(u0, P1, u1, ..., Pk, uk)
          RETURN onnistuu
        ENDIF
      ENDWHILE
    ENDIF
  UNTIL kaikki produktiot N → ... on kokeiltu
  RETURN ei onnistuu
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) jäsenitys(<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  $w_1 = \text{luku} - \text{luku}$ ,  $w_2 = \text{luku}$ . Tämä johtaa rekursiivisiin kutsuihin:

- (5) jäsenitys(<lauseke>, luku – luku, P<sub>1</sub>) ja
- (6) jäsenitys(<lauseke>, luku, P<sub>2</sub>)

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

$P_2 := \text{<lauseke>}(\text{luku})$

Jäsenityksessä (5) oikea produktio on (2) ja oikea jako  $w = w_1 - w_2$ , missä  $w_1 = w_2 = \text{luku}$ . Se johtaa kutsuihin:

- (7) jäsenitys(<lauseke>, luku, P<sub>3</sub>) ja
- (8) jäsenitys(<lauseke>, luku, P<sub>4</sub>)

Ilmeisesti (7) ja (8) ratkeavat suoraan:

$P_3 := \text{<lauseke>}(\text{luku})$  ja  
 $P_4 := \text{<lauseke>}(\text{luku})$

Mutta tällöin myös (5) onnistuu:

$P_1 := \text{<lauseke>} (P_3, -, P_4)$   
 $= \text{<lauseke>} (\text{<lauseke>} (\text{luku}),$   
 $-,$   
 $\text{<lauseke>} (\text{luku}))$



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

```

MODULE Generoi(jäsenyspuu P)
  Olkoon P = N(P1, ..., Pk)
  CASE N OF
    <moduuli> :      generoimoduuli(P1, ..., Pk)
    <asetuslause> :  generoiasetuslause(P1, ..., Pk)
    <joslause> :     generojoslause(P1, ..., Pk)
    <josmuutoin> :   generojosmuutoin(P1, ..., Pk)
    <whilelause> :  generoiwhilelause(P1, ..., Pk)
    ....
  ENDCASE
ENDMODULE

```

Kieliopin tärkeimmille välisymboleille kirjoitetaan koodin generointiin omat moduulinsa. Kirjoitetaan koodin generoiva moduuli asetustalauseelle  $x := y$  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 asetustalause  $x := y$  o  $z$  on jäsenetty tämän kieliopin mukaisesti, tulos on muotoa

$\langle \text{asetustalause} \rangle (\langle \text{tunnus} \rangle (x), :=, \langle \text{lauseke} \rangle (\langle \text{tunnus} \rangle (y), \langle \text{operaattori} \rangle (o), \langle \text{tunnus} \rangle (z)))$

Kääntämisen kannalta tässä on tarpeellontakin informaatiota. Sama tieto on puussa

$\langle \text{asetustalause} \rangle (x, y, o, z)$

Kirjoitetaan käänno-moduuli tämän mukaisesti.

```

MODULE generoiasetustalause(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 jäsenyspuu

$\langle \text{josmuutoin} \rangle (\text{IF}, \langle \text{ehto} \rangle (...), \text{THEN}, \langle \text{lause} \rangle (...), \text{ELSE}, \langle \text{lause} \rangle (...))$

kääntyy seuraavasti:

```

MODULE generiojosuutoin (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.
  generioehto (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 generioehto(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 *)
  generioehto(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 yhteenvedonomaaisesti, mitä kokonaiselle ohjelmalle tapahtuu käännettäessä. Olkoon määritelty seuraava kielioppi (mietä 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 konekielille:

```

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

### 2) Jäsennys

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

```

<ohjelma> (
  <moduuli> (
    <tunnus> (synt),
    <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
synt	INTEGER	-
x	INTEGER	-
y	INTEGER	-

