# Sisällys

# *TAVOITTEET JA SISÄLTÖ*

**Tavoitteet:** Kurssin jälkeen opiskelija tuntee pääpiirteissään informaatioteknologian osa-alueet ja keskeisimmät käsitteet sekä tietokonelaitteiston osat ja toimintaperiaatteen. Opiskelijaa ymmärtää algoritmiseen ajattelun, ohjelmointiin liittyvät peruskäsitteet ja osaa kirjoittaa yksinkertaisia algoritmeja käyttäen pseudokieltä.

**Sisältö:** Kurssilla esitellään aluksi informaatioteknologian perusteita ja keskeistä käsitteistöä. Lisäksi tutustutaan pintapuolisesti tietokoneen rakenteeseen ja toimintaan. Pääpaino on algoritmien ja ohjelmoinnin peruskäsitteiden ja rakenteiden esittelyssä käyttäen pseudokieltä: kontrollirakenteet, modulaarisuus, moduulit, abstrakti tietotyyppi ja tietorakenteet.

**Luentopäivä**: 10.10.2015 klo 10.15-14.30.

**Tentit:** 1. tentti 29.10.2015, katso muut tenttimahdollisuudet kurssisivuilta.

# *OPPIMATERIAALIT*

Tämä jakson materiaalit ovat:

- Opiskelijan käsikirjan luku 2.1.2 (informaatioteknologian peruskäsitteet ja asiat),
- tämä opiskeluopas (ohjaa opiskelun kulkua ja sisältää paljon lisäselvennyksiä ja esimerkkejä) ja
- Johdatus tietojenkäsittelytieteeseen –opintomoniste: algoritmien perusteet (monisteen alkuosa).

## *Opintomoniste*

**Johdatus tietojenkäsittelytieteeseen**, **opintomoniste**. Testimateriaali sisältää suurin piirtein samat asiat kuin opintomonisteen alku. Tällä kurssilla TTP I annetaan vain kyseisen monisteen alku (s. 1-78), joka käsittelee algoritmeja. Monisteen loppuosa julkaistaan ja käsitellään opintojaksolla TTP II.

**Seuraavia opintomonisteen asioita ei vaadita tentissä:**

- s. 49 - 50 esimerkkiä moduulista suorakulmainenko
- s. 55 - 56 esimerkkiä (neliöjuuri)
- s. 56 - 57 esimerkit integroinnista ja seulasta
- s. 67 (sivun alussa oleva kuva pois lukien) - monisteen loppu

Osa näistä asioista kuitenkin käsitellään tässä opiskeluoppaassa, koska ne ovat oppimisen kannalta hyödyllisiä

## *Opiskelijan käsikirja, opiskeluopas, verkkomateriaali, havainnollistusohjelma ViLLE*

**Opiskelijan käsikirja** löytyy vain verkosta. Sen luku 2.1.2 linkkeineen kuuluu osittain tämän kurssin vaatimuksiin, mutta vaadittavat asiat esitetään myös jakson TTP I kotisivuilla.

**Jakson TTP I opiskeluopas** (eli tämä opus) ohjaa tämän opintojakson opiskelua. Jakson sisältö on jaettu kalenteriviikkojen mukaan viiteen osaan. *Aloita viikon asioiden lukeminen aina opiskeluoppaasta ja lue sitä sen jälkeen rinnakkain opintomonisteen kanssa.* Näin ollen luet monet (tärkeimmät) asiat kahteen kertaan, mutta samat asiat on ilmaistu niissä hiukan eri tavalla. Lisäksi opiskeluoppaassa sanotaan se mikä on erityisen tärkeätä ja annetaan lukuisia hyvin yksinkertaisia lisäesimerkkejä. Näin ollen voidaan ajatella, että opiskeluopas toimii luentojen korvaajana. Opiskeluoppaasta löydät jokaisen viikon kohdalta kyseiseen viikkoon liittyvät oppimateriaalit ja harjoitustehtävät. Jokaisen viikon kohdalla kerrotaan myös viikon tärkeimmät asiat.

**Viikon ohjelma** ei katkea aina luonnollisesti esim. luvun lopussa, vaan se saattaa katketa kesken pidempää lukua. Jos sinulla on aikaa, niin kannattaa tietenkin lukea luku loppuun saakka. Usein viikon **harjoitustehtävistä** pari ensimmäistä tehtävää käsittelee edellisellä viikolla käsiteltyjä asioita. Tällainen viikkojako on tehty ihan tarkoituksella, koska tällöin sinulla on paremmat valmiudet tehdä hiukan vaativampia tehtäviä. Opintoryhmässä voidaan poiketa esitetystä aikataulusta, jos se nähdään tarkoituksenmukaiseksi.

**Huomaa, että tietyllä viikolla käsiteltävät asiat opiskella ja harjoitustehtävät tehdä ennen viikon kokoontumista ja niiden opiskelu tulee siis aloittaa jo edellisellä viikolla tai ainakin edellisenä viikonloppuna.**

Kurssisivuilla oleva **verkkomateriaali** sisältää useita havainnollistusohjelmia, jotka selventävät (animoivat) algoritmien toimintaa. Lisäksi kurssisivuilla annetaan linkkejä hyviin **oheismateriaaleihin**.

Kurssisivuilla on linkki ohjelmaan **ViLLE**, joka havainnollistaa algoritmien toimintaa. Ohjelma myös kysyy sinulta kysymyksiä. **Ohjelman käyttö on hyvin olennaista asioiden ymmärtämisen kannalta.**

Viikoittaisissa harjoitustehtävissä käsitellään havainnollistamisohjelmaa ViLLE, jota tulee käyttää kaikkien viikon asioiden omaksumisessa ensimmäiseksi. Harjoitustehtävissä on osa **ViLLE-tehtäviä** ja näiden kanssa menetellään seuraavasti. Ne tulee tehdä samalla aikataululla kuin muutkin ko. viikon tehtävät, jos niistä haluaa bonuksia. Kullakin viikolla ViLLE-tehtävät vastaavat n. kahta tehtävää ja ne lasketaan mukaan bonuksiin kuten muutkin tehtävät. Tehtäviä voi tehdä myöhemminkin, mutta silloin niistä ei saa bonuksia. Tehtäviä ei tarvitse palauttaa mihinkään, koska näen ViLLEstä suoraan mitkä tehtävät olet tehnyt ja milloin.

Lisäksi kurssin aikana Moodlen kautta annetaan muutama ylimääräinen tehtävä, josta saa ylimääräisiä bonuksia. Nämä tehtävät koskevat algoritmien muunnosta pseudokielestä Javaan/Pythoniin. Nämä eivät kuitenkaan kuulu kurssin vaatimuksiin ja niitä ei kysytä tentissä.

# *OPISKELUSTA JA TENTISTÄ*

Toivottavasti olet jo tutustunut opiskelumuotoihin Opiskelijan käsikirjan johdolla. Erityisesti luku 1.4 on tärkeä. Huomaa, että opiskelussa on keskeisessä asemassa itsenäinen opiskelu, jota tukee opintoryhmätyöskentely, kurssisivut, sähköpostiohjaus, kurssikohtaiset moodlen keskustelualueet ja luentopäivät (tarkista kurssisivuilta luentopäivän ajankohta ja paikka). Tällä jaksolla on käytössä opiskelijan käsikirjan luvussa 1.4.3 ja kurssisivuilla esitetty **bonusjärjestelmä**.

**Sähköpostin ja Moodlen seuraaminen on erittäin tärkeää. Moodlessa annetaan usein vinkkejä tehtävien tekemiseen ja seillä voi keskustella jakson aiheista ja tehtävistä.**

Opintojakso suoritetaan hyväksytyllä tentillä ja tentti kestää 180 minuuttia. Tentissä on yksi kysymys, joka käsittelee Opiskelijan käsikirjan luvun 2.1.2 asioita, jotka luetellaan myös jakson TTP I kurssisivuilla. Loput kysymykset käsittelevät algoritmeja eli opintomonisteessa tässä oppaassa esitettyjä asioita. Tentti on hyväksytty, jos saa noin puolet tentin maksimipistemäärästä. Tentissä vaaditaan kaikki jaksolla käsitellyt asiat mukaan lukien tämän opiskeluoppaan materiaali. Tentissä ei saa olla mukana mitään opintojakson materiaaleja. Tentistä puhutaan lisää Opiskelijan käsikirjassa ja kurssisivuilla.

# *VIIKOT 39-40*

## *Sisältö:*

## *Viikko 39: Kertausta: Informaatioteknologian perusteita ja keskeistä käsitteistöä, tietokoneen rakenne ja toiminta, käyttöjärjestelmä.*

## *Viikot 39 ja 40: Algoritmien perusvaatimukset, syntaksi, semantiikka, imperatiivinen paradigma, muuttuja, peräkkäisyys, valinta, toisto.*

## *Oppimateriaali*

Jakson TTP I kotisivuilla kohdassa ”Tentistä” mainitut materiaalit 1. ja 2. (samat kuin mitä on esitetty Opiskelijan käsikirjan luvussa 2.1.2), jotka piti lukea jo aiemmin. Nämä vaaditaan jakson TTP I tentissä.

Opintomonisteen sivut 5-34 (lajitteluesimerkkiin saakka). Teksti on sama kuin testimateriaalin s. 7-21 teksti hiukan laajennettuna. Opintomonisteen CASE-lausetta (s. 28) ja siihen liittyviä esimerkkejä *ei vaadita*. Luettavaa materiaalia on paljon, mutta lähtökohtana on se, että näihin asioihin on tutustuttu jo aiemmin (testimateriaali).

Lue materiaalit alla olevien ohjeiden mukaisesti.

## *Itsenäisen työskentelyn avuksi*

Kertaa viikolla 39 yleisen Opiskelijan käsikirjan luvun 2.1.2 asiat, jos et lukenut niitä opintojen alussa. Viikolla 39 tulee *lukea tarkasti* opintomonisteen s. 26 loppuun saakka ja tämän opiskeluoppaan s. 14 saakka (Toistorakenteet) sekä tehdä viikon 39 harjoitustehtäviä. Viikolla 40 luetaan tarkasti opintomonisteen s. 27-34 ja tämän oppaan s. 14 eteenpäin.

Ennen kuin siirryt opintomonisteeseen lue alla oleva *Alustus* ja siirry sen jälkeen opintomonisteeseen lukemalla samalla alla olevat tarkennukset ja lisäesimerkit.

### *Alustus*

**Paneudu erityisesti opintomonisteen asioihin, jotka alkavat s. 21 (Imperatiivinen paradigma)**. Tavoitteena on ymmärtää algoritminen ajattelu sekä kyetä esittämään yksinkertaisia algoritmeja täsmällisellä tavalla. Tämä on tietenkin kova tavoite, johon ei varmaankaan yhden viikon opiskelulla päästä. Algoritmit ovat keskeisessä asemassa koko jakson loppuajan, joten kuva algoritmeista selvenee myöhemmin.

| |
|---|
| Tässä esitetään algoritmien ja ohjelmoinnin perusrakenteet, jotka ovat tämän ja myös kahden seuraavan kurssin 'kulmakivet'. Palaa näihin asioihin myöhemmin. |

Opettelemme algoritmien perusrakenteet käyttäen itse määriteltyä **pseudokieltä**, jonka syntaksi on hyvin yksinkertainen. Todellisissa ohjelmointikielissä syntaksi saattaa vaihdella suurestikin, mutta aluksi on tärkeätä ymmärtää millaisia *lausetyyppejä* yleensä käytetään ja mikä on *lauseiden toiminta*. Pelkästään jonkun tietyn ohjelmointikielen eri lausetyyppien tarkka käsittely ei anna oikeata yleiskuvaa yleensä ohjelmointikielistä, koska tiukka syntaksi usein hämärtää yleiskuvan muodostumista. Näin ollen on parempi opetella ohjelmointikielten perusrakenteet väljemmin määritellyllä pseudokielellä, joka kuitenkin noudattaa todellisten kielien kuten esim. Javan syntaksia. Käyttämämme syntaksi noudattaa pääpiirteissään Algol-perheen kielien (esim. Modula, Pascal) selkeää syntaksia.

Alla on useita **kommentteja opintomonisteen tekstiin**. Alla olevaa kannattaa lukea rinnan opintomonisteen kanssa, koska siinä selitetään tärkeimmät käsitteet uudestaan vähän kansantajuisemmin ja selvennetään oleellisesti opintomonisteen esimerkkejä. Opintomonisteen esimerkkejä selitetään melkoisen seikkaperäisesti, koska useille algoritmien peruskäsitteet ja lauseiden toiminta on vielä epäselvää. Jatkossa esimerkkejä ei käsitellä tällä tarkkuudella, joten huolehdi nyt, että **ymmärrät** imperatiivisen ohjelmoinnin perusasiat: muuttuja, eri lausetyypit: asetuslause, valinta ja toisto sekä lausekkeen käsitteen.

Sivuilla 20-21 esitetään kaakaonkeittoalgoritmi, jonka yksittäiset askeleet eivät ole tärkeitä, vaan tarkoituksena on vain havainnollistaa, mitä asteittain tarkentaminen tarkoittaa. Kun sen ymmärrät, voit unohtaa koko algoritmin. Huomaa, että tämä algoritmi ei kuvaa tietokoneen ohjelmointia vaikkakin siinä käytetään samanlaisia ohjausrakenteita kuin tietokoneen ohjelmoinnissa.

Luvussa 2.4 esitetään **imperatiivinen paradigma** eli imperatiivinen ohjelmointisuuntaus. Sen keskeiset käsitteet ovat: **lause**, **lauseke** ja **muuttuja**. Algoritmi koostuu *peräkkäisistä lauseista* eli käskyistä (lauseen ja lausekkeen ero selvitetään alla) niin, että algoritmissa voidaan tarvittaessa haarautua kahteen tai useampaan suuntaan tai algoritmi voi sisältää myös toistettavia osia. *Haarautuminen* saadaan aikaan valintalauseella (IF …). *Toisto* voidaan toteuttaa useammalla eri tavalla riippuen käytettävästä kielestä (opintomonisteessa: WHILE-, REPEAT-, DO…WHILE- ja FOR-lauseet). Imperatiivinen ohjelmointi perustuu muistin 'manipulointiin', jolloin *asetuslause* on keskeisessä asemassa. Asetuslauseella

voidaan asettaa uusi arvo muuttujalle (eli jollekin muistipaikalle, tietokoneen muistissa olevalle 'lokerolle'). Algoritmin tavoitteena on asettaa tavoitellut arvot (siis ne arvot, jotka on tarkoitus algoritmilla laskea) tiettyihin muistipaikkoihin käyttäen asetuslauseita, valintalauseita ja toistolauseita. Lisäksi imperatiivisessa ohjelmoinnissa on keskeisessä asemassa *lausekkeen* käsite. Lauseke esiintyy lauseen osana ja lausekkeen arvo voi olla luku (*aritmeettinen* lauseke) tai totuusarvo (*totuusarvoinen* lauseke). Jälkimmäisiä lausekkeita käytetään valintalauseessa määräämään haarautumisen suunta sekä toistolauseissa kontrolloimaan toiston päättymistä. Tarkastellaan seuraavassa lähemmin näitä käsitteitä esittäen samalla opintomonisteessa käytetty syntaksi ohjelmointikielellemme.

