TURUN YLIOPISTO
AVOIN YLIOPISTO-OPETUS

# TIETOJENKÄSITTELYTIETEIDEN **PERUSOPINNOT**

## ESITTELY JA ITSEOHJAAVAN TESTIN MATERIAALI

JORMA BOBERG
Turun Yliopisto
Tietojenkäsittelytieteet

# Yleistä opinnoista ja niiden sisällöstä

Informaatioteknologia kehittyy huimaa vauhtia: ohjelmistot uudistuvat ja monipuolistuvat, tietoverkot ovat nousseet tärkeään asemaan, ohjelmointitekniikka kehittyy, organisaatioiden ja yrityksien kilpailukykyä parannetaan tietojärjestelmäratkaisujen avulla jne. Tietojenkäsittelytieteiden perusopinnoissa keskitytään mm. näiden asioiden teoreettisiin ja konkreettisiin perusteisiin.

Turun avoimen yliopiston järjestämästä tietojenkäsittelytieteiden 23 op perusopintojen sisällöstä vastaa Turun yliopiston Informaatioteknologian laitos. Opintojen sisältö vastaa yliopiston tavallista opetusta ja suoritetut opinnot ovat hyväksikäytettävissä esimerkiksi jatkettaessa opiskeluja Turun yliopistossa.

Näiden perusopintojen suoritus antaa mahdollisuuden päästä Turun yliopiston tietojenkäsittelytieteiden pääaineopiskelijaksi tai Tietotekniikan DI-koulutukseen suoraan ilman valintakoetta ns. **avoimen yliopiston väylän** kautta, jos olet ylioppilas tai sinulla on vähintään kolmivuotinen ammatillinen tutkinto.

Tietojenkäsittelytieteiden opinnot ovat teoreettisia verrattuna esim. opistotasoiseen ja ammattikorkeakoulun tietotekniikan opetukseen. Tarkoituksena on antaa perusvalmiudet uuden tiedon ymmärtämisessä alati kehittyvässä tietotekniikan maailmassa eikä ‘nappulatekniikassa’. Nimittäin kun perusasiat ymmärtää, uuden oppiminen ja itse tietokoneen hyötykäyttö (esim. uusien ohjelmistojen käyttö) sujuu itseopiskelulla. Erityisiä pohjakoulutusvaatimuksia ei ole, joskin tietokoneen peruskäyttö oletetaan tunnetuksi. Lisäksi joissakin opintojaksoissa vaaditaan paikoitellen peruskoulun matematiikan asioiden 'ymmärtämistä', mutta tarvittavat matemaattiset asiat selitetään myös kyseisessä asiayhteydessä. Myös tyydyttävä englannin kielen taito on opinnoissa eduksi.

Tietojenkäsittelytieteiden opintojen tavoitteena on antaa opiskelijoille yleiskuva tietojen-käsittelyn eri osa-alueista ja tieteenalan keskeisistä asioista ja käsitteistä. Opinnoissa käsitellään tietojenkäsittelytieteiden teoreettisia ja konkreettisia perusteita, algoritmista ongelmanratkaisua, ohjelmointia ja tietojärjestelmien mallintamista sekä tietokoneen käyttöä. Tietojenkäsittelylle on ominaista looginen ja abstrakti ajattelutapa, joten opinnoissa menestyminen edellyttää opiskelijalta pitkäjännitteisyyttä. **Opiskelussa on keskeistä itsenäinen harjoitustehtävien ratkaiseminen ja tietokoneella tehtävien töiden tekeminen**.

Kaikki tarvittava oppimateriaali kuuluu kurssimaksuun ja on verkossa.

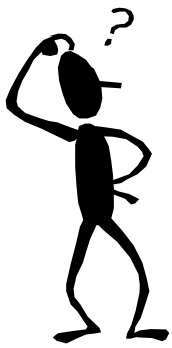
# Itseohjaava testi ja siihen liittyvä materiaali

Tämän monisteen loppuosa sisältää itseohjaavan testin materiaalin. Se antaa sinulle mahdollisuuden tutustua opintojen sisältöön ja arvioida kiinnostustasi ja osaamistasi ennen kuin aloitat tietojenkäsittelytieteiden opinnot. Menettelyllä pyritään vähentämään opintojen keskeyttämisiä. Testimateriaaliin liittyy myös testikysymykset, jonka kysymykset ovat hyvin samantyyppisiä kuin testimateriaalissa esitetyt asiat. Kysymyksissä on mukana myös yksi matemaattista osaamista (lähinnä funktioihin liittyvä) mittaava kysymys, jolla kartoitetaan matemaattisia valmiuksia. Jos et osaa vastata siihen, niin se ei suinkaan tarkoita sitä, että sinun ei kannata aloittaa opintoja. Lisäksi testissä kysytään muutamia yleisiä asioita opiskelumotivaation ja ennakkokäsitysten kartoittamiseksi. On tärkeää, että jokainen arvioi

omia kykyjään ja mielenkiintoaan tutustumalla tähän testimateriaaliin ja vastaamalla testikysymyksiin itsenäisesti. *Palauta vastauksesi tuutorillesi*. Ennen opintoja oppilaitoksellasi järjestetään infotilaisuus, jossa selvitetään opintojen sisältöä ja puhutaan myös tästä materiaalista ja itseohjaavasta testistä. **Selvitä infotilaisuuden ajankohta oppilaitokseltasi.**