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

```

paluu: paluuosoite
        JUMP      b0
x:
y:
synt:  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>
        SUBTRACT y <asetuslause>(x, x, -, y) * <toistolause>
        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    synt <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 vältetään 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 operaatiokoodilla.

Assembler-kääntäjäkään ei yleensä tuota lopullista objektikoodia: käännettyssä 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:

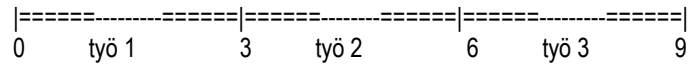
- siirrä kääntäjä oheismuistista keskusmuistiin
- käynnistä kääntäjä, ja syötä sille käännettävä ohjelma joltakin syöttölaitteelta
- käännä ohjelma, ja sijoita käännökseen tulos, objektiohjelma, oheismuistiin
- linkitä käännetty ohjelma ja sen apuohjelmat latausmoduuliksi
- siirrä latausmoduuli oheismuistista 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.

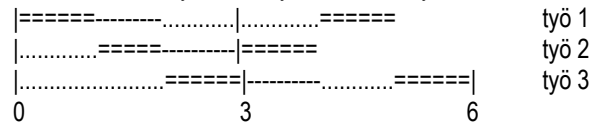
**Esimerkki.** Töiden suorittaminen peräkkäin ja rinnakkain. Suoritetaan kolme samanlaista työtä. Merkitään:

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

Työt suoritetaan peräkkäin:



Kokonaissuoritus aika 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 siirräntä 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 yksiprosessorissa 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:

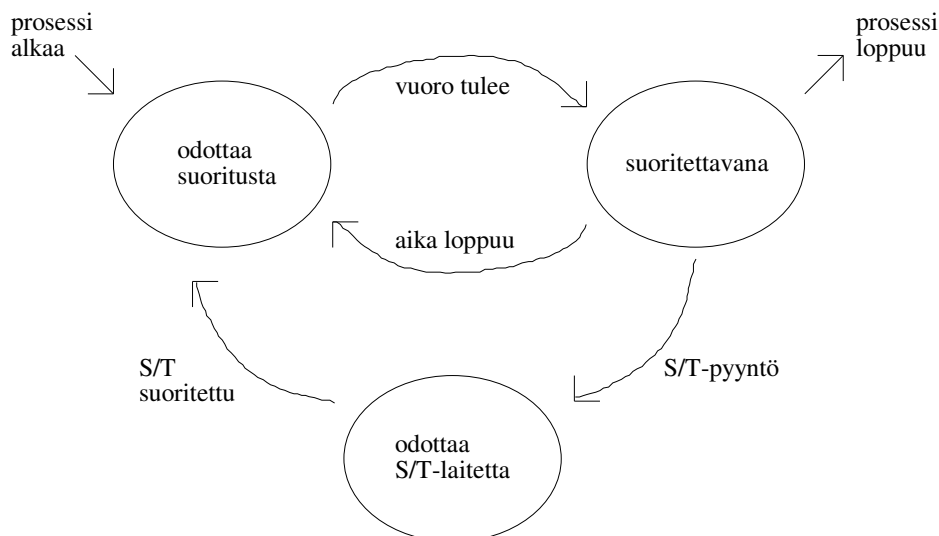
- suoritustila (running)
- suoritustilasta (ready)
- siirräntälaitteen (tai muun resurssin) odotustila (blocked)

Prosessi on käynnistämisen jälkeen suoritustilassa eli odottaa suoritustilastaan. Suoritustilasta 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 suoritustilastaan 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 suoritustilastaan, 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, suoritustilastaan odottava (suoritustilassa oleva) prosessi siirtyy suoritettavaksi (suoritustilaan).