Algoritmin käskyt muuttavat muuttujien eli muistipaikkojen sisältöä, ja algoritmin tila määräytyy muuttujien sen hetkisten arvojen mukaan. Itse asiassa koko ohjelman voidaan ajatella olevan jono muuttujien arvoihin kohdistuvia viittauksia ja niitä hyväksi käyttäviä arvojen muutoksia. Tämän tulkinnan taustalla on imperatiiviseen paradigmaan liittyvä suoritusmalli, ns. von Neumann –arkkitehtuuri, jossa dataa siirretään paikasta toiseen tietokoneen muistissa sekä prosessorin ja muistin välillä. von Neumann –arkkitehtuurissa myös ohjelman käskyt siirtyvät datan tavoin muistissa sekä prosessorin ja muistin välillä, mikä on oleellista tietokoneen toiminnan kannalta.

Alla on tällä opintojaksolla käytetty pseudokielen lauseiden **syntaksi** eli kielioppi lyhyesti, mutta silti täsmällisemmin kuin opintomonisteessa. Javassa on jokaista alla esitettyä lausetta vastaava lause, jonka syntaksi on kuitenkin hiukan erilainen. Lisäksi Javassa on tiettyjä rajoituksia mutta myös laajennuksia kyseisten lauseiden käyttöön ja toimintaan. Alla **lausejono** tarkoittaa yhtä tai useampaa peräkkäin kirjoitettua lausetta tai voi olla myös tyhjä. Lause on taas asetuslause, valintalause ja toistolause, jotka on määritelty alla. Lisäksi lause voi olla luku- tai kirjoituslause tai moduulin kutsu (moduulit käsitellään viikolla 41). Syntaksin merkitystä ja kulmasulkeiden < > käyttöä selvitetään syntaksin määrittelyn jälkeen.

**Asetuslause**:

<muuttuja> := <lauseke>  (useimmiten aritmeettinen lauseke)

**Valintalauseet (valintarakenteet):**

- IF <totuusarvoinen lauseke> THEN <lausejono> ENDIF
- IF <totuusarvoinen lauseke> THEN <lausejono> ELSE <lausejono> ENDIF

**Toistolauseet (toistorakenteet)**:

- *Alkuehtoinen* toisto, jossa lopetusehto (=<totuusarvoinen lauseke>) tutkitaan ennen kuin silmukan runkoa aletaan suorittamaan:

```
WHILE <totuusarvoinen lauseke> DO
   <lausejono>      (* silmukan runko *)
ENDWHILE
```

  Silmukan rungon toistaminen loppuu, kun <totuusarvoinen lauseke> on epätosi.

- *Loppuehtoinen* toisto, jossa lopetusehto tutkitaan rungon lauseiden suorituksen jälkeen:

```
REPEAT
   <lausejono>   (* silmukan runko *)
UNTIL <totuusarvoinen lauseke>
```

  Silmukan rungon toistaminen loppuu, kun <totuusarvoinen lauseke> on tosi.

- *Askeltava* toisto: suoritetaan silmukan rungon lauseet yhden kerran kutakin listan alkiota kohti:

```
FOR <muuttuja> := <listan ensimmäinen arvo>,...,<listan viimeinen arvo>  DO
      <lausejono>  (* silmukan runko *)
ENDFOR
```

Loppuehtoisesta toistosta esitettä myös Java-tyyppinen lause:

```
DO
      <lausejono>   (* silmukan runko *)
WHILE <totuusarvoinen lauseke>
```

Silmukan rungon toistaminen loppuu, kun <totuusarvoinen lauseke> on epätosi.

Syntaksin määrittelyssä käytetään yllä kulmasulkeita[1] (< ja >) ilmaisemaan tiettyä asiaa, jonka mukainen kirjoitelma tulee kyseiseen kohtaa kirjoittaa. Kulmasulkeiden sisällä on siis vapaamuotoista tekstiä, joka kuvaa sitä mitä kyseiseen kohtaan tulee kirjoittaa: lausejono (yksi tai useampi lause kirjoitettuna allekkain tai erikoistapauksessa ei yhtään lausetta), muuttuja, lauseke, … Määrittelyssä olevat osat, jotka eivät ole kulmasulkeiden sisällä, kuuluvat lauseeseen sellaisenaan eli ne tulee kirjoittaa algoritmiin tarkalleen esitetyllä tavalla (esim. asetuslauseessa := ja valintalauseessa IF). Esimerkiksi yllä oleva asetuslauseen määrittely vaatii, että

[1] Tämä on yleisesti käytetty esitystapa.

merkkiparin := (joka tulee lauseeseen sellaisenaan, koska se osa ei ole kulmasulkeiden sisällä) vasemmalla puolella tulee olla muuttujan nimi ja oikealla puolella jokin lauseke; esimerkiksi

a := a+b

Valintalauseesta käytetään kahta eri versiota. Tarkastellaan ensimmäisen version syntaksia. Lause alkaa IF-sanalla, jota seuraa jokin totuusarvoinen (eli looginen) lauseke (esim. a>1) ja jonka jälkeen tulee THEN-sana, jota seuraa lausejono eli yksi tai useampi lause (tai ei yhtään lausetta, jolloin halutaan korostaa, että tässä tapauksessa ei tehdä mitään). IF…THEN-lauseen päättää ENDIF-sana. Tässä lauseessa siis määrätään tehtäväksi tietty toiminta, jos tietty ehto on tosi, mutta ei tehdä mitään, jos ehto ei ole tosi. Jos meillä on sellainen tilanne, että meidän tulee kummassakin tapauksessa (ehto on tosi/epätosi) suorittaa eri toiminta, tulee meidän käyttää IF…THEN…ELSE-lausetta. Kielen **varatut sanat** (esim. IF, WHILE … jotka kuuluvat lauseiden kirjoitusasuun) kirjoitetaan näissä materiaaleissa isoilla kirjaimilla, jotta ne erottuvat muusta tekstistä.

**Yleisenä tärkeänä huomiona syntaksin määrittelystä tulee huomata, että valintalauseiden ja toistolauseiden osina on lauseita!** Näin ollen esim. yhden IF-lauseen haaroissa voi olla toisia valintalauseita tai myös toistolauseita. Määrittelyn mukaan lauseet sisältävät siis lauseita!

Useimmat ohjelmointikielet sisältävät yllä esitetyt lausetyypit ja lisäksi lauseiden toiminta on sama kuin tässä. FOR-lauseesta on olemassa myös rajoitetumpia tai yleisimpiä muotoja. Esimerkiksi Pascalissa FOR-lauseessa läpi käytävä lista ilmaistaan aina seuraavasti:

FOR <muuttuja> := <listan ensimmäinen arvo> TO[2] <listan viimeinen arvo> DO < lausejono>

esimerkiksi lauseessa FOR i := 1 TO 10 DO .... silmukkalaskuri i saa arvoikseen arvot 1, 2, 3, ..., 10. Sen sijaan kielissä C, C++ ja Java[3] ko. lause saa hiukan mystisemmän ulkoasun:

for (i = 1 , i <= 10 , i++) < lausejono>

| Huomaa, että vain asetuslause, IF…THEN-lause ja WHILE-lause ovat välttämättömät, sillä kaikki muut lausetyypit voidaan toteuttaa näiden avulla. |
|---|

Seuraavaksi tarkastelemme lähemmin lausekkeita ja eri lausetyyppejä esimerkkien valossa.

[2] tai DOWNTO, jolloin silmukkalaskuri <muuttuja> vähenee.

[3] Javassa for-lause on erilainen ja se on hyvin lähellä while-lausetta.

**Lauseke** on mikä tahansa kirjoitelma, jolla on jokin arvo (luku, totuusarvo: tosi tai epätosi[4], jne). Esimerkkejä lausekkeista: 2 (lukuvakio), x (muuttuja), 2+x (lauseke, jossa aritmeettinen laskutoimitus +), a=b+1 (totuusarvoinen lauseke, jonka arvo on tosi, jos a=b+1 ja muulloin epätosi). Totuusarvoisia lausekkeita käytetään lähinnä IF-, toistolauseiden ehto-osassa. Totuusarvoisia lausekkeita (ja siis myös muuttujia) voidaan yhdistää käyttäen operaattoreita AND ja OR, joiden merkitys on ilmeinen: Jos kaksi lauseketta on yhdistetty AND-operaattorilla, tulee kummankin lausekkeen olla tosi, jotta koko lauseke olisi tosi. Jos taas lausekkeet on yhdistetty OR-operaattorilla, on koko lausekkeen arvo epätosi, jos kumpikin on epätosi ja aina muulloin tosi[5]. Esimerkiksi lausekkeen[6]

(x>1) AND (x<5)

arvo on tosi, jos muuttujassa x on jokin luku välillä 1...5 pois lukien arvot 1 ja 5. Lisäksi totuusarvoisen lausekkeen eteen voi laittaa NOT-operaattorin, jolloin koko lausekkeen arvo on tosi, jos NOT-operaattoria seuraavan lausekkeen arvo on epätosi. Näin ollen lausekkeen NOT (x>1) arvo on sama kuin lausekkeen x<=1 arvo. Lausekkeissa saa käyttää tavallisia matemaattisia operaattoreita: +, –, * (kertolasku), / (jakolasku) ja vertailuoperaattoreita[7] < , > , = , >= , <= , <>.

**Asetuslauseella** (joskus puhutaan myös sijoitusoperaatiosta tai sijoituslauseesta) muutetaan siis jonkun muuttujan (eli muistipaikan) arvoa eikä sitä saa sekoittaa matemaattiseen yhtälöön. Asetus suoritetaan ns. asetusoperaattorilla, joka tässä opintomonisteessa on merkkipari **:=** . Esimerkiksi lause

a := a+b

saa aikaan sen, että ensin lasketaan (aritmeettisen) lausekkeen a+b arvo käyttäen muuttujien (muistipaikkojen) a ja b arvoja, jonka jälkeen tämä arvo asetetaan muuttujan a uudeksi arvoksi. Tässä tulee sekä a:lla että b:llä olla jokin arvo (annettu asetuslauseella tai jollain muulla tavalla), sillä muutenhan lauseketta a+b ei pystytä laskemaan. Useissa kielissä oletetaan muuttujilla olevan jokin oletusarvo (esim. 0), mutta koska tämä ei ole yleinen käytäntö, niin oletamme, että *muuttujilla (muistipaikoilla) ei ole mitään alkuarvoa.*

**Lisäesimerkki** asetuslauseista. Oletetaan, että muuttujissa a, b ja c ovat seuraavat arvot a=2, b=11 ja c=5. Tarkastellaan seuraavien peräkkäin suoritettavien lauseiden vaikutusta näiden muuttujien arvoihin. Lauseet toimivat vain esimerkkinä ja eivät tee mitään järkevää.

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

Lauseen a := b jälkeen vain muuttujan a arvo on muuttunut; a:n uusi arvo on 11.

Lauseen c := a+c*b jälkeen vain c:n arvo on muuttunut; c:n uusi arvo on 11+5*11=11+55=66.

---

[4] tässä käytettyjä arvoja tosi ja epätosi vastaa todellisissa ohjelmointikielissä usein arvot true ja false.

[5] Siis koko lausekkeen arvo on tosi, jos jompikumpi tai molemmat lausekkeet ovat tosia.

[6] Sulkeet voidaan jättää pois, jos lausekkeen laskujärjestys on selvä. Todellisissa ohjelmointikielissä määrätään tarkkaan milloin sulkeita on pakko käyttää, mutta ylimääräisiä sulkeita saa olla.

[7] Javassa <> kirjoitetaan muodossa !=

Tässä a+c*b on aritmeettinen lauseke, jolla on lukuarvo. Ohjelmointikielissä kertolasku ilmaistaan merkillä *, vaikka normaalisti matematiikassa sitä ei merkitä näkyviin. Jakolasku ilmaistaan merkillä /. Ohjelmointikielissä noudatetaan koulusta tuttua lausekkeiden laskujärjestystä (eli * ja / ovat vahvempia kuin + ja – eli edellä lasketaan ensin c*b) ja lisäksi lausekkeessa voidaan käyttää sulkeita ilmaisemaan laskujärjestystä; esim. lausekkeessa c*(b+a) lasketaan ensin b+a, joka kerrotaan c:llä.

(*** lisäesimerkin loppu ***)

**Huom**. Joissakin kielissä asetusoperaattorina käytetään (ikävä kyllä) symbolia = (esim. C ja Java). Asetusoperaatio ei kuitenkaan ole symmetrinen yhtäsuuruusoperaatio, vaan ohjelman suoritukselle erittäin merkittävän sivuvaikutuksen (side effect) tuottava epäsymmetrinen toimenpide: asetusoperaation vasemmalla puolella olevaan muuttujaan tallennetaan (uusi) arvo.

**Valintarakenteella** IF…THEN…ELSE-lauseessa haaraudutaan ohjelmassa kahteen suuntaan. Opintomonisteessa oleva esitys

IF ehto THEN toiminto1 ELSE toiminto2 ENDIF

tarkoittaa siis tarkemmin seuraavaa:

IF <totuusarvoinen lauseke> THEN <lausejono> ELSE <lausejono> ENDIF

Näin ollen ehto on totuusarvoinen lauseke eli jokin kirjoitelma, josta voidaan sanoa pitääkö se paikkaansa vai ei. Esimerkiksi a>b+c ja a=b+c ovat tällaisia ja niiden totuusarvot riippuvat muuttujien a, b ja c arvoista. Toiminto1 ja toiminto2 tarkoittavat lausejonoa, joka voi koostua yhdestä tai useammasta lauseesta tai olla jopa tyhjä. Haarautuminen (siis suoritetaanko toiminto1 vai toiminto2) riippuu algoritmin nykyisestä tilasta eli algoritmin muuttujien sen hetkisistä arvoista. Algoritmin tila muuttuu yleensä jokaisen lauseen suorituksen jälkeen.