Kuten opintojaksojen kuvauksista käy ilmi, algoritmit ovat hyvin keskeisessä asemassa tietojenkäsittelytieteissä. Algoritmien ymmärtäminen on välttämätöntä opintoja suoritettaessa, joten oheinen itseohjaavan testin materiaalikin koostuu lähinnä algoritmeista ja niiden suunnitteluun ja laatimiseen liittyvistä asioista (eli ohjelmoinnista). Tietenkin tietojenkäsittelytieteiden opintoihin sisältyy paljon muitakin asioita, joten **tämä materiaali ei anna koko kuvaa tietojenkäsittelytieteiden perusopintojen sisällöstä, mutta se antaa kuitenkin kuvan niistä asioista, joita yleensä pidetään opintojen vaikeina asioina.**

Oheinen materiaali on pääosin ensimmäisen opintojakson sisällöstä, joten nämä asiat käsitellään perusteellisemmin opintojen aikana kyseisen kurssin yhteydessä tekemällä aiheeseen liittyviä harjoitustehtäviä. Lisäksi jaksolla havainnollistetaan esim. tässä materiaalissa esitettyjä algoritmeja esimerkkien avulla (se siis sisältää paljon ns. 'rautalankaa', jota tässä materiaalissa ei juuri ole). Ennen näiden asioiden käsittelyä tutustutaan tietojenkäsittelytieteiden perusasioihin ja käsitteisiin. Onkin syytä korostaa, että **pelkän tämän materiaalin nojalla esitettyjä asioita ei ehkä pysty täysin ymmärtämään**, joten esitetyt asiat saattavat tuntua vaikeilta. **Itseohjaavan testin jotkut kysymykset ovat vasta-alkajalle hankalia, mutta tartu niihin rohkeasti ja ennakkoluulottomasti tutkimalla testimateriaalia.**



… ja sitten alkaa varsinainen testimateriaali

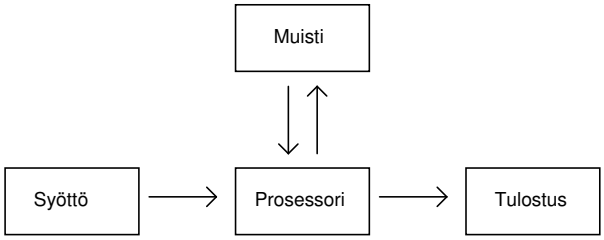
# 1. Johdanto

## 1.1. Tietokoneet ja algoritmit

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. 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 suoritettavaksi 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***.

***Tietokone*** (engl. computer) on kone, joka voi ratkaista hyvin määriteltyjä 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*** (algorithm). Algoritmi on siis menetelmä, jonka mukaisesti tehtävä suoritetaan. 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). Prosessi vaatii aina ***resursseja*** (resource), jollaisia ovat erityisesti laskentaan kuluva aika sekä erilaiset laitteistoresurssit, kuten muistitila. Resurssien tarve vaihtelee algoritmeittain ja prosessoreittain.

Tietokoneen pääkomponentit ovat ***prosessori*** eli ***keskusyksikkö*** (central processing unit, CPU) ja ***muisti*** (memory) sekä erilaiset ***siirräntä***- eli ***syöttö- ja tulostuslaitteet*** (input and output devices, I/O devices). Keskusyksikkö 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*** 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 rakenteeseen tutustutaan opintojaksolla Tietojenkäsittelyn perusteet II.




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. Ohjelman lauseiden luonne on ohjelmointikielikohtainen. Usein yksittäisen lauseen suoritus vaatii toisen, erikseen määritellyn algoritmin noudattamista.

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 opintojaksolla Tietojenkäsittelyn perusteet II.

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.

## 1.2. Tietokonejärjestelmät

***Tietojärjestelmällä*** tarkoitetaan jonkin toiminnon toteuttamiseksi muodostetun organisaation tietojenkäsittelytarpeen tyydyttämiseksi kehitettyä (automatisoitua) järjestelmää, joka koostuu ohjelmiston (software) ja laitteiston (hardware) muodostaman ***tietokonejärjestelmän*** lisäksi ***inhimillisestä komponentista*** (human ware). Vaikka myös ihmisen toiminnan osuutta järjestelmässä voidaan ainakin osittain tutkia ja mallintaa tieteellisesti mm. systeemiteorian keinoin, keskitytään tässä algoritmeihin ja tietokoneisiin ja niiden ominaisuuksiin.

Yleensä kun puhutaan jonkin tehtävän tekemisestä tietokoneella, tarkoitetaan juuri tietokonejärjestelmää. Tietokonejärjestelmä voidaan esittää ohjelmisto- ja laitteistokomponenttien hierarkiana. Se muodostuu ***sovellusohjelmistosta*** (application software), ***systeemiohjelmistosta*** (system software) ja ***tietokonelaitteistosta*** (computer hardware). Sovellusohjelmistolla tarkoitetaan valmisohjelmia (esim. tekstinkäsittely) 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. Tietokonelaitteistolla puolestaan tarkoitetaan fyysisiä laitteita, kuten keskusyksikkö, muisti ja siirräntälaitteet. Tietokonelaitteistoja käsitellään lähemmin opintojaksolla Tietojenkäsittelyn perusteet II.