- Suoritustilassa olevan prosessin aika loppuu: prosessi siirtyy odottamaan suoritustilastaan (suoritustilastaan) ja uusi, suoritustilastaan odottava prosessi siirtyy suoritettavaksi (suoritustilaan).
- Suoritustilassa oleva prosessi päättyy: uusi, suoritustilastaan 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 siirrätälaitteen 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

Suoritettujen prosessien 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 (varmistuen 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 ohjelmavivun 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 ohjelmavivut 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 siirräntän 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 siirräntän, siirtää prosessin siirräntän odotustilaan ja pyytää vuoronantajaa valitsemaan uuden prosessin suoritettavaksi. Siirräntän 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 siirräntän 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 *sivuaajotekniikkaa* (spooling; Simultaneous Peripheral Outputting On-Line): kaikki siirräntä jakamattomalle laitteelle suunnataan jollekin välissä olevalle jaettavissa olevalle laitteelle, esim. levyille, josta se sitten myöhemmin edelleen siirretään jakamattomalle laitteelle.

Tarkastellaan esimerkiksi prosessia, joka haluaa tulostaa kirjoittimelle. Prosessi kutsuu käyttöjärjestelmän siirräntäohjelmaa, joka lähettääkin tulosteen levyille kirjoittimen sijasta. Muiden prosessien kirjoittimelle suunnatut tulosteet lähetetään samalla tavoin levyille 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 muistilaitte, 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.

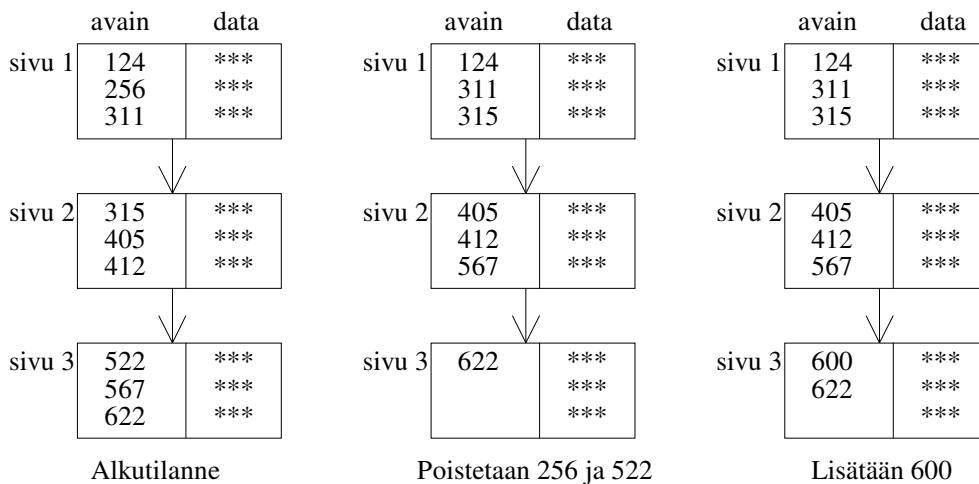