Opintomoniste s. 26 *minimin määräävä algoritmi 2).* Ensin tulee nimetä se muuttuja, johon minimin arvo talletetaan (olkoon se min). Koska ratkaisun tulee perustua kahden alkion vertailuihin, algoritmi tulee sisältämään useita valintoja. Ensin verrataan keskenään kahta muuttujaa (x ja y) ja sen jälkeen näistä pienempää verrataan kolmanteen muuttujaan (z). Algoritmin havainnollistamiseksi oletetaan, että muuttujan x arvo on 5, muuttujan y arvo on 3 ja muuttujan z arvo on 1. Nyt muuttujaan min pitäisi saada luku 1. Ensin tutkitaan onko x<y eli onko 5<3. Koska tämä ei ole totta, suoritetaan kyseisen IF-lauseen ELSE-haara eli lause

IF y < z THEN min := y ELSE min := z ENDIF

Tällöin tutkitaan onko y<z eli onko 3<1. Koska tämä ei ole totta, suoritetaan ELSE-haara eli lause min:=z. Näin ollen muuttujaan min asetetaan muuttujan z arvo eli muuttuja min saa arvon 1. Jos y ja z olisivat olleet yhtäsuuria, niin muuttuja min olisi saanut arvokseen tämän yhtäsuuren arvon (eli edellä olisi suoritettu ELSE-haaran lause).

Huomaa, että IF-lauseen THEN- ja ELSE-haarojen toiminnat voidaan aina vaihtaa keskenään, jos IF-lauseen ehto muutetaan negaatiokseen (vastakohdakseen). Edellä

oleva IF-lause on siis ekvivalentti (siis saa aikaan tarkalleen saman toiminnan) seuraavan lauseen kanssa:

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

*Opintomonisteen s. 26 viimeinen esimerkki***.** Tutkitaan miten muuttujien a ja b arvot muuttuvat suoritettaessa alla olevat neljä lausetta peräkkäin.

```
a := 12
b := 5
IF a < b THEN a := a + b ELSE b := a + b ENDIF
IF a < b THEN
        IF b > 15 THEN b := 1 ELSE b := 2 ENDIF
ELSE
        IF a > 15 THEN a := 1 ELSE a := 2 ENDIF
ENDIF
```

Numeroidaan kaikki lauseet alla olevalla tavalla, jotta niihin voidaan seuraavassa selityksessä viitata.

```
1.      a := 12
2.      b := 5
3.      IF a < b THEN
        3.1     a := a + b
        ELSE
        3.2     b := a + b
        ENDIF
4.      IF a < b THEN
        4.1     IF b > 15 THEN
                4.1.1           b := 1
                ELSE
                4.1.2           b := 2
                ENDIF
        ELSE
        4.2     IF a > 1 THEN
                4.2.1           a := 1
                ELSE
                4.2.2           a := 2
                ENDIF
        ENDIF
```

Ohjelma koostuu siis neljästä lauseesta, joiden osina on myös lauseita. Kaksi ensimmäistä ovat yksinkertaisia asetuslauseita, joissa muuttujaan a talletetaan arvo 12 ja b:hen 5. Lause 3 on IF…THEN…ELSE-lause, jossa sekä THEN- että ELSE-haara sisältävät kumpainenkin yhden asetuslauseen (lauseet 3.1 ja 3.2). Ehdon a<b toteutuvuuden mukaan haaraudutaan vain toiseen haaraan ja näin ollen suoritetaan vain toinen lauseista 3.1 ja 3.2. Koska ehto 12<5 on epätosi, suoritetaan lause 3.2 eli b saa arvokseen lausekkeen a+b arvon. Siis *b=17*. Näin lause 3 on kokonaisuudessaan suoritettu. Seuraavaksi suoritetaan lause 4, mutta tulee muistaa, että b:n arvo ei ole enää 5, vaan 17. Lause 4 koostuu IF…THEN…ELSE-lauseesta, jossa sekä THEN- että ELSE-haara sisältävät jälleen uuden IF…THEN…ELSE-lauseen. Näistä suoritetaan vain toinen eli siis lause 4.1 tai 4.2, mutta ei molempia. Lauseen 4 ehto on

muotoa a<b eli 12<17, joka on tosi eli suoritetaan lause 4.1. Tämä on puolestaan jälleen valintalause, jonka ehto on muotoa b>15 eli 17>15, joka on tosi. Näin ollen lopuksi suoritetaan lause 4.1.1, mutta ei lausetta 4.1.2. Siis *b=1*. Lauseen 4 suorituksesta aiheutui vain lauseen 4.1.1. suoritus. Näin koko algoritmi on suoritettu ja lopuksi a=12 (sama kuin alkuarvo, koska a:n arvoa ei ole muutettu algoritmin missään vaiheessa) ja b=1.

Opintomonisteen s. 27 esimerkissä esitetään myös huomattavasti monimutkaisempi algoritmi, joka määrää *kolmesta muuttujasta suurimman, keskimmäisen ja pienimmän* arvon. Esimerkki on melkoisen mutkikas, joten voit sivuuttaa sen tässä vaiheessa ja palata siihen myöhemmin. Algoritmia kannattaa havainnollistaa joillakin esimerkkitapauksilla yo. havainnollistuksen tapaan.

**Toistorakenteet**. Jos toistojen lukumäärä tiedetään etukäteen, kannattaa käyttää definiittiä toistorakennetta (FOR), koska se on helppo muodostaa. Sen sijaan jos toistojen lukumäärää ei tiedetä etukäteen, tulee käyttää indefiniittiä toistoa (alkuehtoinen WHILE ja loppuehtoinen REPEAT/DO…WHILE). Tällöin tulee tarkkaan miettiä toiston loppumista valvovan lopetusehdon muoto. Nimittäin on hyvin tavallista, että ehto laaditaan usein virheellisesti niin, että toistoja tehdään joko yksi liian vähän tai yksi liikaa. Huomaa, että FOR-lause sisältää automaattisen silmukkalaskurin muutoksen toistettavien lauseiden eli silmukan rungon suorituksen jälkeen. Esimerkiksi lauseessa (oletetaan, että muuttujassa n on positiivinen kokonaisluku)

```
FOR i := 1, 2, ..., n DO  <toistettavat lauseet> ENDFOR
```

silmukkalaskuri (muuttuja) i kasvaa yhdellä aina silmukan rungon viimeisen lauseen suorituksen jälkeen. Kun runko on jo suoritettu i:n arvolla n, silmukan suoritus loppuu. Jos n on alun perin 0, ei runkoa suoriteta kertaakaan. FOR-lause voidaan aina korvata WHILE-lauseella, mutta tällöin tulee silmukkalaskuria muuttaa asetuslauseella. Esim. edellinen FOR-lause voidaan kirjoittaa muodossa

```
i := 1
WHILE i <= n DO
     < toistettavat lauseet>
    i := i + 1
ENDWHILE
```

WHILE-lauseen toiminta on seuraava. Ensin tarkistetaan onko WHILE-sanaa seuraava ehto voimassa. Jos se on voimassa, suoritetaan kaikki lauseet, jotka ovat ennen ENDWHILE-sanaa (nämä lauseet muodostavat silmukan *rungon*). Sen jälkeen tutkitaan jälleen onko ehto voimassa. Jos ehto on voimassa, suoritetaan silmukan runko uudestaan. Jos taas ehto ei ole enää voimassa, silmukka on suoritettu, ja algoritmin suoritus jatkuu ENDWHILE-sanaa seuraavasta lauseesta. Näin ollen silmukan rungossa tulee suorittaa jotain sellaista, joka muuttaa toistoa ohjaavan ehdon totuusarvoa, sillä muutenhan syntyisi ikuinen silmukka[8] (jolloin algoritmi jää

[8] Ikuinen silmukka voidaan oikeissa ohjelmointiympäristöissä (onneksi) keskeyttää tietyillä näppäinyhdistelmillä, esim. Ctrl-C tai Break.

toistamaan ikuisesti silmukan runkoa eikä pääse silmukasta ulos). REPEAT-lauseen toiminta on vastaavanlainen, mutta tulee muistaa, että toiston loppua kontrolloiva ehto toimii WHILE-lauseeseen verrattuna päinvastoin. Valinta WHILE- ja REPEAT-lauseen välillä on usein makuasia. Kuitenkin jos pitää sallia sellainen tilanne, että toistoa ei suoriteta kertaakaan, tulee käyttää WHILE-lausetta, sillä REPEAT-lauseen runko suoritetaan aina vähintään kerran.

WHILE- ja REPEAT-lause ovat FOR-lausetta monipuolisempia, koska niissä voidaan ohjata toiston suoritusta yleisillä totuusarvoisilla lausekkeilla. Koska WHILE-lauseella voidaan toteuttaa kaikki muut toistorakenteet, niin joissakin kielissä (esim. Eiffel) on vain yksi toistorakenne: WHILE-lause. Huomaa, että toistorakenteiden syntaksi todellisissa ohjelmointikielissä poikkeaa tässä esitetystä, mutta niiden looginen toiminta on sama.

**Lisäesimerkki toistorakenteista**. Kirjoitetaan algoritmi, joka antaa ohjeet junalla matkustamiseen. Huomaa, että esimerkki ei kuvaa tietokoneen vaan ihmisen päätöksentekoa algoritmin mukaisesti.

```
kävele rautatieasemalle
IF tänään lähtee junia THEN
    WHILE juna ei ole asemalla DO odota ENDWHILE
    nouse junaan
    WHILE konduktööri ei ole saapunut DO odota ENDWHILE
    IF sinulla on lippu THEN näytä lippu ELSE osta lippu ENDIF
    WHILE juna ei ole määränpäässäsi DO odota ENDWHILE
    astu ulos junasta
ELSE palaa kotiin
ENDIF
```

IF- ja WHILE- lauseiden ehdot voidaan haluttaessa kirjoittaa myös päinvastaisesti, mutta tällöin tietenkin myös algoritmin loppuosaa tulee muuttaa. Samoin lauseita voidaan sisentää eri tavalla kuin yllä. Esitetään algoritmi uudestaan muuttamalla ehtoja ja merkitään selvyyden vuoksi yhtä lausetta yhdellä yhtenäisellä pystyviivalla.

```
kävele rautatieasemalle
IF tänään ei lähde junia THEN
    palaa kotiin
ELSE
    WHILE juna ei ole asemalla DO
        odota
    ENDWHILE
    nouse junaan
    WHILE konduktööri ei ole saapunut DO
        odota
    ENDWHILE
    IF sinulla ei ole lippua THEN
        osta lippu
    ELSE
        näytä lippu
    ENDIF
    WHILE juna ei ole määränpäässäsi DO
        odota
    ENDWHILE
    astu ulos junasta
ENDIF
```

**Huomaa, että tämän oppaan sivulla 26 on tiivistelmä, jossa esitetään imperatiivisen ohjelmoinnin peruskäsitteet.**

**Kommentteja opintomonisteen esimerkeistä s. 31-34**.

*Osoitteenhakualgoritmi*, s. 31, on esitetty melkoisen korkealla tasolla. Nimittäin meidän tulee esim. tietää, miten 'otetaan listalta seuraava henkilö', miten testataan ‘onko annettu nimi löytynyt’, ‘onko lista loppu’ ja tietenkin se, miten nimilista toteutetaan tietokoneessa. Näihin kaikkiin kysymyksiin saamme vastauksen jo tämän opintojakson aikana. WHILE-sanaa seuraavassa ehdossa tulee huomata AND-operaation käyttö. Ehtoja muodostettaessa voi käyttää myös OR-operaatiota, joka toimii intuition mukaisesti. Totuusarvoinen lauseke

ehto1 OR ehto2

on tosi silloin ja vain silloin, kun jompikumpi tai molemmat ehdoista ehto1 ja ehto2 ovat tosia. Sen sijaan totuusarvoinen lauseke

ehto1 AND ehto2

on tosi vain silloin, kun sekä ehto1 että ehto2 ovat kumpikin tosia.

Tarkastellaan seuraavaksi opintomonisteen s. 31 esimerkkiä, jossa määrätään *n-kertoma*. Algoritmi esitetään hyvin matalalla ja konkreettisella tasolla käyttäen vain asetuslausetta ja toistorakennetta. Näin ollen algoritmin esitys on täsmälleen samanlainen kuin tehtäessä se jollain todellisella ohjelmointikielellä, jossa on kyseiset toistorakenteet käytettävissä. Tässä oletetaan, että muuttujassa n on jokin ei-negatiivinen kokonaisluku, jonka kertoma lasketaan, ja saatu tulos tallennetaan muuttujaan k. Algoritmin pitää toimia oikein kaikilla n:n arvoilla.

Selvitetään ensin WHILE-lausetta käyttävän algoritmin ideaa. Koska n!=1*2*3*...*n, tulee algoritmin perustua peräkkäisiin kertolaskuihin. Olkoon k se muuttuja, johon tulos (ja myös välitulokset) talletetaan ja johon kohdistetaan peräkkäisiä kertolaskuja. Ensin asetetaan k:=1. Sen jälkeen k kerrotaan yhdellä, *saatu tulos* kerrotaan kahdella, *saatu tulos* kerrotaan kolmella, saatu tulos kerrotaan neljällä, ..., saatu tulos kerrotaan n:llä. Näin ollen toistettava lause on muotoa

k := k * i

missä i=1,2,...,n. Tässä asetuslauseessa lasketaan aina ensin oikean puolen arvo eli edellisen kierroksen tulos kerrotaan muuttujan i arvolla ja saatu tulos asetetaan muuttujan k uudeksi arvoksi. Lisäksi i:tä tulee kasvattaa yhdellä eli silmukan rungossa on siis kaksi lausetta: k:=k*i ja i:=i+1.

Tarkastellaan algoritmin toimintaa esimerkiksi kun n=4.

