Esipuhe

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

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

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

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

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

Turussa 25.6.2012

Jorma Boberg e-mail: boberg@utu.fi

Kirjallisuutta

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

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

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

Sisällysluettelo

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 keinotekoista 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 tietokoneohjelmina, 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.

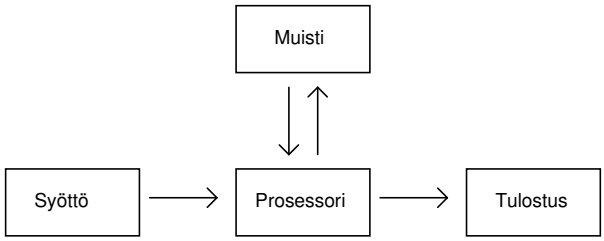
Viime aikoina tietokoneiden käyttö on laajentunut hyvin voimakkaasti perinteisten sovellusalojen ulkopuolelle. Tietokoneet ovat tulleet osaksi jokapäiväistä kulutuselektroniikkaa, ja raja varsinaisten tietokoneiden ja automaattisia toimintoja sisältävien

laitteiden ja koneiden välillä on hämärtynyt. 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 integroituja piirejä (Application Spesific 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 kiihtyvyysanturit. 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 määritellä 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, "konetta", 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, kyseessähän on vain nopea ohjelmoitava ***laskija***. Koneen ohjelmoitavuus 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.

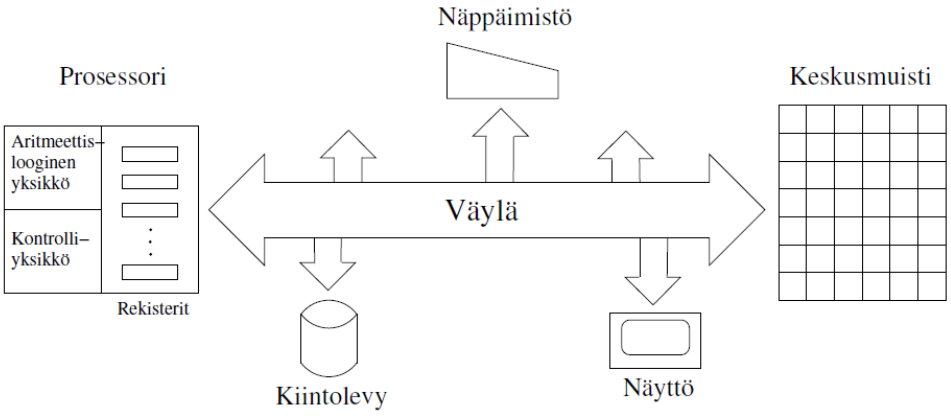


Tietokoneen toiminnan kannalta keskeisimmät komponentit ovat prosessori ja keskusmuisti, joka on jaettu pysyväismuistiin (ROM, Read Only Memory) ja käyttömuistiin eli työmuistiin (RAM, Random Access Memory). ROM-muisti sisältää yleensä ohjelman, jolla tietokoneen laitteisto alustetaan ja käyttöjärjestelmä 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 käsittelykelpoisessa 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 suorittaakin 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: aiheuttavatko tietokoneet uhan ihmisen älylliselle kapasiteetin säilyvyydelle samoin kuin muut koneet ovat aiheuttaneet uhan ihmisen fyysiselle kunnolle vapauttaessaan ihmisen lihastyöstä? 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 tarkoitusperää 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 järkevässä 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 algoritmissa esiintyvästä virheestä, virheellisestä syöttötiedosta tai sähköviasta. Tietokoneiden ongelmana on, etteivät ne (käskemättä) 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 nykykäsityksen mukaan ole tiedon sijainti aivoissa, vaan erilaiset assosiatiiviset 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 enemmät filosofiset pohdiskelut, ja keskittykäämme tarkastelemaan niitä matematiikan, luonnontieteiden ja korkean teknologian perusteita, jotka ovat tehneet tietokoneet mahdollisiksi.