Oheismuistin luku ja kirjoitus tapahtuvat sivu kerrallaan. Oheismuistin käsittelynopeudessa on ratkaisevaa suoritettujen luku- ja kirjoitusoperaatioiden lukumäärä, ei niinkään siirretyn datan volyymi. Siksi kannattaa käyttää melko suurta sivukokoa, ja pyrkiä sijoittamaan peräkkäin käsiteltävät tietueet 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 määritetty.

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 tulokoon 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 määrittämiseksi 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-organisaatioissa (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  $h(x) = x \bmod 7 + 1$ . Oletetaan yksinkertaisuuden vuoksi, että käytettävissä on peräkkäinen joukko oheismuistin sivuja. Suljettu hash lineaarisella ylivuotomenettelyllä toimii seuraavasti:

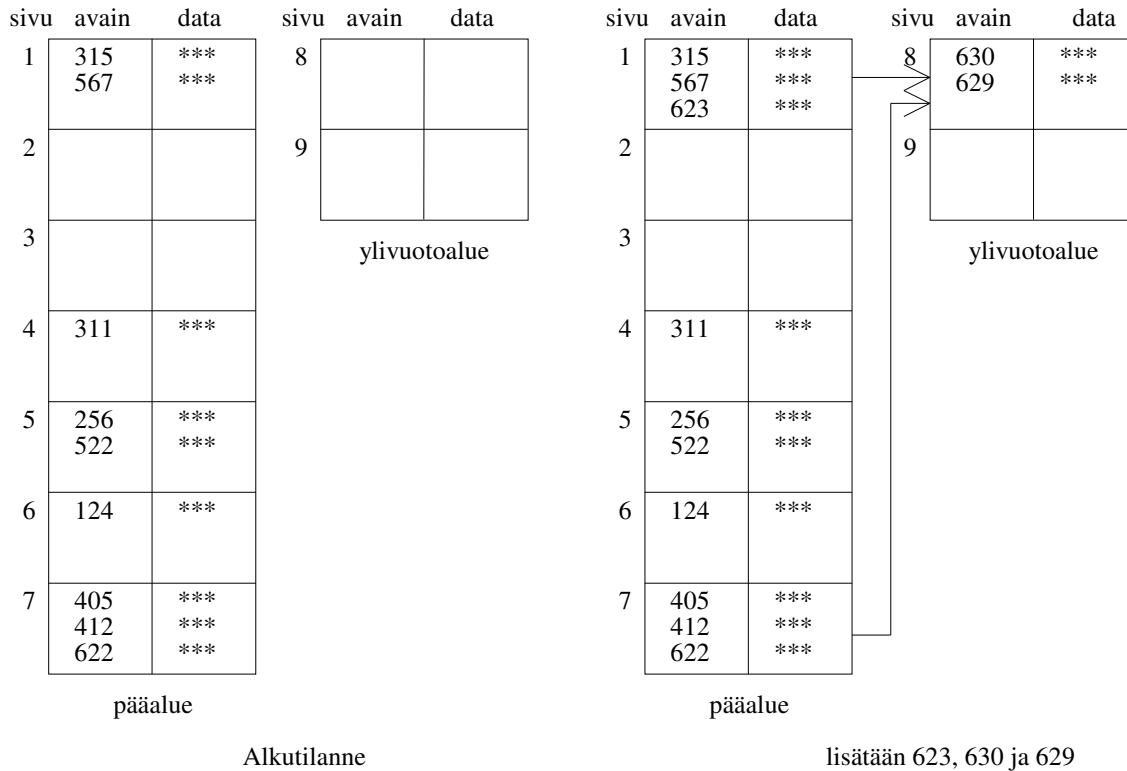
sivu	avain	data
1	315 567	*** ***
2		
3		
4	311	***
5	256 522	*** ***
6	124	***
7	405 412 622	*** *** ***

Alkutilanne

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

lisätään 623 ja 630

Lisäysten suoritus käyttäen avointa hashia ylivuotoalueella:

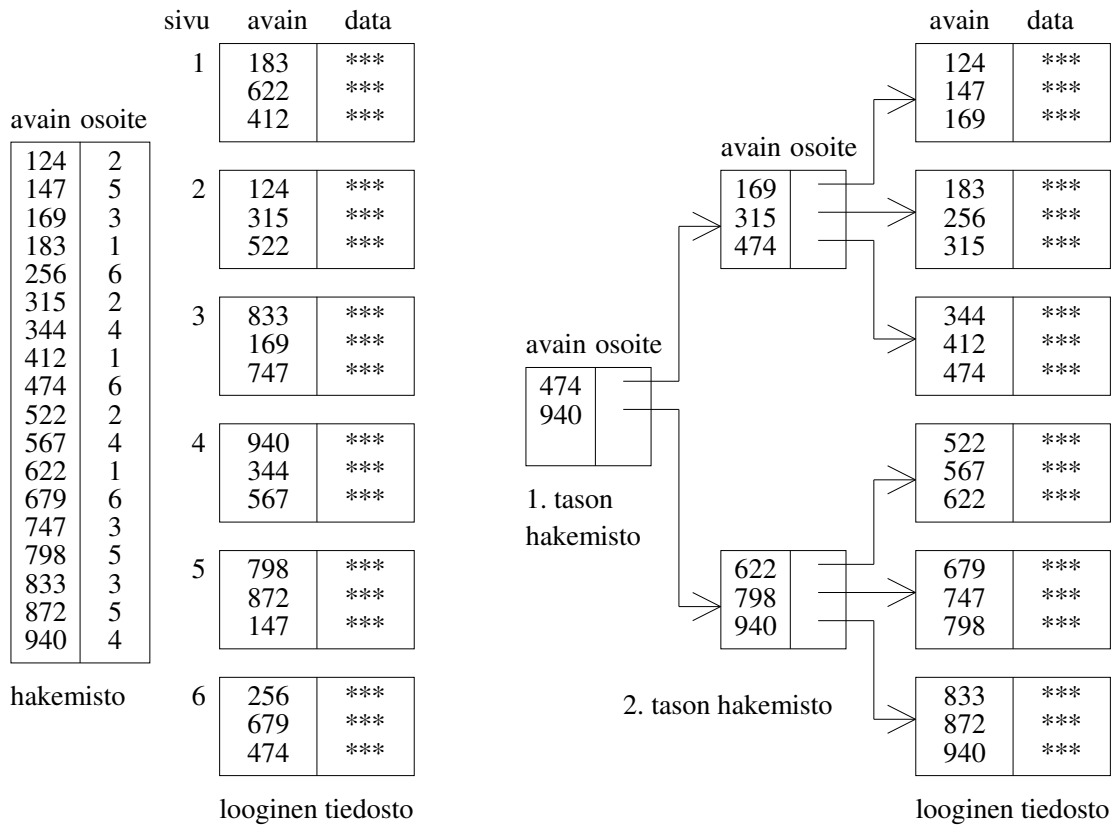


### 5.5.3 Indeksoitu organisaatio

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-osoitepareista. 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.

asno	osoite
124	2
256	1
311	1
315	3
405	1
412	3
522	3
567	2
622	2

ensisijainen  
hakemisto

sivu	asno	saldo	data
1	405	800	***
	256	800	***
	311	300	***
2	622	500	***
	124	500	***
	567	300	***
3	522	800	***
	412	1200	***
	315	300	***

looginen tiedosto

saldo	asno
300	311, 315, 567
500	124, 622
800	256, 405, 522
1200	412

toissijainen  
hakemisto eli  
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 vaihtoehtoihin 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 löytäminen voi kestää liian kauan.

## 6.1.2 Syvyys- ja leveysshaku

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

Leveysshakualgoritmi:

```

MODULE leveysshaku
  Pane alkutila jonoon
  WHILE jono ei ole tyhjä AND lopputilaa ei ole saavutettu DO
    Ota jonosta tila s
    Olkoot  $s_1, \dots, s_k$  ne vielä tutkimattomat tilat, joihin s:stä pääsee yhdellä siirrolla
    Aseta osoittimet tiloista  $s_1, \dots, s_k$  tilaan s
    Lisää tilat  $s_1, \dots, s_k$  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 pinnoa. 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: äärellisessäkin haussa ei välttämättä löydetä parasta ratkaisua.

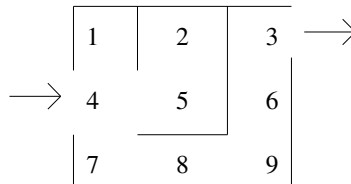
Syvyyshakualgoritmi:

```

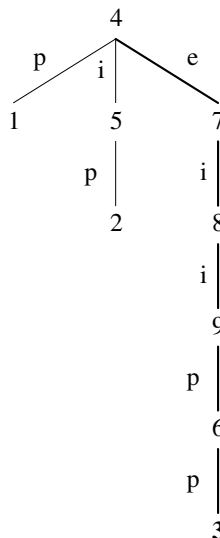
MODULE syvyyshaku
  Pane alkutila pinoon
  WHILE pino ei ole tyhjä AND lopputilaa ei ole saavutettu DO
    Ota pinosta tila s
    Olkoot  $s_1, \dots, s_k$  ne vielä tutkimattomat tilat, joihin s:stä pääsee yhdellä siirrolla
    Aseta osoittimet tiloista  $s_1, \dots, s_k$  tilaan s
    Lisää tilat  $s_1, \dots, s_k$  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 algoritmossa täytyy vielä tarkentaa se, ts. missä järjestyksessä tilat  $s_1, \dots, s_k$  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:

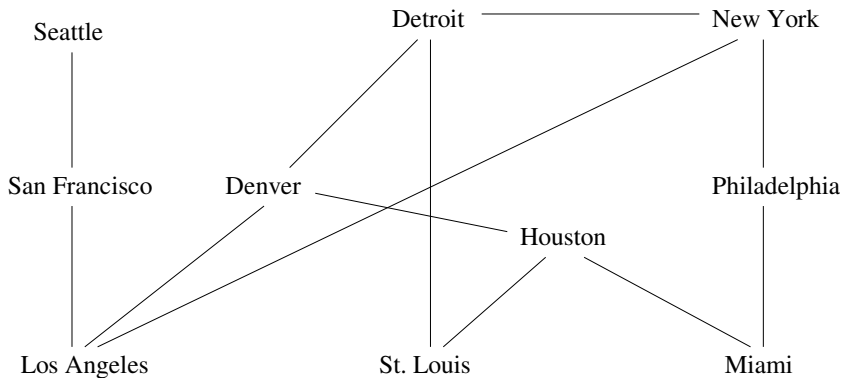
- 4 poistetaan 4, lisätään 1, 5 ja 7, saadaan:
- 1, 5, 7 alkion 1 poisto listan alusta, saadaan:
- 5, 7 alkion 5 poisto alusta ja alkion 2 lisäys listan loppuun, saadaan:
- 7, 2
- 2, 8
- 8
- 9
- 6
- 3

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

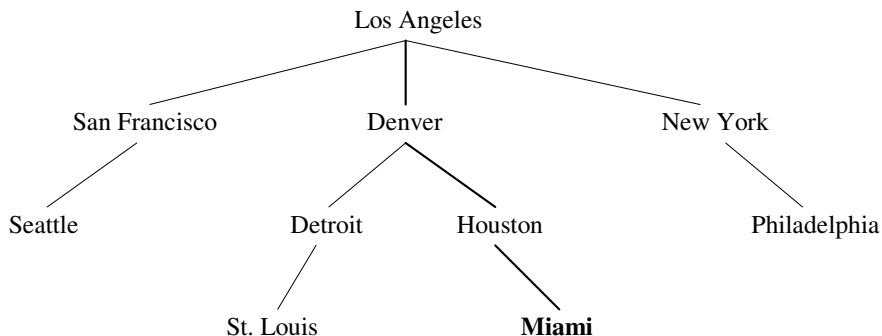
Jos tehtävä ratkaistaan syvyysshaulla samaa heuristiikkaa käyttäen, käydään nytkin koko labyrintti läpi — joskin eri järjestyksessä (kirjoita harjoituksena pinon kehitys näkyviin). Pinon käsittelyssä on huomattava myös se, että alkioit tulee ne laittaa pinoon huonommuusjärjestyksessä, koska tällöin paras 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?



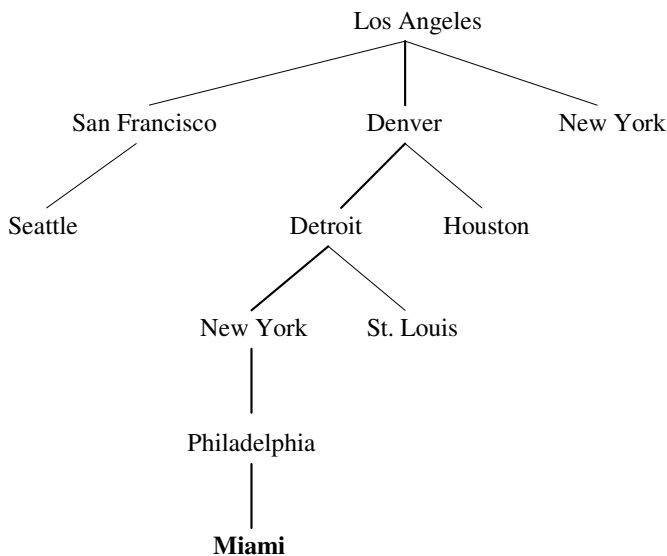
Ongelman tilaesitys on seuraava: tiloina ovat kaupungit, alkutila Los Angeles ja lopputila Miami. Siirrot ovat kaupunkien väliset lennot. Heuristiikka voidaan määrittellä nytkin ilmansuuntien mukaan: pyritään ensin pohjoiseen, ja kokeillaan sitten lähinnä myötätäivään seuraavaa lentoa. Ratkaistaan tehtävä 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 ensimmäistään neljän tai useamman lennon ratkaisua. Käytetään sitten syvyyshakua samalla heuristiikalla:

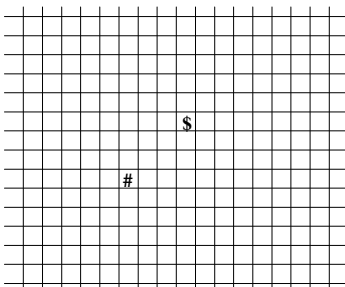


Pinon kehitys:

- 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 syvyyshauhallakin, jos vain osattaisiin heti alkutilassa kokeilla ensin New Yorkin suuntaa, mutta haku ei peräänny valitsemaltaan Denverin reitiltä, ellei saavuta umpikujaan. Näin heuristiikan merkitys on syvyysshaussa 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 hyvyttä arvioidaan erityisellä heuristisella **arviofunktioilla**, joka liittyy 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 arviofunktioista.

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