- ensimmäinen kierroksen jälkeen eli kun silmukan runko on suoritettu yhden kerran loppuun saakka: k=1 ja i=2.
- toisen kierroksen jälkeen: k=1*2=2 ja i=2+1=3
- kolmannen kierroksen jälkeen: k=2*3=6 ja i=3+1=4
- neljännen kierroksen jälkeen: k=6*4=24 ja i=4+1=5

Nyt ehto i<=n (i=5 ja n=4) ei ole enää tosi eli silmukan suoritus on päättynyt ja k:ssa on haluttu tulos. Huomaa, että silmukan runko suoritetaan siis viimeisen kerran kun silmukan alussa i=n eli kun i=4, joka on oikein.

Tarkkaavainen lukija varmaan huomaa, että yhdellä kertominen on turhaa Näin ollen algoritmissa olisi voitu kirjoittaa hyvin i:=2 ennen silmukkaa eli aletaan kertomaan vasta kakkosella. Lisäksi meidän tulee varmistaa, että algoritmi toimii erityistapauksissa, joka tarkoittaa tässä n:n arvoa 0. Jos n=0, niin silloin WHILE-lauseen ehto on heti epätosi, jolloin silmukan runkoa ei suoriteta kertaakaan. Tällöin k:n arvoksi jää sen alkuarvo 1, joka on aivan oikein, koska 0!=1.

FOR-lausetta käyttävä algoritmi on muuten sama kuin WHILE-lausetta käyttävä, mutta nyt meidän ei tarvitse kirjoittaa lausetta i:=i+1, koska se tapahtuu oletusarvoisesti FOR-lauseessa. Tietenkin voitaisiin kirjoittaa yhtä hyvin FOR i:=2,3,...,n, koska ykkösellä kertominen ei vaikuta tulokseen. Myös tässä tulee varmistua siitä, että algoritmi toimii kun n=0, jolloin kertoman arvo määritelmän mukaan on 1. Tällöin FOR-lausetta ei suoriteta lainkaan, koska listassa i:=1,2,...,0 ei ole mitään läpikäytävää (loppuarvo on pienempi kuin alkuarvo). Tällöin muuttujan k arvoksi jää 1, joka on aivan oikein.

Lue tarkkaan s. 32 esimerkki käsitteistä.

Opintomonisteen esimerkki (s. 32), jossa määrätään *lukujonon maksimi*, perustuu seuraavaan ideaan. Olkoon muuttuja, johon suurin arvo talletetaan, nimeltään maksimiehdokas. Kaikki luvut käydään läpi silmukassa ja jokaisen luvun kohdalla tarkistetaan onko tarkasteltava luku suurempi kuin nykyinen maksimiehdokas ja jos se on, asetetaan se uudeksi maksimiehdokkaaksi. Mutta mikä on sopiva alkuarvo maksimiehdokkaalle? Valitaan aluksi maksimiehdokas = lukujonon 1. luku ja sen jälkeen käydään loput luvut lävitse käyttäen toistorakennetta.

Tarkastellaan seuraavaksi esimerkkiä (opintomoniste s. 33), jossa tutkitaan *onko annettu luku n alkuluku*. Jos luku on suurempi kuin 2 voidaan käyttää seuraavaa

sääntöä: Luku n on alkuluku, jos jakolasku n/t ei mene tasan millään t:n arvolla t=2,3,4, ..., n−1. Tämä tutkitaan toistorakenteella. Nyt kannattaa käyttää loppuehtoista REPEAT-lausetta, koska ainakin yksi jakolasku tulee suorittaa, mutta mikä on lopetusehto? Tehtävä on selvitetty, kun jossain vaiheessa jako on mennyt tasan (jakojäännös=0, jolloin n ei ole alkuluku) tai on jo jaettu n−1:llä (silloin t=n, koska jakolaskun jälkeen t kasvaa yhdellä). Huomattakoon, että algoritmi on (opintojen tässä vaiheessa) suhteellisen hankala ja sitä voi selvittää itselleen tutkimalla algoritmin käyttäytymistä ja muuttujien arvojen kehittymistä joissakin esimerkkitapauksessa; esim. n=9 (ei ole alkuluku) n=7 (on alkuluku).

Sivun 33-34 esimerkissä etsitään luonnollisten lukujen (eli vähintään ykkösen suuruisten kokonaislukujen) x ja y *suurin yhteinen tekijä*, joka on suurin sellainen positiivinen kokonaisluku, joka on sekä x:n että y:n tekijä. Esimerkiksi syt(24,54)=6 ja syt(11,13)=1, koska 11 ja 13 ovat alkulukuja. Tentissä ei vaadita, että muistaisit miten syt määrätään ja niinpä tämä esimerkki toimii vain esimerkkinä menetelmästä, joka kirjoitetaan algoritmiksi. Algoritmi perustuu seuraavaan ideaan (jonka matemaattista oikeellisuutta ei tässä todisteta, ja jonka voit siis ottaa tosiasiana): vähennetään toistuvasti aina suuremmasta luvusta pienempi, kunnes luvut ovat yhtäsuuria. Koska luvut voivat alun perin olla yhtä suuria, tulee käyttää WHILE-lausetta, koska tulee sallia tilanne, jossa ei suoriteta toistorakenteen runkoa kertaakaan. Siis lopetusehto on muotoa x<>y. 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 runkoa (joka koostuu nyt yhdestä IF-lauseesta) ei enää toisteta. Silmukan suorituksen jälkeen muuttujassa x (ja myös muuttujassa y) on lukujen 24 ja 54 suurin yhteinen tekijä. Huomaa, että tämäkin algoritmi muuttaa muuttujien x ja y sisältöjä.

## *Viikon tärkeimmät asiat*

Imperatiivinen paradigma ja siihen liittyvät käsitteet: muuttuja, lauseet (asetuslause, valinta, toisto), lauseke, esimerkit, harjoitustehtävät.

## *Harjoitustehtävät viikoille 39 ja 40*

Jos ryhmällänne ei ole lähikokoontumista viikolla 39, niin viikon 39 tehtävien ratkaisuja ei tarvitse lähettää viikolla 39, koska viikkojen 39 ja 40 tehtävät käsitellään viikon 40 ryhmäkokoontumisessa.

**Jos et tule viikon 40 ryhmäkokoontumiseen tai jos olet verkko-opiskelija, niin sinun tulee lähettää viikkojen 39 ja 40 ratkaisut ennen kuin viikon 40 ryhmäkokoontuminen alkaa (Turun monimuotoryhmä ja verkko-opiskelijat 1.10.2015 klo 18.00 mennessä).**

**Viikolla 39:**

1. -2. Tee kurssisivuilla olevan havainnollistusohjelman ViLLE kohdan *Muuttujat, ehtolauseet ja toistorakenteet* tehtävät koskien Muuttujia ja ehtolauseita.
   Ks. ohje s. 4.

3. Kirjoita asetuslauseet, jotka vaihtavat muuttujien x ja y sisällöt keskenään käyttämällä (apu)muuttujaa apu. Esimerkiksi jos x=2 ja y=3, niin näiden lauseiden suorituksen jälkeen x=3 ja y=2.

4. Tarkastellaan kahta algoritmia, jotka kuvaavat liikennevaloissa käyttäytymistä. Tämä kuvaa reaalimaailman asioita eikä ole oikeata ohjelmointia. Tässä on tarkoituksena oppia ymmärtämään IF-lauseiden logiikkaa.

   *Algoritmi 1:*

```
IF valot toimivat THEN
   IF valo on punainen tai keltainen THEN
      pysähdy
   ELSE
      jatka matkaa
   ENDIF
ENDIF
```

   *Algoritmi 2:*

```
IF valot toimivat THEN
   IF valo on punainen tai keltainen THEN
      pysähdy
   ENDIF
ELSE
      jatka matkaa
ENDIF
```

   Missä tilanteissa algoritmit toimivat eri tavoin?
   *Ohje*: Mieti ensin, mitkä lauseet kuuluvat kuhunkin THEN- ja ELSE-haaraan.

**Viikolla 40:**

5. -6. Tee kurssisivuilla olevan havainnollistusohjelman ViLLE kohdan *Muuttujat, ehtolauseet ja toistorakenteet* tehtävät koskien toistorakenteita.
   Ks. ohje s. 4.

7. Mitkä ovat alla olevan algoritmin lauseet ja lausekkeet? Havainnollista[9] tarkasti algoritmin (joka ei tee mitään järkevää) toimintaa. Mitkä ovat muuttujien i ja k arvot lopuksi?

```
i := 2
k := 3
WHILE i * k < 100 DO
    i := i * k
    k := k + 1
ENDWHILE
```

8. Kirjoita opintomonisteen esimerkin s. 31 (luonnollisen luvun kertoman määrääminen) algoritmi uudelleen käyttäen loppuehtoista DO…WHILE-toistorakennetta. Muuttujan n arvo ei saa muuttua.
   *Ohje*: Mieti tarkkaan minkälaiset lopetusehdot kelpaavat.

9. Havainnollista tarkasti alkuluku-algoritmin (opintomoniste s. 33) toimintaa, kun a) n=7 ja b) n=9.

10. Oletetaan, että muuttujassa n on positiivinen kokonaisluku.

    Algoritmin

```
summa := 0
FOR i := 1,2,...,n DO
  summa := summa + i
ENDFOR
```

    suorituksen jälkeen muuttujassa summa on lausekkeen 1+2+...+n arvo. Jos n=1, niin summan arvo on 1. Kirjoita ratkaisu käyttäen WHILE…DO -lausetta. Muuttujan n arvoa ei saa muuttaa.
    *Ohje*: Vrt. n-kertoma.

[9] eli seuraa algoritmin kulkua kirjoittamalla näkyviin muuttujien arvot taulukkomuodossa algoritmin eri vaiheissa. Tätä kutsutaan myös ohjelman pöytätestaukseksi.

# *VIIKKO 41*

## *Sisältö: Modulaarisuus, parametrit, proseduurit, funktiot.*

## *Oppimateriaali*

Opintomonisteen sivut 34 (lajittelu) - 49 esimerkkiin (moduuli suorakulmainenko) saakka ja alla oleva. Lue lisäksi luku 2.6.5.

Harjoitustehtävissä käsitellään havainnollistamisohjelmaa ViLLE, jota kannattaa hyödyntää kun luet moduulien osuutta viikon materiaaleista. Opiskele jälleen alla olevien ohjeiden mukaisesti.

## *Itsenäisen työskentelyn avuksi*

Lue opintomonisteen teksti ja siihen liittyvät alla olevat lisäselitykset.

Tarkastellaan luvun 2.5.4 *lajittelu*esimerkkiä ja sen viimeisintä algoritmia sivulla 35. Lajittelu on erittäin keskeinen tehtävä, joten erilaisia lajittelualgoritmeja on paljon; niistä on kirjoitettu jopa kokonaisia kirjoja. Algoritmien vaatima suoritusaika (ns. aikakompleksisuus) ja tila (ns. tilakompleksisuus) ovat tärkeitä kriteerejä lajittelualgoritmia valittaessa etenkin jos lajiteltavia alkioita on paljon. Näitä asioita tarkastellaan opintojaksolla ”Tietojenkäsittelytieteen perusteet II”. Tässä esitetty algoritmi perustuu parittaisiin alkioiden vertailuihin ja alkioiden vaihtoon. Algoritmi sisältää kaksi sisäkkäistä toistorakennetta (kuten yleensä kaikki lajittelualgoritmit), joka tekee siitä vaikeasti ymmärrettävän.

Seuraavassa taulukossa havainnollistetaan algoritmin toimintaa, kun sisin REPEAT-lause suoritetaan ensimmäisen kerran kokonaisuudessaan eli suoritetaan listan ensimmäinen läpikäynti. Taulukossa ilmaistaan kursiivilla ne nimet, joiden vaihto on tarkastelun kohteena. Taulukossa oleva i:n arvo tarkoittaa i:n arvoa sisemmän silmukan alussa ennen mahdollista vaihtoa, jolloin siis tarkastellaan alkioita nimi[i] ja nimi[i+1]. Kun sisempi silmukka suoritetaan viimeisen kerran (siis vertaillaan listan kahta viimeistä alkiota), on i:n arvo 6 ja se saa arvon 7 sisemmän silmukan lopussa, jolloin silmukka on suoritettu kokonaisuudessaan. Tällöin muuttuja vaihtoja=4.

| **i=1** | **i=2** | **i=3** | **i=4** | **i=5** | **i=6** | **lopuksi** |
|---|---|---|---|---|---|---|
| *jussi* | jussi | jussi | jussi | jussi | jussi | jussi |
| *kati* | *kati* | fredi | fredi | fredi | Fredi | fredi |
| fredi | *fredi* | *kati* | bertta | bertta | bertta | bertta |
| bertta | bertta | *bertta* | *kati* | Kati | Kati | kati |
| sami | sami | sami | *sami* | *sami* | jaana | jaana |
| jaana | jaana | jaana | jaana | *jaana* | *sami* | mari |
| mari | mari | mari | mari | mari | *mari* | sami |

i:n arvoilla 1 ja 4 ei suoriteta vaihtoa.

Kun sisin REPEAT-lause suoritetaan toisen kerran eli suoritetaan listan toinen läpikäynti:

| **i=1** | **i=2** | **i=3** | **i=4** | **i=5** | **i=6** | **lopuksi** |
|---|---|---|---|---|---|---|
| *jussi* | fredi | fredi | fredi | fredi | Fredi | fredi |
| *fredi* | *jussi* | bertta | bertta | bertta | bertta | bertta |
| bertta | *bertta* | *jussi* | jussi | jussi | jussi | jussi |
| kati | kati | *kati* | *kati* | jaana | jaana | jaana |
| jaana | jaana | jaana | *jaana* | *kati* | Kati | kati |
| mari | mari | mari | mari | *mari* | *mari* | mari |
| sami | sami | sami | sami | sami | *sami* | sami |

i:n arvoilla 3, 5 ja 6 ei suoriteta vaihtoja. Tarkkaavainen lukija huomaa, että algoritmi ei ole kovinkaan tehokas. Nimittäin 1. läpikäynnin jälkeen listan viimeinen alkio on jo omalla paikallaan (sami), 2. läpikäynnin jälkeen myös toiseksi viimeinen alkio on jo omalla paikallaan (mari) jne. Näin ollen koko listaa ei tarvitsisi käydä läpi loppuun saakka joka kierroksella, koska osa nimistä on jo omalla paikallaan. Tähän, ja lajitteluun yleensä, palataan myöhemmin.