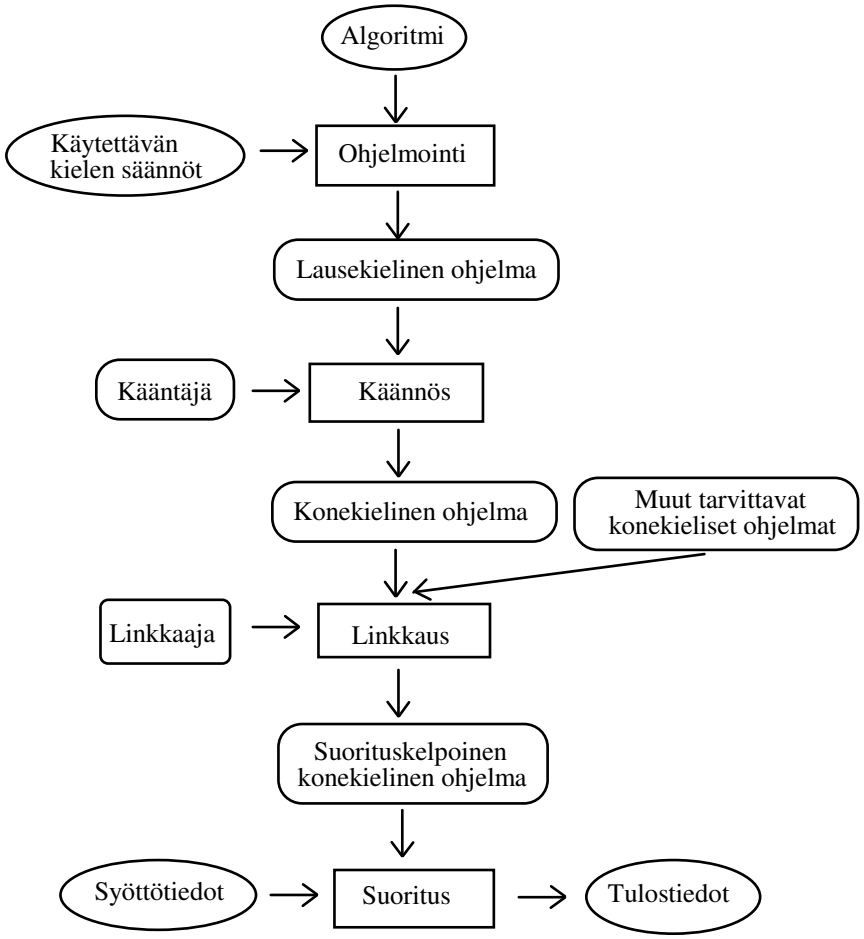
1.4 Ohjelmointikielet, ohjelmat ja ohjelmistot

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 lausekielisellä 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 tarkoitetulla 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 käännös, 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 ***systeemiohjelmistosta*** (system software). Sovellusohjelmistolla tarkoitetaan valmisohjelmia sekä käyttäjien omatekoisia ohjelmia. Systeemiohjelmistolla 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. Systeemiohjelmistoja käsitellään tarkemmin luvussa 5.

1.5 Algoritminen ongelmanratkaisu

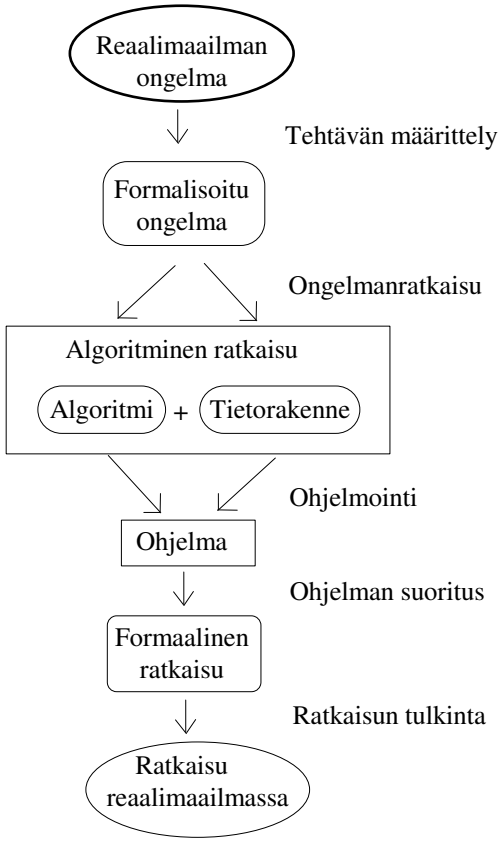
Jonkin ongelman ratkaisemiseksi laaditun algoritmin suorittaminen tietokoneella on systemaattinen toimenpide, joka voidaan yleisellä tasolla kuvata seuraavana algoritmina:

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

Vaikka algoritmin suorittaminen onkin itsessään systemaattinen, automatisoitavissa oleva toimenpide, niin algoritmien muodostaminen ei ole algoritminen toimenpide, vaan luovuutta vaativa älyllinen ponnistus (vrt. kakkureseptin kehittäminen). **Ei siis ole mahdollista kirjoittaa algoritmia algoritmin muodostamiseksi**. Algoritminen 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 reaalimaailmasta. Niinpä tehtävä on luonnollista myös kuvata reaalimaailman 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 löytyminen 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 ohjelmointikielellä. 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 reaalimaailman 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 suoritusmahdollisuuksia 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 määritykseen. 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 syt(n,m), jolla tarkoitetaan suurinta sellaista positiivista kokonaislukua, joka on sekä m:n että n:n tekijä eli jakolaskut n/syt(n,m) ja m/syt(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 määräävän 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 määräävää algoritmia. Yleisyys onkin eräs algoritmien hyvyyskriteeri.