Korkean tason algoritmit esitetään yleensä ohjelmana, mutta matalan tason algoritmit voidaan myös rakentaa fyysisten komponenttien sisään. Esimerkiksi kertolaskualgoritmi voidaan toteuttaa joko ohjelmisto- tai laitteistotasolla. Systeemiohjelmana se voidaan toteuttaa esimerkiksi yhteenlaskua käyttävällä tavanomaisella kynällä ja paperilla allekkain -algoritmilla (ns. koulualgoritmilla), ja laitteistollisesti se voidaan toteuttaa keskusyksikön sisäisillä digitaalielektroniikan komponenteilla, joiden konfiguraatio heijastaa algoritmin rakennetta. On myös mahdollista toteuttaa algoritmi osittain laitteiston ja osittain ohjelmiston avulla. Algoritmin toteutustavan valintaan vaikuttavat mm. kustannus- ja hyötytekijät. Yleensä monimutkaiset toiminnot (esim. kielen kääntäminen) toteutetaan niiden laajuuden ja tavoitteena olevan ylläpidon helppouden takia ohjelmistollisesti ja yksinkertaiset perustoiminnot (esim. peruslaskutoimitukset) laitteistollisesti tehokkuuden vuoksi.

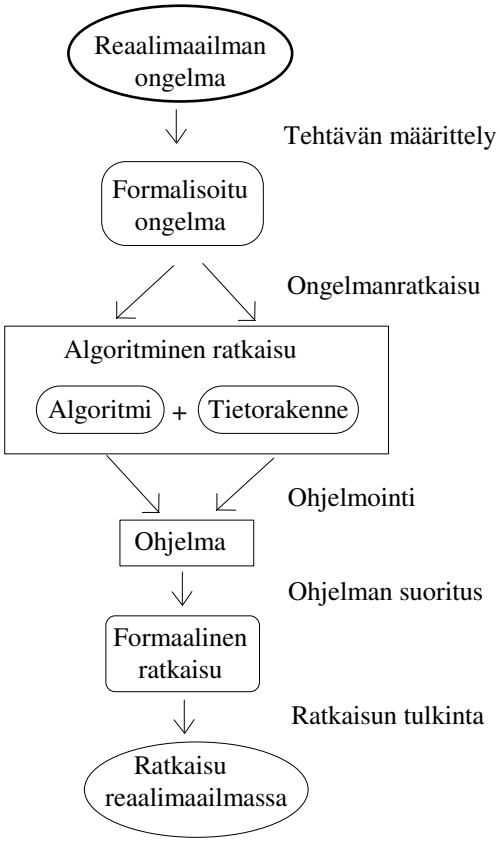
## 1.3. Algoritminen ongelmanratkaisu

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. seuraavalla sivulla oleva kuva).

Ratkaistava ongelma 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 (eli tarkan) 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 kovinkaan hyvin, joutuu tietojenkäsittelyalan ammattilainen usein aloittamaan saamansa toimeksiannon tehtävänmäärittelyn tarkentamisella.




**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-laitteisto -kokonaisuuksien hallinnan.

# 2. Algoritmit ja niiden 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.

Algoritmilta vaaditaan seuraavat ominaisuudet:

- **Yleisyys**: algoritmin on sovelluttava määritellyn tehtävän kaikkiin tapauksiin.
- **Deterministisyys**: algoritmin askeleet on oltava yksikäsitteisesti määritelty ja algoritmin jokaisessa 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
  - **terminoituvuus**: algoritmi todella antaa aina tuloksen, ts. sen suoritus päättyy aina.

Yleisyysvaatimus tarkoittaa, että algoritmi todella ratkaisee asetetun tehtävän kaikissa tehtävänmäärittelyä vastaavissa tapauksissa eli antaa oikean tuloksen kaikilla mahdollisilla syötteillä. Deterministisyys vaatii, että toimenpiteiden suoritusjärjestys on aina kerrottava algoritmissa. 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.

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, ts. toimiiko algoritmi oikein kaikilla mahdollisilla syötteillä? 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. 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. On kuitenkin olemassa päättyväksi tarkoitettuja algoritmeja, joiden suoritus ei välttämättä aina pääty. Näin voi käydä esimerkiksi algoritmissa olevan virheen takia. Algoritmien oikeellisuutta ja terminoituvuutta tarkastellaan lähemmin opintojaksolla Tietojenkäsittelyn perusteet II.

## 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 (eli yhteydestä)
- 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 konstruktioi-

den kieliopillisesta oikeellisuudesta huolehtimiseen. Luonnollisen 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. Luonnollinen kieli on rikasta, joskin liian epämääräistä ja tarpeettoman monimutkaista.