Huomaa, että lajittelualgoritmi käyttää korkeamman tason komentoa *vaihda*, joka voidaan toteuttaa kolmella asetuslauseella käyttäen apuna yhtä apumuuttujaa (tämähän oli viikon 39 tehtävä).

**Modulaarisuus ja moduulit,** opintomoniste s. 36 alkaen.

Aiemmin harjoittelimme algoritmien tekemistä. Itse tietokoneohjelmat koostuvat kuitenkin useista moduuleista[10], jotka koostuvat moduulin otsikosta ja moduulin algoritmista. **Modulaarisuus** on erittäin tärkeä asia, jonka merkityksen ymmärtäminen on ratkaisevassa asemassa algoritmisen ongelmanratkaisun kannalta. Modulaarisuuden rooli korostuu erityisesti ohjelmoinnissa sekä algoritmiteorian käsittelyssä. Modulaarisessa ajattelussa monimutkainen tehtäväkokonaisuus hallitaan jakamalla se pienempiin, helpommin ymmärrettäviin osiin, ja osien sisäisen toiminnan tarkastelu erotetaan niiden välisten riippuvuuksien tarkastelusta.

**Parametrien merkitys tulee ymmärtää**, joten panosta siihen. Parametrien tarkoituksena on tehdä tietyn tehtävän suorittavasta moduulista mahdollisimman yleiskäyttöinen. Parametrien valinta riippuu tietysti tehtävästä, mutta tyypillisimmät parametrisoinnin kohteet ovat: kohde johon toiminta kohdistuu (esim. opintomoniste s. 41-42: x-jauhe) tai tehtävän koko (esim. lusikallisten määrä; n-kertomaa laskettaessa n:n arvo). Moduulin suorittaman tehtävä tulee kuitenkin olla samantyyppinen riippumatta parametrin todellisesta arvosta (esim. astiaan pannaan jauhetta aina samalla tavalla riippumatta siitä, mitä jauhetta lusikassa on; n-kertomaa laskettaessa lasku etenee samalla tavalla riippumatta n:n arvosta).

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 parametreilla. Moduulin **kutsu** on lause (käsky) tai lauseke (kirjoitelma, jolla on arvo), joka aiheuttaa moduulin suorittamisen tietyillä parametreilla. Kutsussa moduulin parametrit saavat arvot, jolloin abstrakti, parametrisoitu moduuli muuttuu **konkreettiseksi**, jolloin se voidaan suorittaa.

Moduulin kutsu jossain toisessa moduulissa saa aikaan kyseisen moduulin suorituksen kyseisessä kohdassa. Moduuleja on kahdenlaisia:

- **proseduureja** (moduulin otsikkorivillä[11] ei ole sanaa RETURNS ja moduulin rungossa ei yleensä ole RETURN-lausetta) ja
- **funktiota** (moduulin otsikkorivillä on RETURNS-sana, jota seuraa vapaamuotoinen kuvaus siitä millaisen arvon funktio laskee ja palauttaa ja lisäksi moduulin rungossa tulee olla vähintään yksi RETURN-lause).

Proseduurin kutsu on lause eli käsky, kun taas funktion kutsun tulos on matemaattisten funktioiden tapaan lauseke. Funktion kutsulla on siis arvo, joka määräytyy funktion määrittelyn mukaisen algoritmin mukaisesti. Tällöin sanotaan, että funktio **palauttaa** (siis ei tulosta) kyseisen arvon funktion kutsukohtaan.

[10] Yleisempi termi on kuitenkin **aliohjelma** tai **metodi**, jota nimitystä käytetään esim. Javassa
[11] MODULE …

Moduulin määrittelyssä olevia parametreja kutsutaan **muodollisiksi parametreiksi** ja moduulin kutsussa olevia **todellisiksi parametreiksi**. Nimitys 'muodollinen' tulee siitä, että niillä ei ole arvoa kun moduulia määritellään. Muodolliset parametrit saavat arvoikseen todellisten parametrien arvot moduulin kutsun yhteydessä eli muodolliset parametrit kiinnitetään (saavat arvon) vasta kutsun yhteydessä.

Huom. Usein moduulin määrittelyssä olevia muodollisia parametreja sanotaan vain parametreiksi ja kutsussa olevia todellisia parametreja kutsutaan **argumenteiksi**.

Esimerkiksi opintomonisteen s. 45 kertoma-funktion muodollinen parametri on n ja kutsuissa olevia todellisia parametreja (argumentteja) ovat 0 ja 4. Kutsun jälkeen muodollinen parametri saa arvokseen kutsussa olevan todellisen parametrin arvon kuten alla esitetään.

**Opintomonisteen esimerkit s. 45-48:**

Ensiksi käsittelemme *n-kertoman* määräävää moduuli sivulla 45. Koska moduuli on puhtaasti laskennallinen ja laskee vain yhden arvon, tulee siitä tehdä funktio. Parametriksi valitaan tietenkin se suure, jonka kertoma lasketaan. Lue tarkkaan esimerkin selostus opintomonisteesta, koska siinä selostetaan parametrien välitys seikkaperäisesti.

Moduuli *kertomat* (s. 46-47) taas tulostaa kertomien arvoja; esimerkiksi kutsu kertomat(5) saa aikaan arvojen 0!, 1!, 2!, 3!, 4! ja 5! tulostuksen. Moduuli kertomat on proseduuri (ei sisällä RETURNS-sanaa otsikkorivillä eikä RETURN-käskyä moduulin rungossa), jolloin sen kutsu on lause eli käsky. Moduulin rungossa kutsutaan funktiota kertoma, jonka kutsu palauttaa kyseiseen kohtaan arvon, joka voidaan tulostaa tulosta-käskyllä. Tässä ei ole tarkemmin määritelty tulosta-käskyn toimintaa, mutta voimme esimerkiksi olettaa, että se tulostaa parametrina annetun lausekkeen arvon kuvaruudulle.

Tarkastellaan esimerkiksi käskyä tulosta(kertoma(3)). Käskyn tulkitseminen aloitetaan sisältä päin eli ensin suoritetaan kutsu kertoma(3), jolloin kontrolli siirtyy funktioon kertoma, joka suoritetaan muodollisen parametrin n arvolla 3. Laskettu arvo 6 palautuu kutsun kertoma(3) paikalle, jolloin suoritettava käsky saa muodon: tulosta(6), jolloin luku 6 tulostuu kuvaruudulle. Näin koko käsky tulosta(kertoma(3)) on suoritettu.

**Huom.** Moduulin suoritusta (kutsua) havainnollistetaan myös tämän viikon harjoitustehtävän 4 avulla.

Opintomonisteen s. 46 esimerkissä MODULE syt aiemmin esitetty (opintomoniste s. 33-34) algoritmi on muutettu funktionaaliseksi moduuliksi, joka palauttaa lasketun arvon.

Opintomonisteen s. 47 esimerkki *alkuluku.* Tarkastellaan esim. kutsua alkuluku(2). Tällöin todellinen parametri 2 kopioidaan muodollisen parametrin n arvoksi ja suoritetaan moduuli. Tällöin suoritetaan vain uloimman IF-lauseen THEN-haara eli palautetaan arvo tosi ja moduulin suoritus päättyy. Jos taas todellisena parametrina on 2:ta suurempi luku, niin siirrytään suorittamaan ELSE (* n>2 *) -haaraa.

*Huom*. Jotta moduuli alkuluku toimisi oikein, vaatii se parametrikseen luvun, joka on ykköstä suurempi. Näin ollen moduulille tulee kirjoittaa kommenttina otsikkorivin jälkeen ns. *alkuehto*[12], joka kertoo ne olosuhteet, joissa moduuli toimii oikein. Olosuhteet tarkoittavat yleensä rajoituksia parametrien arvoille. Tässä alkuehto on muotoa: n>1. Alkuehto on tarkoitettu moduulin käyttäjälle ohjeeksi kertomaan milloin moduuli toimii oikein. Systeemi ei (yleensä) tarkista alkuehdon voimassaoloa ja näin moduuli suoritetaan myös virheellisillä parametrien arvoilla, mutta tällöin algoritmi toimii väärin.

*Tehtävä*: Mitä tapahtuu, jos moduulille alkuluku annetaan syötteeksi 1, joka ei määritelmän mukaan ole alkuluku?

*Huom*. Ohjelmointikielissä on valmiina jakojäännöksen palauttava funktio tai operaattori. Esim. Javassa n % t tarkoittaa jakolaskun n/t jakojäännöstä.

Seuraavaksi **kokoamme algoritmimoduuleja koskevat yleiset ja tärkeät käsitteet:**

*Moduulin esittely* eli *ulkoinen kuvaus* (declaration) kuvaa moduulin abstraktin rajapinnan eli ulospäin näkyvät piirteet, mutta ei kerro mitään moduulin toteutuksesta. Algoritmimoduulin esittely samaistetaan usein moduulin otsikkoon, jossa kerrotaan moduulin nimi ja sen parametrit sekä niiden tyyppi sekä mahdollinen tulos. Moduuleja on kahta tyyppiä: funktioita (otsikossa RETURNS) ja proseduureja (otsikossa ei RETURNS). Lisäksi tässä kerrotaan kommenttina mitä moduuli tekee tai meidän pseudokielessä funktion yhteydessä toiminta voidaan kertoa myös otsikkorivillä RETURNS-sanan jälkeen.

*Moduulin määrittely* (definition) eli *sisäinen kuvaus* kuvaa moduulin sisäisen toteutuksen. Se koostuu moduulin rungosta, joka sisältää varsinaisen algoritmin. Moduulin sisäiset muuttujat ja niiden arvot eivät näy moduulin ulkopuolelle ja ne ovat olemassa vain moduulin suorituksen ajan.

*Moduulin kutsu* (call, invocation) on lause tai lauseke, joka aiheuttaa moduulin suorittamisen tietyillä parametreilla. Kutsussa moduulin parametrien arvot spesifioidaan, jolloin abstrakti, parametrisoitu moduuli muuttuu **konkreettiseksi** ja siitä tulee suorituskelpoinen. Funktion kutsu palauttaa arvon ja on siis lauseke. Proseduurin kutsu ei palauta arvoa, joten se on lause.

Opintomonisteen s. 48 viimeisessä esimerkissä käsitellään *kilpikonnagrafiikkaa*. Ensimmäisessä moduulissa piirretään neliö. Tietenkin se, mihin kohtaa ruutua ja mihin asentoon moduuli neliön piirtää, riippuu siitä, missä kohtaa ruutua konna alun perin on ja siitä mihin suuntaan konna osoittaa ennen moduulin kutsua. Monikulmiota piirrettäessä tulee kääntyä yhteensä 360 astetta, koska tulee päätyä lähtöpisteeseen. Neliötä piirrettäessä tulee yhdellä kertaa kääntyä 90 astetta, koska 360/4=90.

[12] Käsitteisiin alku- ja loppuehto (tai lopputila) palataan jaksolla ”Algoritmien ja ohjelmoinnin peruskurssi”.

Kolmiota piirrettäessä, tulee yhdellä kertaa kääntyä 360/3=120 astetta ja yleisesti n-kulmiota piirrettäessä tulee kääntyä 360/n astetta.

Tarkastellaan yo. käsitteitä s. 48 moduulin neliö valossa. Se on proseduuri ja sen kutsu on lause (käsky). Moduulin ulkoinen kuvaus on MODULE neliö(sivu), joka pitää varustaa vielä tekstillä siitä, mitä moduuli tekee: moduuli piirtää neliön, jonka sivun pituus annetaan parametrina. Sisäinen kuvaus koostuu moduulin käskyistä. Moduulin kutsu voisi olla esim. muotoa neliö(3), joka piirtää neliön, jonka sivun pituus on 3.

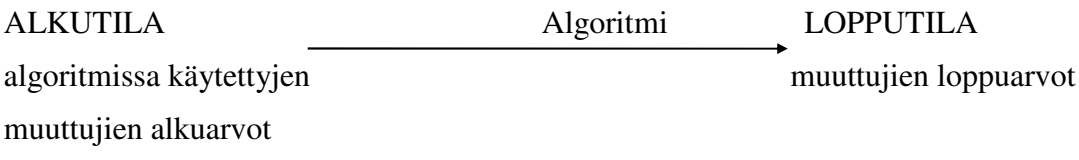
*Huom (ei vaadita).* Joillakin ohjelmointikielillä (esim. Java ja C) voidaan kirjoittaa funktionaalisia algoritmimoduuleja, joita voidaan käyttää sekä lauseen että lausekkeen tapaan[13]. Jos tällaisen funktion kutsua käytetään lauseena, niin tällöin funktion palauttamaa tulosta ei käytetä mihinkään ('menee harakoille'). Tällöin funktion kutsulla on jokin sivuvaikutus. Tällainen käyttötapa on - ainakin aloittelijalle - monimielistä ja turhaa, joten me käytämme funktion kutsua aina lausekkeena, jolloin kutsu esiintyy esim. asetuslauseen oikealla puolella tai totuusarvoisen lausekkeen osana.

## *Tiivistelmä: Imperatiivisen ohjelmoinnin peruskäsitteet ja työkalut*

- **muuttuja** (muistipaikka)
- **lauseke**
  - muuttuja (arvo)
  - aritmeettinen lauseke (lukuarvoinen)
  - totuusarvoinen lauseke (eli looginen lauseke)
  - merkkijonolauseke ("tekstiä")
  - funktionaalisen moduulin kutsu, jonka arvo on luku-, totuus- tai merkkijonoarvoinen
- **lause**
  - asetuslause
  - valintalause (IF)
  - toistolause (FOR, WHILE, REPEAT)
  - luku- / kirjoituslauseet
  - proseduraalisen moduulin kutsu
- **moduuli** eli **metodi** eli **aliohjelma**
  - erota moduulin esittely ja kutsu
  - ymmärrä moduulin parametrit ja niiden välitystapa

[13] Näitä kutsutaan lausekelauseiksi.

Algoritmi koostuu peräkkäisistä lauseista, joista keskeisessä asemassa on asetuslause. Algoritmin tila = muuttujien senhetkiset arvot. Jokaiseen lauseen suorituksen jälkeen siirrytään uuteen tilaan ja koko prosessia voidaan kuvata seuraavasti:




**Huom**. Kurssisivuilla tämä tiivistelmä on hiukan tarkemmin sisältäen myös pseudokielen syntaksin (joka on myös edellä).