Deterministisyysvaatimus on tiukka vaatimus, jota ei aina voida helposti saavuttaa edes tehtävänmäärittelyä 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 algoritmissa. 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 oltaisikaan 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 tehtävänmäärittelyä 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 päättyä, 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 päättyä. Näin voi käydä esimerkiksi algoritmissa 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 määräajassa. 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 Ohjelmointikielet

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 reaalilukujen (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 ohjelmointikielellä. Erilaisia ohjelmointikieliä on satoja. Ohjelmointikielten lukuisuus johtuu osittain siitä, että ohjelmointi on verrattain uusi ilmiö, eikä ole löydetty

yhteisesti hyväksyttyä parasta tapaa kirjoittaa ohjelmia. Eri sovellusalueilla käytettävät algoritmit poikkeavat merkittävästikin toisistaan, joten eräät ohjelmointikielet on suunniteltu juuri tietyntyyppisten 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

Prosessorin 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. Prosessorin tulee ymmärtää ne symbolit, joilla algoritmi esitetään, eli prosessorin tulee tuntea algoritmin esityksessä käytetyn kielen aakkosto ja kielioppi.
2. Prosessorin 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?". Ohjelmointikielet 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:nnen 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 tehtävänmäärittelyn 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 muutamaan 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 algoritmin abstraktiotaso laskee jokaisella tarkentamisella. Tehtävän ratkaisu suunnitellaan ensin hyvin korkealla abstraktiotasolla, ja tätä ratkaisua konkretisoidaan asteittain, kunnes esityksen abstraktiotaso on algoritmin 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 ratkaisuista 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 pikakaakaon 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 vesihanan alle
(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 algoritmin

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.

Esimerkkialgoritmimme 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 määrätyssä 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 yhteenveto 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 ohjelmointikielet 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 käskeä, 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*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 määritellä niiden tyyppi; esim. int x eli esitellään muuttuja x, johon voidaan tallentaa vain (tietyllä arvoalueväliillä 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 ohjelmointiajattelussa. 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 kaakaoon 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 vaihtoehtoisiin 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 vaihtoehtoisen toiminnon välillä:

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

Jos *ehto* toteutuu, suoritetaan *toiminto1*, muutoin suoritetaan *toiminto2*. Joka tapauksessa suoritetaan tarkalleen toinen vaihtoehtoisista 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* >=, *pienempi tai yhtäsuuri* <= ja *erisuuret* <>. 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>=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 yhtaikaa, 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 keskimmäinen näistä keskimmäinen ja muuttujaan suurin näistä suurimman luvun arvo. Koska prosessori voi verrata ainoastaan kahta lukua yhtaikaa, ratkaisu perustuu parittaisiin vertailuihin. Ratkaisu on nyt huomattavasti monimutkaisempi kuin edellä ollut pelkän minimin määrääminen. 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  *)
                        keskimmäinen:=y
                        suurin := z
                ELSE (*  x<y, x<z ja y>=z  *)
                        keskimmäinen := z
                        suurin := y
                ENDIF
        ELSE (*  x<y, x>=z  *)
                pienin := z
                keskimmäinen := x
                suurin := y
        ENDIF
ELSE  (*  x>=y  *)
        IF y < z THEN
                pienin := y
                IF x < z  THEN
                        keskimmäinen := x
                        suurin := z
                ELSE
                        keskimmäinen := z
                        suurin := x
                ENDIF
        ELSE
                pienin := z
                keskimmäinen := y
                suurin := x
        ENDIF
ENDIF
```

Koska IF-lauseen päättää aina ENDIF-sana, IF-lauseen THEN- ja mahdollisen ELSE-osarakenteen alku- ja loppukohdat 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 kuluvassa 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 määräämiseksi 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 ***iteraation*** 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 indefiniittiä 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, ..., 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 ohjelmointikielessä on vain alkuehtoinen WHILE-lause (kuten aina onkin), koska sen avulla voidaan toteuttaa kaikki muut toistorakenteet.*** Selvyyden 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 indefiniittiä 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>=0, kertoman (merkitään n!) määrääminen ja sen tallennus muuttujaan k. Kertoma määritellään seuraavasti: 0! = 1, n! = 1*2*...*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                              k := 1
FOR i := 1,2, ..., n DO             i  := 1
    k := k * i                      WHILE  i <= n DO
ENDFOR                                  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 aritmeettisia 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 lainkaan. Jos ehto muutetaan muotoon i<100, tuloksena on summa 1+3+5+ ... +99, jolloin silmukasta poistuttaessa i:n arvo on 101.

Esimerkki. Lukujonon maksimin määrääminen. 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,…,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 ykköstä 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,…,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,…,n–1, tiedämme, että n on alkuluku.

```
IF n = 2 THEN
        tulosta: n on alkuluku
ELSE (* nyt n>2 *)
        t := 2  (*  ensimmäinen tekijäehdokas t=2  *)
        REPEAT
                jakojäännös:= jakolaskun n/t jakojäännös  (* tässä t on enintään n–1 *)
                t := t + 1  (*  seuraava tekijäehdokas  *)
        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 tekijäehdokkaita 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 syt(x,y). Algoritmi perustuu seuraaviin tosiseikkoihin (voit ottaa ne faktana, ellet tunne matemaattisia perusteita):

1) jos x>y, niin syt(x,y) = syt(x–y,y),
2) jos x<y, niin syt(x,y) = syt(x,y–x) ja
3) jos x=y, niin syt(x,y)=x=y.