Maailmankuvan tarve tekee luonnollisen kielen käsittelystä tekoälyongelman: reagoidakseen oikein koneen täytyisi ymmärtää, mitä tekstissä sanotaan. Useinkaan tekstissä ei sanota läheskään 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 esitettävä formaalisella kielellä, eli ohjelmana, joka on kirjoitettu jollakin ohjelmointikielellä. Enemmän tai vähemmän 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. Eräät ohjelmointikielet onkin suunniteltu juuri tietyntyyppisten algoritmien esittämiseen. 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. Aakkoston kielioppisäännöt ovat täsmällisiä ja riittävän yksinkertaisia, jotta tietokone pystyy tehokkaasti tulkitsemaan ohjelmat.

### 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, ainakin kokonaisuudessaan. (Tämä riippuu ohjelmointiympäristöstä.)

**Esimerkki**. Syntaksi.

> a + = b (väärin ohjelmointikielessä Pascal, mutta oikein Javassa ja 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.** Semantiikka ja syntaksi.

> 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, mutta toiset semanttiset virheet havaitaan vasta 3. vaiheessa. Esimerkiksi lauseen "Kirjoita vuoden n:nnen kuukauden nimi" semanttista järkevyyttä ei voida tarkistaa, 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. Ohjelmointikielten kielioppeja tarkastellaan opintojaksolla Tietojenkäsittelyn perusteet II.

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. 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 sellaisenaan 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.

Tarkastellaan esimerkkinä tehtävää, jossa on suunniteltava pikakaakaonkeittoalgoritmi 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

(3.1) Nosta vesipannu kupin yläpuolelle
(3.2) Kallista pannua niin, että vesi valuu kuppiin
(3.3) Odota, kunnes kuppi on täynnä
(3.4) Suorista pannu vaakasuoraan
(3.5) Laske vesipannu liedelle

Edelleen jotkut 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

jne.

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.

# 3. Imperatiivinen ohjelmointi

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 eli ***proseduureja***. Proseduurin merkitys on siinä, että sen suorituksen avulla algoritmille annetuista syötteistä saadaan halutut tulosteet.

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 (käskyt kirjoitetaan vain peräkkäin). Sen sijaan käskyjen peräkkäisyydestä poikkeava suoritusjärjestys (vaihtoehtoiset toiminnot, toimenpiteiden toistaminen) osoitetaan algoritmissa erityisin rakentein, joita käsitellään tässä luvussa.

## 3.1. Muuttuja, lause, lauseke ja asetuslause

Imperatiivinen algoritmi keskittyy kuvaamaan sitä prosessia, jonka tuloksena halutut tulosteen syntyvät. Prosessin etenemisen apuna käytetään ***muuttujia***, tietyn merkityksen omaavia arvoja, jotka muuttuvat prosessin aikana. Imperatiivisessa paradigmassa muuttuja tarkoittaa tietokoneen muistipaikkaa. Algoritmin käskyt muuttavat muistipaikkojen sisältöä ja prosessin tila määräytyy muuttujien kulloistenkin 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 siis 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 nykyään tilanne on päinvastoin. Ei-imperatiivisten paradigmojen tärkein etu on 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*** (sentence, 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***

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. Yksinkertaisimmillaan lauseke voi olla vakio tai muuttuja (esim. x:=0 tai x:=y), mutta se voi sisältää useita laskutoimituksia, muuttujaviittauksia ja vakiota (esim. x:=2*x+3*y). 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. Tässä on hyvä huomata, että monissa

ohjelmointikielissä (esim. Java ja C) käytetään asetusoperaattorin := sijasta yhtäsuuruutta = (joka on huono valinta).

**Esimerkki**. Oletetaan, että muuttujissa a, b ja c on seuraavat arvot a=2, b=11 ja c=5. Tarkastellaan seuraavien peräkkäin suoritettavien lauseiden vaikutusta näiden muuttujien arvoihin.

```
a := b
c := c*b+a
```

Lauseen a := b jälkeen vain muuttujan a arvo on muuttunut; a:n uusi arvo on 11.
Lauseen c := c*b+a jälkeen vain c:n arvo on muuttunut; c:n uusi arvo on 5*11+11=66.
Tässä c*b+a on (aritmeettinen) lauseke, jolla on lukuarvo. Ohjelmointikielissä kertolasku ilmaistaan merkillä *, vaikka normaalisti matematiikassa sitä ei merkitä näkyviin. Jakolasku ilmaistaan merkillä /. Laskujärjestys on koulusta tuttu: lauseketta lasketaan vasemmalta oikealle niin, että * ja / ovat vahvempia kuin + ja -. Esimerkiksi lausekkeessa a+b*c lasketaan ensin b*c. Lisäksi lausekkeessa voidaan käyttää sulkeita ilmaisemaan laskujärjestystä; esim. lausekkeessa c*(b+a) lasketaan ensin b+a, joka kerrotaan c:llä.

Tähän mennessä ei ole juurikaan puututtu siihen millaista tietoa algoritmeissa käsitellään, esimerkiksi millaista tietoa voidaan asettaa muuttujan arvoksi. Tässä vaiheessa sovitaan, että käsiteltävä tieto voi olla numeerista (kokonaisluku tai desimaaliluku), tekstiä, totuusarvo (tosi ja epätosi) tai lista (jono peräkkäisiä alkioita).