## *Viikon tärkeimmät asiat*

Modulaarisuus, parametrit, funktion ja proseduurin määrittely ja kutsu, parametrien välitysmekanismi, esimerkit, harjoitustehtävät.

## *Harjoitustehtävät*

*Ohje koskien moduuleja*: Mieti ensin tuleeko moduulista tehdä proseduuri vai funktio ja mitkä ovat tarvittavat parametrit. Tarkista myös se, että moduuli tekee tarkalleen sen tehtävän, mikä tehtävän määrittelyssä on määrätty.

1. Oletetaan, että käytössä on funktio lue(), joka lukee syötejonon (syötevirran) ensimmäisen luvun ja palauttaa sen arvon (samalla kyseinen luku häviää syötejonosta). Kirjoita algoritmi, joka lukee syötejonosta sata lukua ja laskee niiden keskiarvon. Anna kolme eri ratkaisua käyttäen 1) FOR-, 2) WHILE…DO ja 3) DO…WHILE -lausetta.
   *Ohje*: Luku voidaan lukea aina samaan muuttujaan, koska lukuja ei tarvitse säilyttää. Mieti ensin minkälaisia muuttujia tarvitset laskun suorittamiseksi (imperatiivisessa ohjelmoinnissa arvot ja laskujen välitulokset tallennetaan aina muuttujiin) ja mieti tarkkaan millainen looginen lauseke tulee olla toistorakenteen jatkumista testaavassa ehdossa. Toistorakenteen sisällä (rungossa) luetaan luku muuttujaan, joka lisätään lukujen sen hetkiseen summaan. Tässä ei pyydetä tekemään moduulia, vaan pelkkä algoritmi.

2. Tarkastellaan opintomonisteen s. 35 lajitteluesimerkin viimeistä ja lopullista versiota. Voiko jomman kumman tai kummankin loppuehtoisen toiston korvata definiitillä FOR-toistolla? Jos ei voi, niin kerro miksi, ja jos voi, niin kerro miten ja pohdi myös sitä onko se tällä uudella tavalla selvempi? Kirjoita algoritmiin myös nimien vaihto käyttämällä kolmea asetuslausetta ja jotain apumuuttujaa. *Huom*. Varmistakaa, että algoritmin toiminta on ymmärretty.

3. -4. Tee kurssisivuilla olevan havainnollistusohjelman ViLLE kohdan *Moduulit* tehtävät.
   Ks. ohje s. 4.

5. Tarkastellaan alla olevaa keinotekoista (siis ei tee mitään järkevää, vaan toimii esimerkkinä) ohjelmaa, joka sisältää luvun palauttavan funktionaalisen moduulin xyz kutsun. Ohjelman suoritus alkaa main-moduulin lauseesta a := –1. Kirjoita näkyviin ohjelman suorittama tulostus ja selitä ohjelman toiminta.
   *Ohje*: ks. todellisen parametrin arvon kulkeutuminen n-kertoman yhteydessä opintomonisteen s. 46. Alla a on todellinen parametri ja n on muodollinen parametri, jolla on eri arvo moduulin eri kutsuilla xyz(a).

```
MODULE main()
    a := –1
    WHILE a<5 DO
      tulosta(a, xyz(a)) (* tulosta a:n ja lausekkeen xyz(a) arvo samalle riville *)
      a := a +1
    ENDWHILE
ENDMODULE

MODULE xyz(n) RETURNS luku
    IF n<0
    THEN RETURN 1
    ELSE
        IF (n>=0 AND n<=2)
        THEN RETURN 2
        ELSE RETURN n+1
        ENDIF
    ENDIF
ENDMODULE
```

6. Kirjoita moduuli *pot*, joka palauttaa potenssifunktion $x^n = x*x* \ldots *x$ (n kpl x:iä, * tarkoittaa kertolaskua) arvon, kun reaaliluku x ja kokonaisluku n, n≥0, annetaan moduulille parametreina. Jos n=0, niin määritellään $x^0=1$. Käytä alkuehtoista WHILE-lausetta.
   Tarkastellaan kutsua pot(3,4). Havainnollista moduulin toimintaa kirjoittamalla näkyviin muuttujien arvoja, kun algoritmia suoritetaan.
   Kirjoita lisäksi tätä moduulia käyttävä lause, joka sijoittaa muuttujaan y lausekkeen $5*3^6 + 11^8$ arvon.
   *Ohje*: Itse algoritmi on hyvin samanlainen kuin n-kertoman laskeminen.

# *LUENTOPÄIVÄ YLIOPISTOLLA 10.10.2015*

Luentopäivä pidetään klo 10:15 - n.14:30. Luentopäivän ajankohdan **päivämäärä ja kellonaika sekä paikka varmistetaan hyvissä ajoin ennen luentopäivää kurssisivuilla**.

Luentopäivänä käsiteltäviin asioihin (esim. opintomonisteen esimerkit tai harjoitustehtävät) voi jokainen opiskelija vaikuttaa esittämällä toivomuksia tuutorilleen tai suoraan etäopettajalle etukäteen tai paikan päällä. Ota mukaan kaikki opintojaksoa koskeva materiaali ja mieti kysymyksiä, joihin haluat vastauksen.

Päivän aikana käsitellään mm. seuraavat asiat perusteellisesti: työkalut algoritmien muodostamiseen (eli eri lausetyypit ja niiden toiminta), miten algoritmia aletaan rakentamaan, muutama perusalgoritmi hyvin alkeellisella tasolla, moduulit ja parametrit. Lisäksi tarkastelemme myös myöhemmin käsiteltäviä asioita: mitä abstrakti tietotyyppi tarkoittaa, listan käsite sekä vektori ja muutama siihen liittyvä algoritmi. 'Teemme yhdessä' ainakin yhden vanhan tenttitehtävän.

Kurssisivulla kerrotaan myös luentopäivän tarkempi sisältö ennen luentopäivää ja lisäksi luentopäivänä jaettu materiaali on saatavilla myös kurssisivuiltamme.

Huomaa myös että luentopäivä on luennon lisäksi myös **interaktiivinen** tapahtuma, jolloin opettaja toivoo opiskelijoiden ottavan osaa opetukseen esittämällä kysymyksiä ja vastaamalla opettajan esittämiin kysymyksiin, joiden avulla opettaja kartoittaa sitä mikä on jäänyt epäselväksi.

# *VIIKKO 42*

## *Sisältö: Iteraatio, rekursio, useita esimerkkejä*

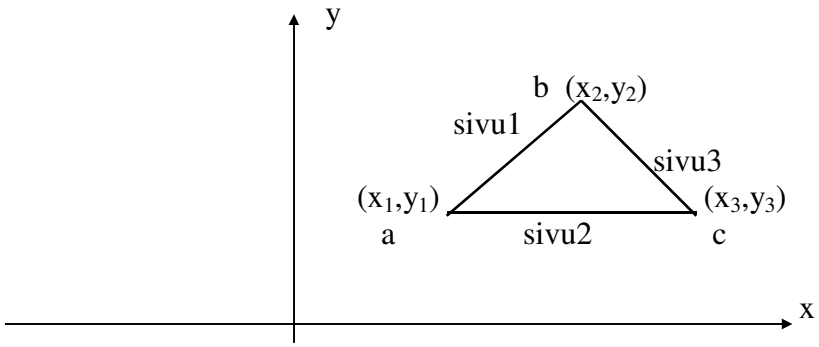
## *Oppimateriaali*

Opintomonisteen sivut 49 (alkaen moduulista *suorakulmainenko*) - lukuun 2.8 saakka.

## *Itsenäisen työskentelyn avuksi*

Opintomonisteen s. 49 on moduuli, joka *tutkii onko kolmio suorakulmainen* ja palauttaa tuloksen totuusarvona 'tosi' tai 'epätosi'. Esimerkki on hankalahko ja sitä *ei vaadita* tentissä, mutta modulaarisen suunnittelun kannalta se on opettavainen.

Tarkastellaan esimerkkiä ensin alla olevan kuvan valossa. Oletetaan, että moduulia suorakulmainenko kutsutaan parametreilla a=($x_1$,$y_1$), b=($x_2$,$y_2$) ja c=($x_3$,$y_3$), jotka ovat xy-tason pisteitä. Tällöin esim. sivu1 tarkoittaa sellaisen janan pituutta, jonka päätepisteet ovat ($x_1$,$y_1$) ja ($x_2$,$y_2$). Moduuli etäisyys laskee ja palauttaa kyseisen pituuden, jonka arvo on $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ .




*Pythagoraan lause*: Suorakulmaisessa kolmiossa kateettien (sivu1 ja sivu3) neliöiden summa on sama kuin hypotenuusan (sivu2) neliö.

Koska hypotenuusa (suoran kulman vastainen sivu) on aina pisin sivuista, ei sivuja tarvitsisi välttämättä lajitella, kuten opintomonisteen ratkaisussa on tehty, vaan riittäisi määrätä pisin sivu.

Suorakulmaisuutta testaava moduuli on rakennettu modulaarisesti: sivujen pituuden laskemiseksi ja sivujen järjestämiseen on kirjoitettu omat moduulinsa, moduulit etäisyys ja järjestä. Näistä ensimmäinen on syytä tehdä funktioksi, koska tarkoituksena on laskea kahden pisteen välinen etäisyys, joka on reaaliluku. Parametreina ovat tarkasteltavat pisteet, joten funktiolla voidaan laskea minkä tahansa

pisteparin etäisyys. Kahden pisteen välinen etäisyys voidaan laskea matematiikasta tutulla kaavalla (esitettiin yllä), joka ei ole tässä yhteydessä oleellista. Moduuli järjestä on taas syytä tehdä proseduuriksi, sillä onhan tehtävä luonteeltaan käskymuotoinen: lajittele luvut suuruusjärjestykseen.

Opintomonisteen s. 50 moduulissa *järjestä* lajitellaan 3 lukua. Lajittelualgoritmin toimintaa kannattaa tutkia soveltamalla sitä kolmeen valitsemaasi lukuun (esim. 8,6,4). Tässä käytetystä lajittelualgoritmista (josta käytetään nimitystä *vaihtolajittelu*) esitetään opintomonisteessa sivulla 61 yleisempi versio, joka järjestää vektorin (taulukon) alkiot. Se noudattaa tarkalleen moduulin *järjestä* ideaa, mutta taulukon lajittelussa tulee käyttää kahta sisäkkäistä silmukkaa, jossa käydään läpi taulukon alkiot. Se käsitellään ensi viikolla ja siihen kannattaa tutustua jo nyt.

Huomaa myös moduulin etäisyys RETURN-lauseessa oleva totuusarvoinen lauseke[14]:

```
neliö(sivu1) + neliö(sivu2) = neliö(sivu3).
```

Tässä muuttujien sivu1, sivu2 ja sivu3 arvot sijoitetaan ko. totuusarvoiseen lausekkeeseen (yhtälöön), jonka jälkeen voidaan todeta onko lauseke tosi vai epätosi. Tämä arvo (siis tosi tai epätosi) palautetaan kutsuvaan moduulin tarkasteltavan kutsun paikalle. Lauseen

```
RETURN neliö(sivu1) + neliö(sivu2) = neliö(sivu3)
```

toiminta saadaan aikaan myös seuraavalla IF-lauseella:

```
IF neliö(sivu1) + neliö(sivu2) = neliö(sivu3) THEN RETURN tosi
ELSE RETURN epätosi
ENDIF
```

**Tärkeä huom**. Opintomonisteen moduulit *järjestä* (s. 50), *vaihda* (s. 50) ja *vaihtolajittelu* (s. 61) eivät ole funktionaalisia (RETURN puuttuu, joten ne eivät palauta mitään), joten moduulin suorituksen oletetaan vaikuttavan näissä sen parametreihin, vaikka se ei käy ilmi moduulin määritelmän syntaksista. Esimerkiksi jos u=2 ja v=3, niin kutsun vaihda(u,v) suorituksen jälkeen opintomonisteessa oletetaan, että u=3 ja v=2. Näin ei kuitenkaan todellisuudessa aina ole, vaan asia riippuu käytettävän ohjelmointikielen parametrien välitystavasta. Esimerkiksi Javassa ja myös ViLLEn kielessä käytetään ns. *arvoparametrivälitystapaa*, joka tarkoittaa sitä, että todellisten parametrien arvot kopioituvat vastaavien muodollisten parametrien arvoiksi mutta ei päinvastoin eli moduulin suorituksen jälkeen parametrien arvot eivät ole muuttuneet. Tällöin aliohjelmassa tehdyt muutokset parametreihin eivät vaikuta kutsussa oleviin vastaaviin todellisiin parametreihin. Tällaisissa kielissä moduuli vaihda ei siis toimi halutulla tavalla, joten joudumme kirjoittamaan kyseiseen kohtaan kolme lausetta, jotka suorittavat vaihdon käyttäen apuna apumuuttujaa. Parametrien välitystapaa ja sen vaikutusta käsitellään tarkemmin kahdella seuraavalla jaksolla. Tällä kurssilla tulee aliohjelmasta tehdä aina kun se on mahdollista funktionaalinen (RETURN), jotta asiasta ei tulisi sekaannusta. Näin ollen esim. vaihtolajittelu kannattaa tehdä funktionaalisesti, kuten tämä oppaan s. 39 kerrotaan.

[14] Ohjelmointikielissä on valmiina tärkeimmät matemaattiset funktiot; esimerkiksi neliöön korottaminen, neliönjuuri, sini, kosini, logaritmit, luvun pyöristys jne. Sen sijaan esimerkiksi kertoman laskeminen täytyy useimmiten ohjelmoida itse. Huomaa myös, että neliö(x)=x*x.