Esimerkiksi syt(3,1) = 1, syt(8,6) = 2 ja syt(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=syt(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: Nimilistaa 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:

Alkuperäinen nimilista	1. läpikäynti (4 vaihtoa)	2. läpikäynti (3 vaihtoa)	3. läpikäynti (2 vaihtoa)
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 määritellä edellistä tarkemmin, jos tiedetään, että lajiteltavia nimiä on n kappaletta. Tarkoittakoon lisäksi merkintä nimi[i] nimilistan i:nnettä nimeä. Tarvitsemme siis lukumuuttujan i, joka saa arvot 1,…,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 käyttäjäänsä 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 voidaankin 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 sangen 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 tarkoitetut 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 määritellä yksityiskohtaisemmin samaa asteittaisen tarkentamisen menetelmää käyttäen. Tarkentaminen päättyy, kun jokainen algoritmikomponentti voidaan suoraan muitta 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 osaratkaisuista alkuperäisen tehtävän ratkaisu, sanotaan ***reduktiiviseksi***. Reduktiivisessa ongelmanratkaisussa kokonaisuuden osittamisessa tehtävät rajauspäätökset ovat tärkeitä. On pyrittävä löytämään tehtävälle mahdollisimman luonteva ja hyvin määritelty 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***. Modulaarisessa 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, ajattelematta, 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 pikakaakaon keittämiseen tarkoitetusta 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 lieden 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' yksityiskodat 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 lieden 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 kärpästä yhdellä iskulla'. Kertasäästö kuitenkin kostautuu 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 määrääminen 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 tehtävänmäärittelyä 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 spesifioidaan 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 ***muodollisiksi 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*'

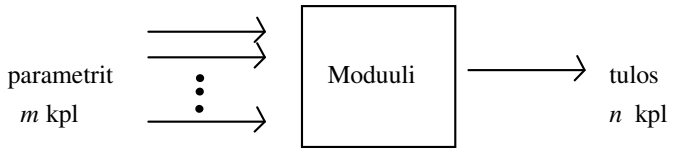
voitaisiin kirjoittaa muodossa

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.

Funktiomoduulin 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 syötteeksi. Tämä vastaa funktioteoreettista ***komposition*** käsitettä. Funktioiden kompositioihin perustuvaa ohjelmointisuuntausta sanotaan funktionaaliseksi. Funktionaaliseen ohjelmointiin palataan myöhemmin monisteen 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 ohjauskäskyt. 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 samannimisia 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-lausetta.

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,..., 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 todellisessa 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 tarkoitetut 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>=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, …, 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>=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 proseduuri, 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*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 proseduuri
        {
                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 konstruoitujen arvojen tulostamiseen, siis tulostuslaitteeseen kohdistuvaan sivuvaikutukseen. Täten moduulista on syytä tehdä proseduuri. 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 syt(3,1) = 1, ja syt(8,6) = 2. Huomaa, että yhtäsuuruusmerkin = käyttö on tässä yhteydessä matemaattisesti täysin korrektia: onhan tarkoituksena nimenomaan lausua kahden arvon, syt-funktiomoduulin 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,…,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, kilpikonnaa, 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 liikkuessa jälki, muutoin ei. Nimensä kilpikonnagrafiikka on saanut lattialla liikkuvasta robotista, jonka elektroniikkaa peittävä kuori sai sen muistuttamaan kilpikonnaa. 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 hauskalla tavalla. Konnan ohjaamiseen käytettävissä oleva käskyjoukko on pieni ja selkeä, mutta siitä huolimatta saadaan helposti piirrettyä monimutkaisia kuvioita. Tässä käytetyt komennot ovat *etene*(x), *käänny*(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 käänny(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 kynä alkutilanteessa ylhäällä.

```
MODULE neliö(sivu)
        laske kynä
        REPEAT 4 TIMES
                etene(sivu)
                käänny(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 välttämätöntä. Yksittäiset ohjelmointikielet määräävät menettelytavan tämäntapaisissa 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 proseduuri. 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 valintarakenne-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 proseduuri, jonka runkona on muutaman rivin mittainen peräkkäisrakenne tai funktio, jossa yksinkertaisesti luetellaan vaihtoehtoiset 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 (päättyvän) 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 aitaa maalaamatta DO
                suorita laudan maalaus
                siirry seuraavaan lautaan
        ENDWHILE
ENDMODULE
MODULE maalaa(aita)
        IF aitaa maalaamatta THEN
                suorita laudan maalaus
                maalaa(loput 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 ***rekursioaskeleeksi***, 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*(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 kertoma(4):n lasku sisältää aidosti kertoma(3):n laskun jne. Tämä tarkoittaa sitä, että kertoma(4):n lasku ainoastaan keskeytyy kertoma(3):n laskun ajaksi. Kun kertoma(3) on laskettu, jatketaan kertoma(4):n kesken jäänyttä laskua kertolaskulla 4 * 6 = 24. Vasta nyt kutsu 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 kertoma(i) n-nimistä parametria n_i:lla. Esimerkiksi n_2 syntyy kertoma(2):n laskun alkaessa ja se saa arvonsa kutsusta (n_2=2). Kertoma(2):n lasku keskeytyy ja n_2 passivoituu kutsun kertoma(1) suorituksen ajaksi, jolloin puolestaan parametri n_1 on käytössä. Kertoma(1):n laskun loputtua muistipaikka n_1 häviää muistista, ja kertoma(2):n lasku jatkuu operaatiolla n_2 * 1, missä 1 on kutsun kertoma(1) saama arvo. Operaation tulos 2 palautuu kutsun kertoma(2) arvoksi ja n_2 häviää. Kullakin ajanhetkellä on siis vain yksi aktiivinen parametrin n esiintymä, johon kyseiset muuttujaviittaukset kohdistuvat.

Esimerkki. Myös syt(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 yksinkertaisemman 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
= 3
```