## 3.2. Ohjausrakenteet ja lauseet

Algoritmi on peräkkäisten toimenpidekokonaisuuksien eli lauseiden jono. Nämä lauseet (eli käskyt) suoritetaan yksi kerrallaan siinä järjestyksessä kuin ne on kirjoitettu. Viimeisen lauseen suoritus päättää algoritmin suorituksen.

Algoritmin perusrakenteet ovat ***peräkkäisyys***, ***valinta*** ja ***toisto***, joita kutsutaan myös ***ohjaus***- tai ***kontrollirakenteiksi***. Algoritmit rakennetaan näitä perusrakenteita yhdistelemällä. Ohjausrakenteet peräkkäisyys, valinta ja toisto riittävät minkä tahansa algoritmin esittämiseen.

**Jokainen algoritmi kirjoitetaan käyttäen asetus-, valinta- ja toistolauseita**. Asetuslause esiteltiin jo aiemmin. Seuraavassa esitetään tässä monisteessa käytetyt valinnan (haarautumisen) toteuttavat valintalauseet **IF**...**THEN**... ja **IF**...**THEN**...**ELSE** sekä toiston toteuttavat toistolauseet **FOR**, **WHILE** ja **REPEAT**...**UNTIL**, jotka edustavat ohjelmointikielissä yleisesti käytettyjä lauserakenteita (syntaksi kuitenkin vaihtelee eri ohjelmointikielissä).

Seuraavassa kerrotaan tarkemmin näiden lauseiden **syntaksi** ja **semantiikka**. Lauseen syntaksin esityksessä lihavoidut sanat (ns. kielen varatut sanat) ovat kuuluvat aina lauseeseen (ja tulee olla aina mukana lauseessa), mutta muu osa on kuvailevaa ja näihin kirjoitetaan kuvauksen mukaiset osat. Näissä osissa käytetään ilmauksia (kursivoitu) *ehto*, *toiminto*, *toiminto1* ja *toiminto2*. *Ehto* on totuusarvoinen lauseke, ja *toiminto* koostuu yhdestä tai useammasta lauseesta, joista kukin on asetus-, valinta tai toistolause.

### 3.2.1. Peräkkäisyys

**Esimerkki**. Matka Joensuusta Uppsalaan.

Aja taksilla Joensuun rautatieasemalle
Nouse Turkuun menevään junaan
Jää junasta Turun satamassa
Astu Tukholmaan menevään laivaan
Nouse maihin Tukholman satamassa
Kävele metroasemalle
Aja metrolla rautatieasemalle

Nouse Uppsalaan menevään junaan
Jää junasta Uppsalassa

Algoritmeja ei pystytä yleensä esittämään peräkkäisillä käskyillä, vaan algoritmin yleisyysvaatimuksen mukaisesti on kyettävä toimimaan eri tilanteissa. Mitä tapahtuisi aiemmin esitetyssä 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?

Myöskään matkustusesimerkin kohdalla peräkkäisyys ei ole riittävä. Miten menetellä, jos taksista puhkeaa rengas, juna myöhästyy satamasta, laiva ajaa karille tai terroristiryhmä päästää metroasemalle hermokaasua?

### 3.2.2. Valinta(lauseet)

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 eli perustua johonkin yksiselitteiseen valintaehtoon. Epädeterministisiin ongelmia ja niiden ratkaisemista käsitellään jonkun verran opintojaksolla Tietojenkäsittelyn perusteet II.

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

**IF** *ehto* **THEN** *toiminto* **ENDIF**

Valintarakenteen *ehto* on totuusarvoinen lauseke (jonka arvo on tosi tai epätosi) ja se määrää tilanteen, jossa *toiminto* suoritetaan. Jos *ehto* pitää paikkansa, *toiminto* suoritetaan. Muussa tapauksessa suoritusta jatketaan IF–THEN -rakennetta 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 siis suoritetaan tarkalleen toinen vaihtoehtoisista toiminnoista, minkä jälkeen jatketaan IF–THEN–ELSE -rakennetta seuraavasta lauseesta.

*ehto* on siis totuusarvoinen lauseke (esim. a<b), mutta se voi sisältää myös loogisilla operaattoreilla (ne kirjoitetaan ’meidän kielessä’ kuvaavasti: AND, OR ja NOT) yhdistettyjä totuusarvoisia lausekkeita. Ohjelmointikielien syntaksi 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 ja b<c), niin kyseinen lauseke on tosi vain jos kumpikin lauseke on tosi. Vastaavasti looginen 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. Lausekkeissa voi käyttää tarvittaessa myös sulkeita osoittamaan laskujärjestyksen.

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. Lauseissa esiintyneet *toiminto*, *toiminto1* ja *toiminto2* voivat olla mitä tahansa

perusrakenteista muodostettuja toimenpidekokonaisuuksia; erityisesti ne voivat sisältää toisia valintarakenteita, kuten seuraavasta esimerkistä näemme.

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 ohjelman toimintaan. 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. alla olevat esimerkit). Huolellisesta esityksestä ja sisennyksistä huolimatta mutkikkaat valintarakenteet tekevät algoritmista helposti epäselkeän.