Opintomoniste s. 52. **Rekursiota** pidetään usein vaikeasti ymmärrettävänä asiana, vaikka rekursiivisia rakenteita esiintyy ympärillämme monissa eri yhteyksissä, ei pelkästään algoritmeissa ja ohjelmissa. Toisinaan rekursio jopa esitetään ikään kuin merkillisenä temppuna, jota kannattaa käyttää vain jossakin erikoistapauksissa. Kuitenkin rekursio on modulaarisuuden mahdollistaman asteittain tarkentavan ongelmanratkaisun suora seuraus: algoritmissa käytetään hyväksi pienemmän tehtävän ratkaisevaa moduulia, joka puolestaan perustuu vielä pienempiin moduuleihin jne. Mikään ei nimittäin estä käyttämästä moduulin ratkaisemiseen moduulia itseään. On vain huolehdittava siitä, että jokainen rekursiivisella kutsulla ratkaistava alitehtävä on aidosti alkuperäistä ongelmaa yksinkertaisempi, sillä muutenhan ongelmanratkaisu ei edisty lainkaan! Tehtävän koolla tarkoitetaan käsiteltävän tiedon (tietorakenteen) kokoa. Esimerkiksi luku 4 on pienempi kuin 5 ja lista, jossa on n−1 alkiota, on n-alkioista listaa pienempi. (tietorakenteista ml. listat ja puut puhutaan myöhemmin tällä jaksolla).

Kannattaa muistaa, että modulaarisuusperiaatteen mukaisesti ongelmakokonaisuutta voidaan lähestyä analyyttisesti, osittamalla. Kutsuvan moduulin kannalta on täysin yhdentekevää, millä tavalla aliongelma ratkaistaan, kunhan tarkalleen haluttu ongelma tulee ratkaistua. Vastaavasti kutsuttavan moduulin merkitys tai toiminta ei mitenkään riipu kutsuvasta moduulista. Modulaarisen rekursiivisen algoritmin ymmärtäminen ei siis tässä mielessä mitenkään eroa modulaarisen ei-rekursiivisen algoritmin ymmärtämisestä. Erityisesti tulisi välttää rekursion mieltämistä "palaamiseksi moduulin alkuun". Rekursiota ei tulisi yrittää ymmärtää lineaarisesti, jäljittämällä koneen (iteratiivista!) toimintaa, vaan ajattelemalla ratkaisun koostuvat useista itsenäisistä osaratkaisuista. Varsin usein vaikeus ymmärtää rekursiota johtuu siitä, ettei ole täysin ymmärretty modulaarisuuden ideaa.

**Rekursiivisen algoritmin suunnittelussa on otettava huomioon kolme seikkaa.**

1. Rekursiivisten alitehtävien tulee olla ehdollisia ja jollekin (yksinkertaiselle) tehtävän tapaukselle tulee moduulissa antaa ei-rekursiivinen ratkaisu.
2. Rekursiivisten alitehtävien tulee olla aidosti alkuperäistä ongelmaa pienempiä; tarkemmin sanottuna rekursiivisen kutsun tulee lähentyä sitä yksinkertaista ongelmaa, joka moduulissa ratkeaa suoraan, ei-rekursiivisesti.
3. Rekursiivinen ratkaisu ei (yleensä) sisällä toistorakennetta vaan valintarakenteen, jossa erotetaan kohtien 1. ja 2. mukaiset tehtävät.

**Opintomonisteen esimerkit**. Rekursiota käsittelevät esimerkit s. 53-54 on kuvattu melko seikkaperäisesti.

Opintomonisteen s. 53 esimerkki: *rekursiivinen moduuli n!:n laskemiseksi*. Ensin tulee ‘keksiä’ rekursiivinen palautuskaava n!:lle eli esittää n!:lle kaava, jossa kaavan oikealla puolella esiintyy kertoma, mutta n:ää pienemmällä parametrin arvolla. Koska

$$n! = 1 * 2 * ... * (n-1) * n,$$

niin huomaamme, että n−1 ensimmäistä tulon tekijää muodostavat (n−1)!:n eli voimme kirjoittaa: n! = (n−1)! * n eli

kertoma(n) = kertoma(n−1) * n.

Kertoman määritelmän mukaan 0!=1 eli kertoma(0)=1, joten arvo 0 kelpaa triviaalitapaukseksi (ns. rekursion kanta, johon rekursio pysähtyy), joka ratkeaa suoraan. Nyt voimmekin kirjoittaa jo rekursiivisen moduulin. Sen runko koostuu vain yhdestä IF-lauseesta, jossa käsitellään erikseen triviaalitapaus (n=0, jolloin palautetaan arvo 1) ja yleinen tapaus (n>0, jolloin palautetaan lausekkeen n*kertoma(n−1) arvo). Kertoman rekursiivisen algoritmin toimintaa on havainnollistettu myös ViLLEssä, joskin siinä pitää n:n olla vähintään 1.

Opintomonisteen s. 53-54 moduulissa syt on aiemmin esitetyn iteratiivisen moduulin syt ratkaisu kirjoitettu rekursiiviseksi moduuliksi. Nyt toistorakennetta ei tarvita lainkaan, koska toiston suorittavat rekursiiviset kutsut. Moduulin ensimmäinen IF-lause sisältää rekursion kannan, jossa palautetaan arvo ja moduulin suoritus päättyy. Kaksi muuta kutsua suorittavat moduulin uudestaan, jolloin joko ensimmäinen parametri tai toinen parametri ovat pienentyneet.

Tarkastellaan opintomonisteen s. 55 esimerkkiä iteraatiosta, Esimerkkiä *ei vaadita tentissä*, mutta se on opettavainen ja kannattaa lukea. Esimerkin matemaattinen osuus voidaan ohittaa ja tutkia pelkästään algoritmia, joka laskee lukuja *rekursiivisen palautuskaavan*

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

perusteella. Tässä a on mielivaltainen positiivinen luku. Opintomonisteessa todistetaan (voit ohittaa ja katsoa pelkästään algoritmit), että laskemalla lukujonon termejä tarpeeksi pitkälle, saatu luku $x_n$ on hyvin lähellä arvoa $\sqrt{a}$. Havainnollistetaan asiaa jälleen esimerkillä. Oletetaan, että a=2, jolloin algoritmi laskee siis likiarvon päättymättömälle desimaaliluvulle $\sqrt{2} \approx 1.41421356...$ 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

$x_3=1/2(x_2+2/x_2)=1/2(3/2+4/3)=17/12\approx1.42$

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 montaakaan lukujonon termiä, kun jo saadaan hyvä likiarvo $\sqrt{2}$ :lle).

Sivulla 56 alussa esitetty moduuli käyttää lukujonoa ($x_n$), jolloin kaikki luvut ovat talletettu esimerkiksi vektoriin[15] (joita käsitellään ensi viikolla). Lukujonon kaikkia lukuja ei kuitenkaan tarvitse säilyttää, sillä uusi luku lasketaan käyttämällä hyödyksi vain edellistä lukua. Sen jälkeinen moduuli käyttääkin vain kahta muuttujaa: edellinen ja nykyinen. Uusi likiarvo (=nykyinen) lasketaan käyttäen palautuskaavaa, jossa viimeksi laskettu likiarvo (=edellinen) esiintyy palautuskaavassa oikealla puolella. Ennen silmukkaa asetetaan muuttujan nykyinen arvoksi ensimmäinen tarkasteltava likiarvo. Uutta silmukan kierrosta aloitettaessa (siis ennen uuden likiarvon laskemista) tulee asettaa edelliseksi likiarvoksi viimeksi laskettu likiarvo eli asetetaan edellinen:=nykyinen. Huomaa, että ensimmäiseksi likiarvoksi nykyinen tulee algoritmissa asettaa $x_2$ eikä $x_1$:tä. Nimittäin jos asetamme nykyinen:=1, niin REPEAT-silmukan runko suoritetaan vain kerran, koska erotus edellinen-nykyinen on negatiivinen kun ensimmäisen kerran testataan REPEAT-lauseen lopetusehtoa. Tämä johtuu siitä, että lukujono on a:n arvosta riippumatta vähenevä vasta termistä $x_2$ (eli $x_2>x_3>x_4>\ldots$) alkaen kuten edellä tapauksessa a=2 ja myös opintomonisteen yleisestä, a:sta riippumattomasta, väitteestä (2) nähdään.

Opintomonisteen s. 56-57 esimerkkejä *integroinnista* ja *seulasta alkulukujen generoimiseksi* ei vaadita tentissä ja niiden käsittely sivuutetaan.

## *Viikon tärkeimmät asiat*

Rekursio, n-kertoma rekursiivisesti, esimerkit, harjoitustehtävät.

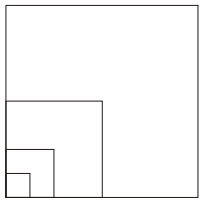
## *Harjoitustehtävät*

1. a) Tarkastellaan opintomonisteen s. 47 moduulia alkuluku. Mitä tapahtuu kutsulla alkuluku(1)?
   b) Rekursiivinen kertoman palauttava moduuli (opintomoniste s. 53) toimii vain jos n on ei-negatiivinen kokonaisluku. Mitä mahtaa tapahtua kutsulla kertoma(-1)?
   c) Tee ViLLEn kohdan *Rekursio* tehtävä.
   Ks. ohje s. 4.

2. Kirjoita moduuli Max2, jolla on kaksi lukuarvoista parametria ja joka palauttaa näistä suurimman arvon. Jos luvut ovat yhtä suuria, moduuli palauttaa ko. luvun arvon. Miten sen avulla voidaan kirjoittaa moduuli Max3, joka palauttaa kolmesta luvusta suurimman. Entä Max4?
   *Ohje*: Moduulin Max2 otsikkorivi on muotoa:
   MODULE Max2(x,y) RETURNS suurimman luvun
   Moduuli Max2 sisältää yhden IF-lauseen. Moduuleissa Max3 ja Max4 kannattaa kutsua moduulia Max2 ja niissä ei tarvita välttämättä IF-lausetta lainkaan.

[15] Huomaa siis että algoritmi, joka käyttää 'muuttujia' $x_i$ ei ole suoraan toteutuskelpoinen, mutta jos kirjoitamme $x_i$:n tilalle x[i], joka tarkoittaa vektorin x i:nnettä komponenttia, niin algoritmi on ok.

3. Tee kilpikonnagrafiikkaa käyttävä tilainvariantti algoritmi, joka piirtää alla olevan kuvion (neljä sisäkkäistä neliötä, joiden sivun pituus puolittuu) olettaen, että konna on alun perin vasemmassa alanurkassa 'nokka kohti pohjoista' ja että uloimman neliön sivun pituus on 10. Miten muutat algoritmisi moduuliksi, jolla on parametreina uloimman sivun pituus ja piirrettävien neliöiden lukumäärä. Millä sen kutsulla voidaan piirtää 12 sisäkkäistä neliötä, joista pisimmän sivun pituus on 50.
   *Ohje*: Tee silmukka, joka sisältää opintomonisteen s. 48 moduulin *neliö* kutsun.



4. Kirjoita moduulia alkuluku (opintomoniste s. 47) käyttävä moduuli, joka *palauttaa* ensimmäisen n:ää suuremman alkuluvun. Kirjoita moduuliasi käyttävä lause, joka sijoittaa muuttujaan k ensimmäisen lukua 123 suuremman alkuluvun.
   *Ohje*: Moduulilla on parametrina n. Tee toistorakenne, jossa kutsutaan moduulia alkuluku sopivilla parametreilla. Käytä WHILE-lausetta.

5. Kirjoita moduulia alkuluku käyttävä moduuli, joka *tulostaa* kaikki n:ää pienemmät alkuluvut nousevassa suuruusjärjestyksessä, kun käytössä on komento tulosta(x), joka tulostaa muuttujan x arvon. Kirjoita moduuliasi käyttävä lause, joka tulostaa kaikki lukua 100 pienemmät alkuluvut.
   *Ohje*: Moduulilla on parametrina n. Tee toistorakenne, jossa kutsutaan moduulia alkuluku sopivilla parametreilla. Käytä WHILE-lausetta.

6. Potenssifunktion $x^n$ ($n \geq 0$ ja n on kokonaisluku) arvo voidaan laskea rekursiivisesti seuraavalla tavalla:

   $x^0 = 1$
   $x^n = x * x^{n-1}$, kun $n > 0$.

   Kirjoita tähän perustuva rekursiivinen moduuli.
   *Ohje*: Tässä ei ole toistorakennetta.

# *VIIKKO 43*

## *Sisältö: Tieto- ja talletusrakenteet sekä abstrakti tietotyyppi. Talletusrakenteet tietue, taulukko ja linkitetyt rakenteet. Abstrakti tietotyyppi lista ja sen implementointi. Puu ja binääripuu.*

## *Oppimateriaali*

Opintomonisteen luvusta 2.8 alkaen s. 66 saakka se mukaan lukien ja alla oleva. Lue ensin alla oleva alustus ennen kuin siirryt opintomonisteeseen.

## *Itsenäisen työskentelyn avuksi*

### *Yleinen koko lukua 2.8 koskeva alustus*

Tähänastisissa esimerkkialgoritmeissa käsiteltävän tiedon rakenteeseen on viitattu vain epäsuorasti: on puhuttu esimerkiksi lukujonoista haluttaessa tuoda esiin käsiteltävän tiedon - toisiaan seuraavat luvut - lineaarinen rakenne. Varsin usein on kuitenkin tarpeen esittää käsiteltävän tiedon rakenne tarkasti. Tietorakenteet onkin syytä ymmärtää ohjausrakenteiden (peräkkäisyys, valinta, toisto) rinnalla ongelmanratkaisun kuvauksen komponentteina.

**Abstraktit tietotyypit** tuntuvat tässä vaiheessa varmaankin vaikeilta, joten niihin palataan vielä myöhemmin opintojen aikana; esimerkiksi opintojaksoilla ”Algoritmien ja ohjelmoinnin peruskurssi” ja erityisesti jaksolla ”Olio-ohjelmoinnin perusteet”. Tässä vaiheessa tulee erottaa tiedon **talletusrakenne** (siis se miten kyseinen rakenne talletetaan tietokoneen muistiin käyttäen ohjelmointikielten rakenteita eli tyyppejä) ja tiedon **abstrakti rakenne** (abstrakti tietotyyppi), joka määrittelee tiedon abstraktin rakenteen ja sen millaisia operaatioita kyseiseen rakenteeseen voidaan kohdistaa. Talletusrakenteet ovat siis konkreettisia välineitä, joita ohjelmointikielissä on valmiina. Näitä ovat tietue, taulukko ja linkitetyt rakenteet, jotka esitellään lyhyesti tällä kurssilla. Oliokielissä ei yleensä[16] ole sellaista tietueen käsitettä, mikä opintomonisteessa on määritelty, vaan sen tilalla käytetään *luokan* käsitettä, joka voidaan nähdä tietueen laajennuksena. Lue seuraavaksi huolellisesti luku 2.8.1 ja jatka sitten seuraavasta.