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

```
Ota erilleen sanan ensimmäinen kirjain
Käännä sanan loppuosa takaperin
Lisää ensimmäinen kirjain käännetyn 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 proseduuri, 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 'sanan ensimmäinen kirjain' korrekti käyttö vaatii, että sanan on oltava vähintään yhden kirjaimen mittainen. Ratkaisu ei siis sovellu tyhjille 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 esitetyssä 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,...,n–1. Tarkka ratkaisu on saavutettu, kun i=n–1; viimeinen eli n. alkio ei voi yksinään olla epäjärjestyksessä, koska kaikki muut alkiot 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 reaaliluku. Iteratiivinen menetelmä reaaliluvun *a* neliöjuuren likiarvon laskemiseksi voidaan johtaa seuraavasti:

$$x = \sqrt{a}$$

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

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 määräämistä, 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≈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.

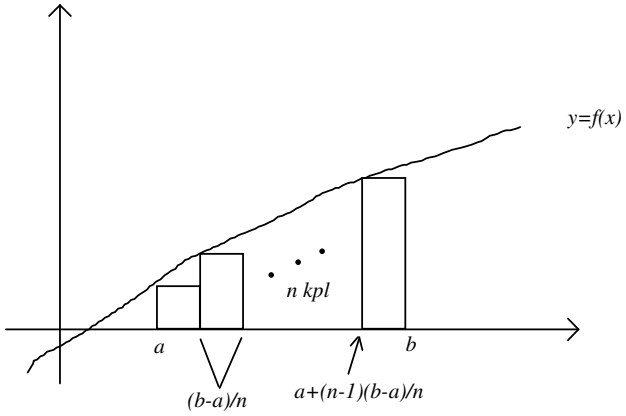
(* Alkuehto: a > 0 ja ε > 0 *)
MODULE neliöjuuri(a, ε) RETURNS $\sqrt{a}$:n likiarvo
    n := 2
    x_2 := (1+a)/ 2
    REPEAT
        $$x_{n+1} := \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$
        n := n+1
    UNTIL $x_{n-1} - x_n < \varepsilon$ (* ei tarvita itseisarvoa, koska lukujono on vähenevä *)
    RETURN x_n
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. Algoritmissa lopetetaan uusien termien laskeminen kun tämä ero on riittävän pieni, pienempi kuin sallittu virhe ε . 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 kirjoitta 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 ε > 0 *)
MODULE neliöjuuri(a, ε) RETURNS $\sqrt{a}$:n likiarvo
    nykyinen := (1+a) / 2
    REPEAT
        edellinen := nykyinen
        $$\text{nykyinen} := \frac{1}{2}\left(\text{edellinen} + \frac{a}{\text{edellinen}}\right)$$
    UNTIL edellinen – nykyinen < ε
    RETURN nykyinen
ENDMODULE

Esimerkki. Numeerinen integrointi. Iteratiivisia menetelmiä käytetään yleisesti myös numeeristen integraalien laskennassa, esimerkiksi pinta-alojen tai tilavuuksien määräämiseksi. 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 deklaratiivisessa 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 rekursiivisiksi*** (inherently recursive). Ratkaisun 'kunnollisuus' tarkoittaa tässä sitä, että vaikka jokainen rekursiivinen moduuli voidaankin muuntaa muodollisesti iteratiiviseksi – ja konekielelle käännettäessähän 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.

Yhteenvetona 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 logiikkaohjel-moinnissa.

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 reaalimaailman ongelman ratkaisua, jolloin tilannetta mallinnetaan joukolla loogisesti yhteenkuuluvia tietoja. Nämä tiedot halutaan usein koota yhteen ja esittää ne yhtenä ***tietorakenteena*** (data structure).

Imperatiiviset ohjelmointikielet ovat yleensä tyypitettyjä, 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 tyypitetyissä kielissä (esim. Java, C++ ja Pascal) ohjelmoijan tulee aina esitellä muuttuja ja ilmoittaa sen tyyppi. Esim. Javassa voidaan määritellä int x, joka tarkoittaa sitä, että x on muuttuja, johon voidaan tallentaa vain kokonaislukuarvoja. Sen sijaan esim. Python-kieli on dynaamisesti tyypittävä 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 tyypiksi tulee automaattisesti kokonaisluku (int). Näin voidaan ajatella asian olevan myös tämän monisteen pseudokielessä. Kielissä on valmiina nimetyt tyypit yksinkertaiselle tiedolle ja välineet erilaisten tietorakenteiden konstruoimiseksi. Yksinkertaisiksi tyypeiksi käsitetään yleensä kokonaisluvut (tyypin nimi Javassa: int), reaaliluvut eli desimaaliluvut (double), totuusarvot *true* ja *false* (boolean) ja merkit (char).

Tyypin 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 tyypin 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 tietotyypiksi*** (abstract data type, ADT). Termi abstrakti tarkoittaa tässä sitä, että 1) tarkastellaan mallin mukaisten alkioiden oleellisia piirteitä 2) malli on riippumaton käytettävästä ohjelmointikielestä 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 linkitetyt 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 oliokielissä 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 määritellä 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 alkiota eli jonka pituus on k. Olkoon lisäksi M 2-ulotteinen taulukko eli matriisi, jonka ensimmäisessä ulottuvuudessa (rivit) on n alkioita ja toisessa (sarakkeet) m alkiota. Matriisia M sanotaan tällöin n×m-taulukoksi. Taulukon yksittäisiin alkioihin (komponentteihin) viitataan notaatioilla V[i], missä $1 \le i \le k$, ja M[i,j], missä $1 \le i \le n$, $1 \le j \le m$. 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 4×6-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 yksiulotteiseen taulukkoon (vektoriin) T, joka sisältää esim. lukuja. Vaihtolajittelu perustuu kahden sisäkkäisen FOR-silmukan käyttöön. Ensin ensimmäistä alkiota 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 alkiota 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 kaksiulotteiseen taulukkoon Etä. Etäisyystaulukko voisi olla esimerkiksi seuraavanlainen 6×6-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
```