**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
EDIF
```

**Esimerkki**. 1) Tarkastellaan seuraavaksi tilannetta, jossa meillä on kaksi lukumuuttujaa x ja y, joihin on talletettu 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
```

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
```

### 3.2.3. Toisto(lauseet)

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).

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 ***määrätyksi*** eli ***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 etukäteen, ennen toistorakenteen suorituksen aloittamista. Tällöin puhutaan ***määräämättömästä*** eli ***indefiniittisestä toistosta***. Tällainen dynaaminen toistorakenne, jossa toistojen lukumäärä määräytyy toiston aikana, lisää aidosti kielen ilmaisuvoimaa.

Etenkin indefiniittiä toistorakennetta käytettäessä on algoritmissa 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)

Seuraavassa oletetaan, että n on lukumuuttuja tai lauseke, jonka arvo on kokonaisluku. Yksinkertainen definiitti toisto on muotoa:

**REPEAT** n **TIMES**
    *toiminto*
**ENDREPEAT**

jossa *toiminto* suoritetaan n (n on muuttuja tai lauseke, jolla on kokonaislukuarvo) kertaa.

Ns. askeltava toisto on muotoa:

**FOR** jokaiselle listan L alkiolle **DO**
    *toiminto*
**ENDFOR**

jossa *toiminto* suoritetaan kerran kutakin järjestetyn listan L alkiota kohden. Jos lista on tyhjä, toimintoa ei suoriteta kertaakaan eli koko toistolause sivuutetaan. Toiminnossa voidaan lisäksi viitata (ja tyypillisesti viitataankin) vuorossa olevaan listan alkioon, joten toistettava *toiminto* voi kohdistua eri objekteihin eri toistokerroilla. Joskus 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 kerralla i=2 jne. ja tyypillisesti toiminnossa käytetään hyväksi *silmukkamuuttujan* i arvoa.

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

Indefiniitit toistorakenteet voidaan jakaa kahteen osaan. ***Alkuehtoinen*** toisto (pre-tested loop):

**WHILE** *ehto* **DO**
    *toiminto*  (* silmukan runko *)
**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. Näin ollen toistorakenteen rungossa tulee tehdä sellaisia toimenpiteitä, jotka vaikuttavat ehdon voimassaoloon.

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

**REPEAT**
    *toiminto* (* silmukan runko *)
**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. Lopetusehto toimii siis päinvastoin kuin WHILE-rakenteessa.

Definiitti toisto voidaan aina korvata indefiniitillä toistolla, joko alkuehtoisella tai usein myös loppuehtoisella toistolla. Päinvastainen ei luonnollisestikaan pidä paikkaansa. 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, ...) sijasta lauseiden yhteenkuuluvuus ilmoitetaan ohjelmointikielissä eri tavoin: esimerkiksi sulkeilla '{' ja '}', tai sanoilla BEGIN ja END tai vain sanalla END.

**Esimerkki.** Esimerkkejä definiitin toiston käytöstä.

**FOR** jokainen rengas **DO** Tarkista ilmanpaine **ENDFOR**

**FOR** päivä := maanantai, ..., sunnuntai **DO** tee päivänaskareet **ENDFOR**

**FOR** kirjain := a, ..., ö **DO**
    **IF** kirjain on a, e, i, o ,u, y, a, ä tai ö **THEN** tulosta: kirjain on vokaali
    **ELSE** tulosta: kirjain on konsonantti
    **ENDIF**
**ENDFOR**

**Esimerkki**. Osoitteenhakualgoritmi. Olkoon annettuna henkilön nimi, jonka osoite tulisi hakea listasta, joka sisältää nimiä ja osoitteita. Vaikka listan nimien määrä tunnettaisiin, ei tarvittavien toistokertojen määrää tiedetä, koska ei tiedetä, monesko nimi haettava nimi on listassa (jos ollenkaan). On siis käytettävä indefiniittiä toistoa. Käytetään alkuehtoista toistoa, jossa toiston loppumista testaava ehtolauseke koostuu kahdesta ehdosta, jotka on yhdistetty AND-operaatiolla. Tämä yhdistetty ehtolauseke on tosi, jos kumpikin ehdoista on tosi. Alla esitetty algoritmi esitetään hyvin korkealla tasolla eikä se siis vastaa tarkasti todellista ohjelmaa.

**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≥0 (tällaisia lukuja sanotaan matematiikassa *luonnollisiksi* luvuiksi) kertoman (merkitään n!) määrääminen. Kertoma määritellään seuraavasti: 0! = 1, n! =1*2*...*n, n > 0. Esimerkiksi 5! = 1*2*3*4*5 = 120. Oletetaan, että ennen alla olevien algoritmien suorittamista muuttujassa n on jokin ei-negatiivinen kokonaisluku. Algoritmin suorituksen laskun tulos eli n!:n arvo on lopuksi muuttujassa k. Algoritmi esitetään sekä FOR- että WHILE-rakenteella hyvin tarkalla tasolla.

```
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
```

Edellä <= tarkoittaa: pienempi tai yhtäsuuri. Tehtävä: Seuraa oikeanpuoleisen algoritmin toimintaa kirjoittamalla muuttujien i, n ja k arvot näkyviin taulukon muodossa olettamalla, että muuttujassa n on aluksi luku 4. Mikä arvo on muuttujassa k ja i silmukan suorituksen jälkeen? Algoritmi voidaan kirjoittaa myös seuraaavassa muodossa, jossa kertominen suoritetaan päinvastaisessa järjestyksessä alkaen n:stä.