```

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

```

**Esimerkki.** Jätkänsakki. Jätkänsakissa 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:

	x	

Oletetaan, että pelaaja O käyttää samaa arviofunktioita. 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:

o		
	x	

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:

		x
	x	

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

o		x
	x	
o		

o		x
	x	
	o	

o		x
	x	o

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

### Minmax-tekniikka

Heuristiselle arviofunktioille 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 pelaaparemmiin
  Olkoon alkutila s
  REPEAT
    Olkoot  $s \rightarrow s_i$  ( $i = 1, \dots, k$ ) mahdolliset siirrot
    Laske kullekin  $s_i$ :lle luku
       $g(s_i) = \min \{f_X(s_{ij}) \mid s_i \rightarrow s_{ij} \text{ on vastustajan mahdollinen siirto}\}$ 
    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 jätkänskakki-pelissä, jos pelaajat käyttävät minmax-tekniikkaa. Tilanne oli siis:

o	x
	x

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,  $s_{ij}$ :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  $s_{ij}$ :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



o		x
	x	
o		

jossa X:llä on siirtovuoro, menetelmä epäonnistuu taas. Mistä apu? On kaksi vaihtoehtoa: minmax-tekniikan laajenus siten, että lasketaan n (n>1) siirtoparin päähän (minmax<sup>n</sup>-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 jätksinkertainen 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 rinnakkaiskäsitellyn 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 (yhtäaika).

**Esimerkki.** Kahden samanpituisen kokonaislukuvektorin erotus vastinalkioittain.

$$\begin{array}{rcl} X & = & (12 \quad 23 \quad 11 \quad 9 \quad -11 \quad 34 \quad 56 \quad 101) \\ Y & = & (0 \quad 1 \quad 200 \quad 200 \quad 2 \quad 4 \quad 2 \quad 49) \\ X - Y & = & (12 \quad 22 \quad -189 \quad -191 \quad -13 \quad 30 \quad 54 \quad 52) \end{array}$$

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 suoritus aika 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
```