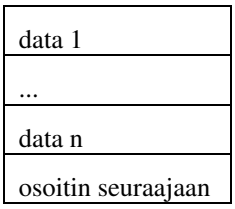
Esimerkiksi LähinKaupunki(Etä, Jyväskylä) = Pieksämäki. Muuttujan lähinmatka alkuarvoksi olisi parempi antaa joku matriisin Etä rivillä a oleva luku eikä tällaista keksittyä suurta lukua. Useissa ohjelmointikielissä taulukkoa ei voida indeksoida kaupunkien nimillä, joten tällöin joudutaan kaupungit koodaamaan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

Esimerkki. n×m-lukutaulukon Taulu kaikkien alkioiden summan laskeminen.

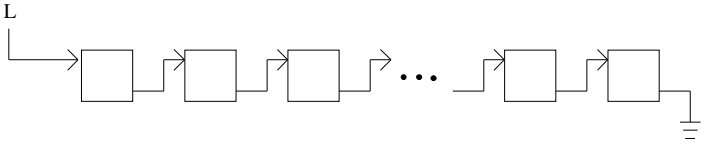
```
MODULE taulusumma(Taulu, n, m) RETURNS lukujen summa
        summa := 0
        FOR i := 1, ..., n DO
                FOR k := 1, ..., m DO
                        summa := summa + Taulu[i, k]
                ENDFOR
        ENDFOR
        RETURN summa
ENDMODULE
```

2.8.2.3 Linkitetty rakenne

Linkitetyssä 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:



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

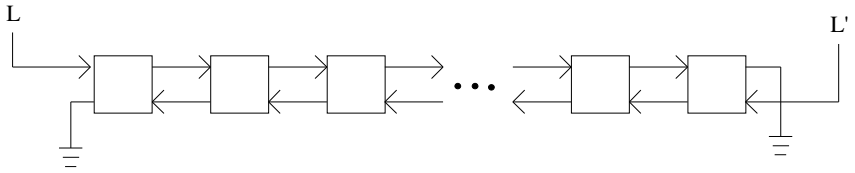


Linkitetty rakenne on rekursiivinen: jokaista alkiota seuraa rakenne, joka on alkuperäisen rakenteen kaltainen, joskin pienempi. Poikkeuksen tekee rakenteen viimeinen alkio, jolla ei ole seuraajaa. Voidaan myös sanoa, että viimeisen alkion seuraaja on tyhjä. Tätä

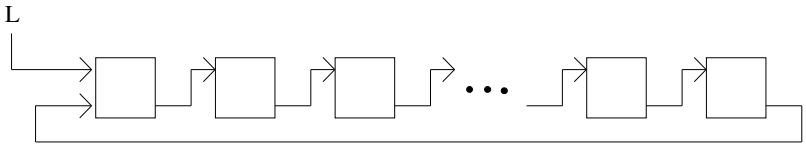
merkitään 'maadoittamalla' viimeisen alkion seuraaja-osoitin kuten yllä olevasta kuvasta näkyy.

Linkitetyistä rakenteista esiintyy useita muunnelmia käyttötarpeen mukaan. Esimerkiksi jos halutaan kulkea rakenteessa 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. Rengasmainen rakenne mahdollistaa käsittelemisen syklisesti.



- ***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 tietotyypiksi. Itse tietorakenne voidaan tallentaa muistiin halutulla tavalla (esimerkiksi jono voidaan toteuttaa käyttäen vektoria tai linkitettyä rakennetta) samoin kuin operaatiot voidaan implementoida (koodata) ohjelmoijan mieltymyksien 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 operaatiota ovat esimerkiksi: alkion poisto ja lisäys listan mielivaltaiseen kohtaan, listan tyhjyyden tarkistus, listan ensimmäisen alkion ja loppuosan palauttavat operaatiot. Lista voidaan myös määritellä 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 oliokielissä 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 alkiota, 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 i:nneksi ($1 \leq i \leq k+1$) alkioksi, 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 tuhoudu. *)
        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[1]. 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***.

[1] isä-sanan valinta perustuu sen parempaan taipuvuuteen verrattuna vanhempi-sanaan, joten kyseessä ei siis ole missään tapauksessa sukupuolinen syrjintä!