```
k := 1
WHILE n>1 DO
        k := k * n
        n := n - 1
ENDWHILE
```

Tässä on kuitenkin houmattava, että algoritmi muuttaa muuttujan n arvoa. Jos muuttujan n alkuperäinen arvo halutaan säilyttää, niin silloin tulee käyttää apumuuttujaa kuten edellisessä ratkaisussa.

**Esimerkki.** Lukujonon suurimman luvun määrääminen. Maksimiarvo talletetaan muuttujaan maksimi ja lisäksi algoritmi käyttää (apu)muuttujia maksimiehdokas ja seuraava.

```
maksimiehdokas := lukujonon ensimmäinen luku
REPEAT
        seuraava := lukujonon seuraava luku
        IF seuraava > maksimiehdokas THEN maksimiehdokas := seuraava ENDIF
UNTIL lukujono on 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
```

Ohjelmointikielissä viitataan taulukon tiettyyn alkioon indeksin avulla. 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
ENDWHILE
```

**Esimerkki.** Alkulukutesti (vaikeahko). Algoritmi tulostaa tiedon siitä, onko annettu kokonaisluku n, n>1, alkuluku. Alkulukuja ovat ykköstä suuremmat kokonaisluvut, jotka ovat jaollisia vain ykkösellä ja itsellään; esim. 11 on alkuluku, mutta 10 ei ole. Algoritmi käyttää apumuuttujia t ja jakojäännös. Algoritmin idea on seuraava: luku n jaetaan luvuilla 2,3, ..., n (muuttuja t) toistuvasti, kunnes jako menee jossain vaiheessa tasan (jakojäännös=0) tai jakaja on edennyt lukuun n saakka. Algoritmi olettaa, että voimme tutkia mikä on jakolaskun jakojäännös. Jos jako menee jossain vaiheessa tasan, niin tiedämme, että n ei ole alkuluku. Muussa tapauksessa n on alkuluku.

```
IF n = 2 THEN
        tulosta: n on alkuluku
ELSE
        t := 2   (*  ensimmäinen tekijäehdokas t=2  *)
        REPEAT
                jakojäännös:= jakolaskun n/t jakojäännös
                t := t + 1   (*  seuraava tekijäehdokas  *)
        UNTIL jakojäännös=0 OR t ≥ n
        (*  silmukan loputtua joko jakojäännös=0, jolloin n ei ole alkuluku tai t=n, jolloin n on alkuluku  *)
        IF jakojäännös = 0 THEN
                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 (ns. looginen tai), joka toteutuu, jos jompikumpi tai molemmat osat toteutuvat. Tämän yhdistetyn ehdon toteuduttua on vielä testattava, kumpi ehdon osa aiheutti yhdistetyn ehdon toteutumisen.

**Esimerkki.** Tarkastellaan seuraavaa esimerkkiä toistorakenteesta, jonka opetuksena on se, että toiston jatkuvuutta määrittelevä ehto (tässä tulee i ≠ 100) tulee laatia harkiten.

```
i := 1
summa := 0
WHILE 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.

### 3.2.4. Lajitteluesimerkki

Seuraavassa esimerkissä tarkastellaan esitettyjen ohjausrakenteiden ja ***asteittain tarkentavan suunnittelumenetelmän*** käyttöä lajittelualgoritmissa, joka järjestää eli 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 aina alusta saakka, kunnes listassa ei tarvinnut tehdä enää yhtään vaihtoa. Tällöin lista on aakkosjärjestyksessä. 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 (otetaan käyttöön apumuuttuja nimi):

```
WHILE lista ei ole lajiteltu DO
    nimi := listan ensimmäinen nimi
    REPEAT
        IF nimi ja sen seuraaja ovat väärässä järjestyksessä
        THEN  vaihda nimien paikkaa
        ENDIF
        nimi := listan seuraava nimi
    UNTIL lista on käyty loppuun (* kahta viimeistä nimeä on jo verrattu keskenään *)
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   (*  siirrä i seuraavan nimen kohdalle *)
    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  *)
```

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.

## 3.3. Modulaarisuus

Tässä kappaleessa tarkastellaan modulaarisuusperiaatetta, jonka ymmärtäminen on abstraktin, käsitteellisen ajattelutavan omaksumisen ohella 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.

### 3.3.1. 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 kulloisen tarkastelun kannalta merkitykselliset yksityiskohdat on voitava esittää sen avulla. Itse asiassa on vain hyvä, jos 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 mallissa riippuu kulloisesta näkökulmasta ja siitä yhteydestä, missä ja mihin tarkoitukseen mallia käytetään.

Tietojenkäsittelytieteessä abstrahointia käytetään jatkuvasti kaikilla osa-alueilla. Esimerkiksi ***koneabstraktioin*** kuvataan erilaisia tietokonearkkitehtuureja, tietokoneen fyysistä organisaa-

tiota. Tällä tarkoitetaan tarvittavien perusoperaatioiden suorittamiseksi vaadittujen tietokoneen komponenttien fysikaalista toteutusta ja komponenttien konfigurointia toisiinsa nähden. Koneabstraktiolla voidaan myös kuvata koneen loogista toimintaa. Näitä asioita tarkastellaan lähemmin opintojaksolla Tietojenkäsittelyn perusteet II.