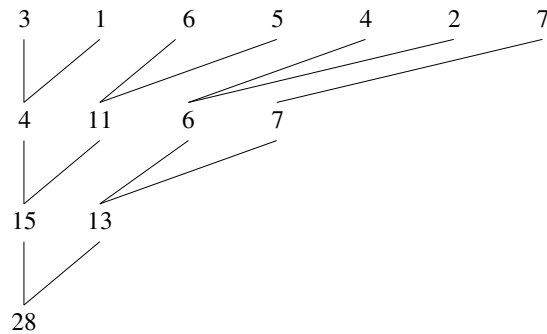
Tässä indeksi  $k$  viittaa sekä vektorin  $k$ :nteen alkioon että  $k$ :nteen käytössä olevaan prosessoriin. Rinnakkaisalgoritmin suoritus aika ei riipu  $n$ :stä: kukin prosessori  $k$  ( $k = 1, \dots, n$ ) suorittaa yhden vähennyslaskun. Näin  $n$ -kertaistamalla prosessorien määrä algoritmi saadaan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

**Esimerkki.** Vektorin alkoiden summan laskeminen. Peräkkäisalgoritmi:

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

Olkoon  $V.length=n$ . Tämänkin peräkkäisalgoritmin suoritus aika 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 yhtäaikaan  $\lfloor n/2 \rfloor$  prosessoria käyttäen ( $\lfloor n/2 \rfloor$  tarkoittaa pyöristystä alaspäin). Jos  $n$  on pariton, niin viimeinen luku kopioidaan sellaisenaan seuraavalle kierrokselle, johon siis tulee  $\lceil n/2 \rceil$  (pyöristystä 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 suoritus aika on suunnilleen se aika, joka kuluu yhdeltä prosessorilta algoritmin suorittamiseen. Suoritus aika on siis  $T(n) \sim \log n$ . Jos peräkkäis algoritmi 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 peräkkäis algoritmi, mutta toisaalta prosessoreita tarvitaan peräti 500 000.

### 6.2.2 Rinnakkaisalgoritmien kompleksisuus

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

Naiivin lajittelun aikakompleksisuus on neliöllinen, ja parhaiden peräkkäis algoritmien (esimerkiksi limityslajittelu) aikakompleksisuus on  $T(n) \sim n \log n$ . Parhaan rinnakkaisen lajittelualgoritmin aikakompleksisuus on  $\log n$ , jolloin tarvitaan  $n \log n$  prosessoria. Näin paras rinnakkainen lajittelualgoritmi on aikakompleksiuudeltaan  $n$  kertaa parempi kuin paras peräkkäis algoritmi, 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 peräkkäis-laskennassa 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. Vapaa-järjesteiset 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 rinnakkais-laskentaan.

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 luettelamalla listan ja sen alilistojen alkiot hakasuluissa, esimerkiksi:

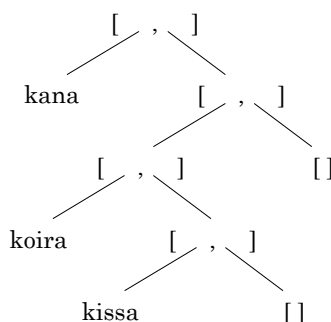
```
[koira kissa kana]
[3 7 14 2]
[norsu]
[tässäkin on [hieman monimutkaisempi] lista]
[]
```