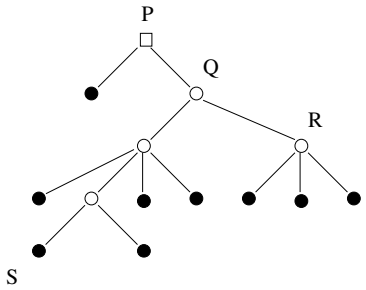
Usein sovitaan, että puun jokaisella solmulla on tarkalleen k poikapuuta, joista osa voi olla tyhjiä (puu, jossa ei ole yhtään solmua). Tällaista puuta kutsutaan ***k-haaraiseksi*** tai ***k-ariseksi*** (puun ariteetti on k) puuksi. k-haarainen puu voidaan siis määritellä rekursiivisesti myös seuraavasti:

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-tyhjien 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 täyttämättömillä 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[2] voidaan määritellä (ja sen määräävä algoritmi voidaan kirjoittaa) rekursiivisesti seuraavalla tavalla:

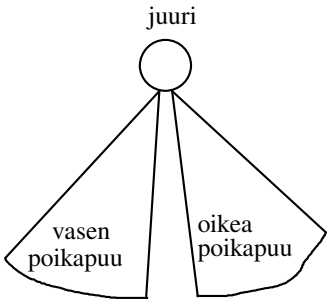
Määritelmä. Olkoon p k-haarainen puu.

- 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 $1+\max(h_1, h_2, ..., h_k)$, missä h_i on puun p i:nnen pojan korkeus.

2.8.3.4 Binääripuut

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

[2] 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 luettelemalla puun solmut tietyssä (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 vieraillaan 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=tyhjä tai p≠tyhjä), puun p oikean ja vasemman alipuun määrääminen (merkitään: p.oikea ja p.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 alkioiden tulostukseen välijärjestyksessä. Merkitään solmun p tietokentän arvoa merkinnällä p.arvo.

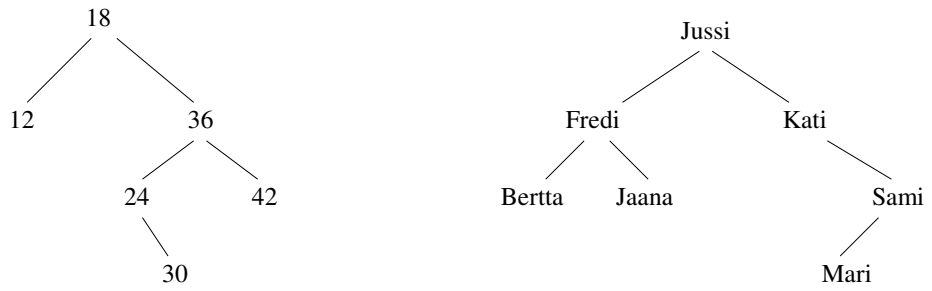
```
MODULE välijärjestys(p)
        IF p <> tyhjä THEN
                välijärjestys(p.vasen)
                tulosta(p.arvo)
                välijärjestys(p.oikea)
        ENDIF
ENDMODULE
```

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 alkioista järjestetyn binääripuun p *)
         uusilist := PuuListaksi(p)  (* konstruoi järjestetyn binääripuun p alkioista uuden järjestetyn listan *)
         RETURN uusilist
ENDMODULE
```

Moduuli ListaPuuksi: Parametri list viittaa aluksi listan alkuun, eli listan ensimmäiseen alkioon. Listan alkioista 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 lisäystarkastelu 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 alkiosta 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 siirtelemään. 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 alkiota ennen tulevat listaan kaikki sitä välijärjestyksessä edeltävät alkiot ja sen jälkeen tulevat kaikki sitä välijärjestyksessä seuraavat alkiot. 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 siirtelemään. Uusi solmu tulee listan viimeisen alkion seuraajaksi, ja sen arvoksi tulee lisättävä alkio. Muodostettavan listan ensimmäiseksi alkioksi 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 list loppuun
                PuuListaksi(p.oikea)
        ENDIF
        RETURN list
ENDMODULE
```

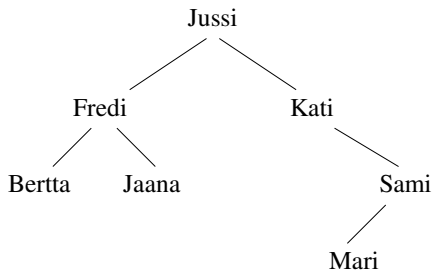
Algoritmin toteutuksen yksityiskohdat riippuvat käytettävän ohjelmointikielen mekanismeista, etenkin osoittimien (viittausten) toteutuksesta ja parametrinvälityksen yksityiskohdista. Nämä seikat sivuutetaan tässä. Edellä oleva versio kuitenkin toimii, jos käytetään arvoparametrivälitystapaa (parametrien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

2.8.3.5 Binääripuun toteutus

a) Binääripuu voidaan toteuttaa linkitettynä 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 (Fredi (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, Fredi, 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.