### 3.3.2. Algoritmimoduulit ja niiden parametrisointi

Aiemmin kuvattiin, miten algoritmeja voidaan suunnitella asteittain tarkentavalla menetelmällä. Jokaisessa tarkennusaskeleessa algoritmi jaetaan pienempiin komponentteihin, osa- eli 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. Itsenäistä algoritmikomponenttia, joka voidaan suunnitella käyttöyhteydestä riippumattomasti ja liittää mihin tahansa algoritmiin, jossa kyseinen osatehtävä esiintyy, kutsutaan ***algoritmimoduuliksi*** (algorithm module) tai seuraavassa lyhyesti vain ***moduuliksi*** (ohjelmointikielissä käytetään myös nimityksiä aliohjelma, metodi tai rutiini).

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.

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 pikakaakon keittämiseen tarkoitetusta algoritmista näytti seuraavalta:

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 käy hyvin vaikeaksi. Modulaarista esitystä käyttäen voimme yhdistää ristiriitaiset tavoitteet: esittää algoritmin riittävän yksityiskohtaisella tasolla, jotta robotti osaa sen suorittaa, mutta samalla säilyttää kuvauksen niin korkealla abstraktiotasolla, että ihminenkin sen ymmärtää.

Moduuli 'Valmista pikakaakaota'
Suorita moduuli ‘Keitä vettä’
Suorita moduuli ‘Pane kaakaojauhoja kuppiin’
Suorita moduuli ‘Kaada vettä kuppiin’

Moduuli 'Keitä vettä'
Suorita moduuli ‘Täytä vesipannu’
Aseta pannu liedelle
Suorita moduuli ‘Keitä liedellä pannussa oleva vesi ’
Siirrä pannu lieden sivuun

Moduuli 'Täytä vesipannu'
Aseta vesipannu vesihanan alle
Avaa vesihana
Odota, kunnes pannu on täynnä
Sulje vesihana

Moduuli ‘Keitä liedellä pannussa oleva vesi '
Pane liesi päälle
Odota, kunnes vesi kiehuu
Pane liesi pois päältä

Moduuli 'Pane kaakaojauhoja kuppiin'
Suorita moduuli ‘Ota kaakaopurkki esiin’
Suorita moduuli ‘Pane 3 lusikallista kaakaojauhetta kuppiin’
Pane lusikka kuppiin
Suorita moduuli ‘Pane kaakaopurkki pois’

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

jne.

Moduulien esittämisessä on oleellista se, minkälaisia perustoimintoja robotti osaa tehdä. Edellä oletetaan, että robotti osaa esim. avata purkin kannen, mutta komentoa ‘ota kaakaopurkki esiin’ se ei ymmärrä, se on sille liian abstrakti. Samaa koskee myös tietokoneen ohjelmointia: meidän tulee tietää käytettävän kielen perusrakenteet ja -operaatiot kyetäksemme laatimaan ohjelman jonkun tehtävän suorittamiseksi. Kaikissa imperatiivisessa (siis useimmissa) ohjelmointikielissä suoritusta kontrolloidaan peräkkäisyyden, valinnan ja toiston avulla, kuten aiemmin tuli esille.

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 niin ikään 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.

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 alunperin 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.

Modulaarisuudesta saatavien etujen hyödyntäminen edellyttää tarkkaa ja huolellista tehtävänmäärittelyä jokaisen osatehtävän kohdalla.

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

Lopuksi annetaan vähän esimakua parametrisoinnista tarkastelemalla esimerkkiä 'elävästä elämästä'. Tietokoneen ohjelmoinnissa modulaarisuus ja parametrisointi ovat käytännössä aivan välttämättömiä ominaisuuksia. Näihin asioihin palataan lähemmin jaksolla Tietojenkäsittelyn perusteet I ja konkreettisesti jaksolla Algoritmien ja ohjelmoinnin peruskurssi.

**Esimerkki**. Toimenpiteen 'Pane kolme lusikallista kaakaojauhetta kuppiin' tosiasiallinen sisältö ei juurikaan 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 ’Laita *n* lusikallista *x*-jauhetta astiaan *y*'
    REPEAT *n* TIMES Laita lusikallinen *x*-jauhetta astiaan *y* ENDREPEAT

Moduuli 'Laita lusikallinen *x*-jauhetta astiaan *y*'
    Ota lusikkaan *x*-jauhetta
    Kaada lusikassa oleva jauhe astiaan *y*

Edellä määritellään kaksi moduulia. Moduuli käynnistetään (suoritetaan) kirjoittamalla moduulin otsikkorivi siten, että parametrien kohdalla on vakioarvot tai lauseke, jolla on arvo (käyttö). Tällöin kutsu saa aikaan moduulin suorituksen tietyillä parametrien arvoilla), 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', '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).*

… mutta mitä sitten seuraa?
Sen saat selville osallistumalla opintoihin!