Yleistetyt listat ovat rakenteeltaan kaksiosaisia: listan ensimmäistä alkioita 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  $X \mid L$  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 alkioille  $X$  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  $\lambda$ -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 alkioiksi jollakin operaatiolla *op*. Määriteltäessä redusointifunktiota *reduce* määritellään operaatio *op* (edellä *add* tai *times*), ja operaation tulos *base* kohdistettaessa se tyhjiin 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 *)
```

Proessori 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 kalkyyliä 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 suoritus aikaan, 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 suoritus aikaan 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, etenkin 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, käännökseen tulos on usein hyvin monimutkainen ja hidas suoritettava.

### 6.3.3 Logiikkaohjelmointi

Logiikkaohjelmointi on viime vuosisadan alkupuoliskolla kehitettyyn *predikaattikalkyyliin* perustuva deklarativinen ohjelmointimenetelmä. Siinä ongelma kuvataan epädeterministisesti predikaattilogiikan avulla, ja ohjelman suoritus tapahtuu yleensä syvyysshaun ohjaamana päättelyä. Logiikkaohjelmointi poikkeaa tavanomaisesta imperatiivisesta ohjelmointiajattelusta funktionaalista ohjelmointiakin enemmän; onhan relaatio funktiotakin vieraampi käsite konventionaalisessa ohjelmoinnissa. Logiikkaohjelmalla on tavallaan kaksinkertainen merkitys — sekä deklarativinen että proseduraalinen semantiikka: ohjelmoija tulkitsee ohjelmaa deklarativisesti ja kone proseduraalisesti. Deklarativinen tarkastelukulma sopii ongelman määrittelyyn mainiosti, mutta valitettavasti deklarativisen näkökulman yhteensovittaminen proseduraalisen kanssa ei aina onnistu täydellisesti.

Logiikkaohjelma samoin kuin ratkaistava ongelman 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*:  
 $p \leftarrow q_1, q_2, \dots, q_n$
3. Negatiiviset klausuulit eli *tavoitteet*:  
 $\leftarrow q_1, q_2, \dots, q_m$

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. logiikkaohjelmalla 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 objekta.

Logiikkaohjelmaa suorittaessaan prosessori yrittää osoittaa tavoitteen todeksi käyttäen annettuja klausuuleita systemaattisesti hyväkseen. Päätely 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.

Päätelyn 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 proseduurin *munii(X)* (eli osoitettava, että X munii) ja proseduurin *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)

Proessori 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 ajatteluaan sisältävän kaiken oleellisen tiedon (ns. *suljetun maailman oletamus*).

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:

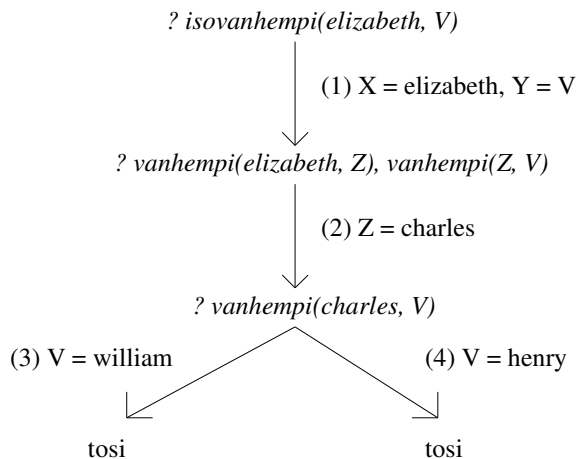
**Esimerkki.** Logiikkaohjelma.

- (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

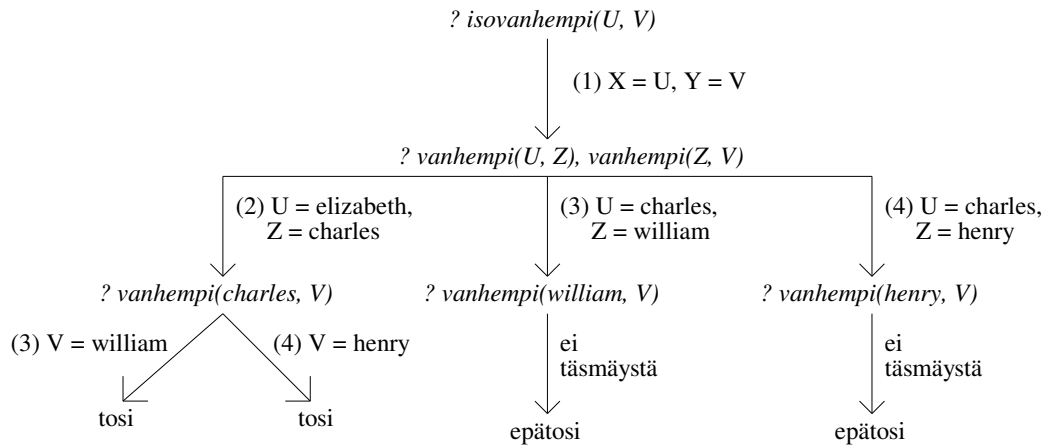
Tavoite on totta, eli Elizabeth on Henryn isovanhempi. Jos tavoite on "etsi kaikki Elizabethin lastenlapset" tai "onko Elizabeth (kenenkään) isovanhempi?", ohjelman suoritus etenee seuraavasti:



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 (lopputulokset: tosi tai epätosi), minkä jälkeen palataan takaisin edelliseen sellaiseen haaraumiskohtaan, 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

? isovanhemi (elizabeth, henry)	"Onko Elizabeth Henryn isovanhemi?"
? isovanhemi (elizabeth, V)	"Etsi kaikki Elizabethin lastenlapset V."
? isovanhemi (U, william)	"Etsi kaikki Williamin isovanhemmat U."
? isovanhemi (U, V)	"Etsi kaikki isovanhemi-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 alkioita. 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 alkioita. 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 tietämuskanta- ja asiantuntijajärjestelmissä, siis paljolti samantapaisissa tehtävissä kuin funktionaalista ohjelmointia USA:ssa.