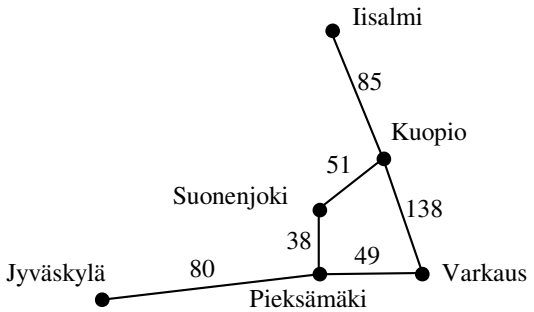
2.8.3.6 Graafi

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***[3] 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.

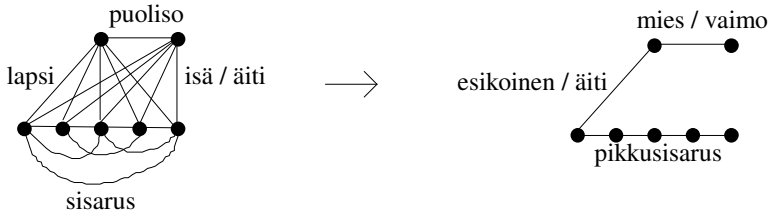
[3] puiden yhteydessä aste määriteltiin hiukan eri tavalla: puun solmun aste on sen ei-tyhjien poikapuiden lukumäärä

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 vastakkaissuuntaista 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 toki syntyä 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 'pikkusisarus'. Lisäksi sukugraafissa hyvä ratkaisu on käyttää symmetrisen relaation 'puoliso' sijasta relaatioita 'mies' ja 'vaimo':

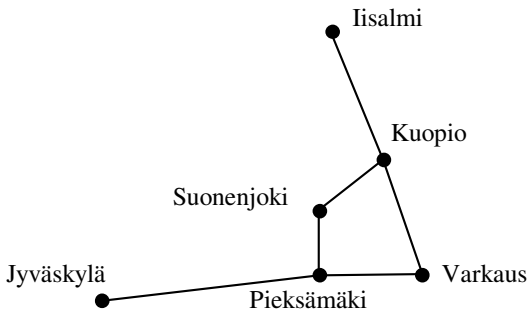


2.8.3.7 Graafin toteutus

a) 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 sarakeindekseinä, ja i. rivin j. sarakkeen alkioksi 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 aliohjelmina, 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 implementointiosa yhdessä. Näissäkin rakenteissa on itse tallennusrakenteisiin mahdollisuus viitata suoraan käyttämättä pelkästään tyypille määriteltyjä operaatioita.

Kaikista turvallisin tapa on ns. ***kapselointi*** (encapsulation), jossa tietorakenteen ja operaatioiden implementointi on täysin kätketty 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 alkioksi  *)
            IF p.pinon_koko = pinon_max_koko THEN  'virhe'
            ELSE
                p.pinon_koko:=p.pinon_koko+1
                p.alkiot[p.pinon_koko]:=a
            ENDIF
        ENDMODULE
        MODULE pop(p: Pino, a: T)
        (*  ottaa pinon p päällimmäisen alkion ja sijoittaa sen a:n arvoksi  *)
            IF p.pinon_koko=0 THEN 'virhe'
            ELSE
                a := p.alkiot[p.pinon_koko];
                p.pinon_koko := p.pinon_koko – 1
            ENDIF
        ENDMODULE
        MODULE make(p: Pino)
        (* alustaa ja luo tyhjän pinon *)
            p.pinon_koko := 0
        ENDMODULE
    ENDDEFINE
```

```
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
        ENDDEFINE
```

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 alkioksi 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 alkiota tyhjästä pinosta (p.pinon_koko=0) ja että ei viitata vektorin 'yli' (p.pinon_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 tarkoittaakin 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 tyyppikonstruktoreista riippumaton käsite).

Oliokeskeisen ohjelmoinnin keskeiset käsitteet ovat ***luokka*** (class) ja ***olio*** eli ***objekti*** (object). Luokka on staattinen käännösaikainen 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 (metodit eli moduulit), joiden avulla jokainen luokan alkio, olio, voidaan karakterisoida. Luokan käsite vastaa siis jo aiemmin käsiteltyä tyypin 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***.

Oliokielissä on yleensä valmiina mekanismit, joiden avulla voidaan helposti noudattaa – ja tulee noudattaa – edellä esitettyä kapselointiperiaatetta. Tällöin tietorakenteen ja siihen kohdistuvien operaatioiden implementointi kätketään 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 muutosmetodit: push (laita alkio pinoon) ja pop (ota alkio pinosta).

Esimerkki. Määritellään Eiffel-kielellä 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 määritellä 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 alkiota
                        ...
                        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 alkioina 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 ***periytymisen*** avulla. Periytymisellä tarkoitetaan yliluokan (superclass) olioiden käytöksen (so. metodien ja attribuuttien) siirtymistä osaksi aliluokan (subclass) olioiden käytöstä. Aliluokassa voidaan uudelleen määritellä (eli ylikirjoittaa) yliluokan attribuutteja ja metodeja ja toisaalta aliluokassa voidaan määritellä 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 tieto-objekteja 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. Olio-ohjelmoinnin keskeisin – ja myös vaativin tehtävä – onkin luokkarakenteen suunnittelu ennen ensimmäisenkään algoritmin toteutusta. Työ aloitetaankin 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***.