Opintomonisteessa perustietorakenteet esitellään hyvin lyhyesti. Kun loogisesti yhteenkuuluvia tietoalkioiden muodostamia kokonaisuuksia kootaan peräkkäin, saadaan lineaarinen **listarakenne** (siis ADT lista). Lineaarisuuden vaatimuksesta luopumalla, ts. sallimalla ketjujen haarautuminen, päästään hierarkkisiin

[16] Paitsi C++:ssa, jossa on 'kaikki mahdollista'. C++ ei kuitenkaan ole puhdas oliokieli, vaan sillä voi kirjoittaa myös perinteellistä C:n kaltaista ohjelmaa.

**puurakenteisiin**. Kun rakennetta edelleen yleistetään luopumalla solmujen välisten suhteiden hierarkkisuusvaatimuksesta, päädytään verkkorakenteisiin eli **graafeihin**. Puita ja graafeja ei vaadita tällä kurssilla paitsi aivan perusasiat puista. Ne ovat kuitenkin erittäin tärkeitä rakenteita tietojenkäsittelytieteissä. Näin ollen sinun kannattaa lukea läpi ainakin opintomonisteen s. 67-69. Näihin palataan jaksolla TTP II.

Jos listan koko on tiedossa jo ennen algoritmin suoritusta, kannattaa lista toteuttaa **staattisesti** varaamalla sen tarvitsema muistitila etukäteen. Tällöin käytetään vektoria[17]. Listarakenne voi olla myös aidosti **dynaaminen**, jolloin lista toteutetaan usein ns. osoittimilla, joiden avulla listan alkiot ketjutetaan toisiinsa. Dynaamisuus tarkoittaa sitä, että rakenne voi muuttua (kasvaa/lyhentyä) algoritmin suorituksen aikana. Olio-ohjelmoinnissa luokka vastaa ADT:ta ja oliokielissä on yleensä valmiina valtava määrä luokkia, joita voi käyttää suoraan miettimättä miten tieto on talletettu tietokoneen muistiin ja miten eri operaatioiden algoritmit on toteutettu. Myös dynaamista listaa varten oliokielissä on useita luokkia. Tällaisessa luokassa on valmiina esim. moduulit (metodit), joilla

- päästään listan seuraavaan alkioon
- voimme tutkia onko lista tyhjä
- voimme tehdä alkion lisäyksen ja poiston listan alkuun/loppuun/tiettyyn indeksin osoittamaan kohtaan.

Tässä on hyvä huomata, että ulospäin **tietotyypin käyttäjän tarvitsee tietää vain kyseisten operaatioiden nimet, niiden käyttötapa ja niiden toiminta** (ns. **ulkoinen kuvaus**).

Luvussa 2.8.5 käsitellään *oliokeskeisen ohjelmoinnin* eli *olio-ohjelmoinnin* peruskomponentteja, *luokkia*. Tätä osuutta opintomonisteesta ei vaadita tällä kurssilla, mutta se käsitellään kahdella seuraavalla kurssilla. Niinpä sinun kannattaa lukea jo nyt pintapuolisesti opintomonisteen kohdat 2.8.4 (Abstraktien tietotyyppien implementoinnista poislukien esimerkit) ja 2.8.5 (Oliokeskeinen ohjelmointi). Luokka on tietueen laajennus siinä mielessä, että **luokkaan kootaan loogisesti yhteenkuuluvien tietojen lisäksi myös kaikki operaatiot eli metodit (funktiot ja proseduurit), joita kyseisiin tietoalkioihin ja niiden välille voidaan kohdentaa**. Oliokeskeinen ohjelmointi on dynaamista: jokainen olio tulee aina luoda ja alustaa halutuilla tiedoilla ohjelman suorituksen aikana. Vasta sen jälkeen olioon voidaan kohdistaa luokassa määriteltyjä operaatioita. Useimmat oliokielet (kuten Java, C++ ja Eiffel) ovat imperatiivisia eli perustuvat käskypohjaiseen ohjelmointityyliin, joten 'matalan tason' ohjelmointi on samantyyppistä kuin ohjelmointi tämän kurssin pseudokielellä ja perinteellisillä imperatiivisilla ohjelmointikielillä (Pascal, Fortran, Cobol, C, Basic). Kuitenkin oliokielissä ohjelma koostuu useista luokista, jotka (yleensä) edustavat tietoabstraktioita, jotka koostuvat itse tiedoista ja metodeista, joilla näitä tietoja voidaan käsitellä. Tämä ajattelumalli on hyvin erilainen kuin perinteellisessä ohjelmoinnissa, jossa itse algoritmi ja sen askeleet ovat keskeisessä asemassa. Varmaan juuri sen vuoksi siirtyminen perinteellisestä mallista

[17] Java-ohjelmointikielessä käytetään termin vektori sijasta termiä (1-ulotteinen) taulukko, koska Javassa on luokka Vector, jolla toteutetaan dynaaminen lista.

olioparadigmaan saattaa olla yllättävän vaikeaa (ja ohjelmoija tarvitsee siis jonkin asteisen 'resetin').

Siirry nyt takaisin opintomonisteeseen ja sen lukuun 2.8.2. Harjoitustehtävissä käsitellään myös havainnollistamisohjelmaa ViLLE, jota kannattaa hyödyntää kun tarkastelet taulukoita (vektoreita).

## Opintomonisteen s. 61-62 esimerkit.

**Lisäesimerkki**. Esitetään ennen opintomonisteen esimerkkejä yksinkertainen moduuli, joka palauttaa parametrina annetun n-komponenttisen *vektorin alkioiden summan*.

```
MODULE summa(T, n) RETURNS vektorin T alkioiden summa
        s := 0
        FOR i := 1, ... , n DO
                s := s + T[i]
        ENDFOR
        RETURN s
ENDMODULE
```

Ratkaisu on erittäin tyypillinen. Summa kerätään alkio alkiolta muuttujaan s lisäämällä vanhaan summaan tarkasteltavan alkion arvo. Jotta algoritmi toimisi oikein, tulee s:n alkuarvoksi asettaa 0. Tarkastellaan esimerkkinä 3-komponenttista (siis n=3) vektoria, joka sisältää luvut 7, 5 ja 9. Tällöin T[1]=7, T[2]=5 ja T[3]=9. Aluksi algoritmissa asetetaan s=0. Kun i=1, saa s arvon 0+T[1]=7. Kun i=2, s:n vanhaan arvoon lisätään T[i]=T[2]=5 eli s saa arvon 7+5=12. Lopuksi i=3, jolloin s saa arvon 12+9=21. Huomaa tässäkin, että parametrisointi lisää moduulin yleiskäyttöisyyttä: moduulilla summa voidaan laskea minkä tahansa vektorin n:n ensimmäisen alkion summa. Esimerkiksi kutsu summa(T,2) palauttaa vektorin T kahden ensimmäisen alkion summan eli arvon 12.

*Tehtävä*: Mikähän mahtaa olla lausekkeen summa(T,1)+summa(T,2)+summa(T,3) arvo?

Taulukoiden yhteydessä on tärkeätä **erottaa** taulukon **alkio** (eli komponentti, lokeron sisältö) ja **indeksi**, jolla viitataan taulukon komponentteihin. Edellä i on kyseinen indeksi, joka siis tarkoittaa alkion järjestysnumeroa[18] (i=1,2,..) vektorissa (i:s komponentti), kun taas merkintä T[i] tarkoittaa i:nnen komponentin sisältöä.

Yleensä algoritmissa käydään läpi kaikki taulukon alkiot, jolloin läpikäymiseen käytetään usein FOR-lausetta. Tietenkin yllä oleva moduuli voidaan toteuttaa myös käyttäen WHILE-lausetta, mutta tällöin silmukkalaskuria i tulee lisätä yhdellä silmukan rungossa (FOR-lauseessahan tapahtuu automaattinen silmukkalaskurin kasvatus silmukan rungon lopussa):

[18] Joissakin kielissä (esim. C, C++ ja Java) vektorin alkiot numeroidaan (ikävä kyllä) lähtien nollasta, joka aiheuttaa helposti inhimillisen virhemahdollisuuden.

```
MODULE summa(T, n) RETURNS vektorin T alkioiden summan
        s := 0
        i := 1
        WHILE i <= n DO
                s := s + T[i]
                i := i + 1
        ENDWHILE
        RETURN s
ENDMODULE
```

(*** lisäesimerkin loppu ***)

**Huom**. Koska vektori eli 1-ulotteinen taulukko (ja myös useampiulotteiset taulukot) on staattinen rakenne, niin sen luonnin yhteydessä tulee ilmoittaa kuinka moni komponenttinen vektori luodaan, vaikka me emme pseudokielessämme sitä tee. Useissa kielissä vektorin kokoon voidaan viitata tietyllä lausekkeella: esim. Javassa voidaan kirjoittaa T.length, joka antaa vektorin T komponenttien lukumäärän (saa käyttää harjoitustehtävissä ja tentissä!). Tällöin koko vektorin alkioiden summan palauttava moduuli saa muodon:

```
MODULE summa(T) RETURNS vektorin T alkioiden summan
        s := 0
        i := 1
        WHILE i <= T.length DO
                s := s + T[i]
                i := i + 1
        ENDWHILE
        RETURN s
ENDMODULE
```

**Huom**. moduuli vaihtolajittelu (s. 61) olisi parempi kirjoittaa funktionaaliseksi eli kirjoittaa

```
MODULE vaihtolajittelu(T,n) RETURNS taulukon T lajiteltuna
```

jolloin tulee lisätä ennen ENDMODULE sanaa lause RETURN T. Nimittäin proseduraalisen moduulin vaihtolajittelu kutsu muuttaa parametriaan, vaikka tämä muutos ei ilmene moduulin määrittelystä. Tästä puhuttiin jo tämän oppaan s. 31 huomautuksessa.

Opintomonisteen s. 61 esimerkissä (ei vaadita tentissä), jossa määrätään kaupunkia a (parametri) *lähinnä oleva kaupunki*, käytetään avuksi muuttujia lähin ja lähinmatka. Niihin talletetaan tähän saakka tutkituista lyhin etäisyys ja sitä vastaavan kaupungin nimi. Aluksi tulee tietenkin asettaa muuttujaan lähinmatka jokin suuri luku, joka on varmasti suurempi kuin todellinen lyhin matka. Huomaa myös FOR-lauseen erikoinen muoto:

```
FOR j := Iisalmi,...,Varkaus DO ...
```

Läheskään kaikissa ohjelmointikielissä FOR-lauseen silmukkalaskurin (j) arvo ei voi käydä läpi tällaista arvojoukkoa eikä myöskään kelpaa taulukon indeksiksi (viittaus Etä[a,j]). Tällöin joudumme koodaamaan kaupunkien nimet esimerkiksi numeroilla

1,2,...,7, jolloin silmukkalaskuri käy läpi kokonaislukuarvoja ja samaten taulukon indeksit ovat kokonaislukuja. Esitetty ratkaisu ei ole myöskään yleinen. Nimittäin jos taulukossa oleva lyhin matka on suurempi kuin asetettu alkuarvo 100000, niin algoritmi toimii väärin. Sen vuoksi olisikin turvallisempaa asettaa alkuarvoksi, jokin taulukon todellinen luku; esim. lauseen lahinmatka := 100000 sijasta olisi parempi kirjoittaa esim. lahinmatka := Etä[Iisalmi,Kuopio].

Opintomonisteen s. 62 esimerkissä lasketaan *matriisin alkioiden summa*. Koska matriisi on 2-ulotteinen taulukko, voidaan summaus tehdä kahdella sisäkkäisellä FOR-silmukalla. Ulompi (i-silmukka) käy läpi matriisin rivit ja kutakin riviä kohti käydään ko. rivin kaikki alkiot (k-silmukka). Siis alkiot summataan muuttujaan summa seuraavassa järjestyksessä:

Taulu[1,1], Taulu[1,2], Taulu[1,3], ... , Taulu[1,m],
Taulu[2,1], Taulu[2,2], Taulu[2,3], ... , Taulu[2,m],
...
Taulu[n−1,1], Taulu[n−1,2], Taulu[n−1,3], ... , Taulu[n−1,m],
Taulu[n,1], Taulu[n,2], ... , Taulu[n,m].

Monisteessa s. 64 määritellään tarkasti abstrakti tietotyyppi lista käyttäen apuna rekursiota. Lista on yleinen rakenne, jolle voidaan määritellä eri tavalla toimivia poisto- ja lisäysoperaatioita. Esimerkiksi kun määritellään poisto-operaatio seuraavasti: listasta *poistetaan aina kauimmin listassa ollut alkio*, saadaan abstrakti tietotyyppi *jono*, jolla on paljon käytännön sovelluksia. Esimerkiksi halutessamme mallintaa kassa- tai jotain muuta jonoa, tulee määritellä jonon alkioiden tyyppi (esim. henkilötiedot) ja tarvittavat operaatiot: alkion lisäys ja poisto jonoon. Nämä yhdessä muodostavat rakenteen abstraktin mallin. Kun jonorakenne implementoidaan jollakin talletusrakenteella, tulee alkion lisäys ja poisto toteuttaa tietyllä tavalla, jotta se toimisi halutulla tavalla; esim. suoritetaan lisäys aina listan alkuun ja poisto lopusta, jolloin saadaan haluttu ominaisuus: kun jonosta poistetaan alkio, niin se on kauimmin listassa ollut alkio. *Pinolle* taas on luonteenomaista se, että *listasta poistetaan aina viimeksi lisätty alkio*. Jono- ja pinorakenteen käyttö on erittäin yleistä tietojenkäsittelyssä; esim. aritmeettinen lauseke voidaan jäsentää ja laskea sen arvo käyttäen hyväksi pinoa sekä tietyissä hakumenetelmissä (joita käytetään esim. pelien pelaamisessa) voidaan käyttää sekä pino- että jonorakennetta hyödyksi. Näihin asioihin palataan opintojaksoilla ”Olio-ohjelmoinnin perusteet” ja ”Tietojenkäsittelytieteen perusteet II”, jossa hyödynnetään näitä rakenteita mm. kun haetaan reittiä ulos labyrintista!

Opintomonisteen s. 65 esimerkkiä ei vaadita. Se on kuitenkin opettavainen mutta vaativa. Oliokielissä (esim. Javassa) on valmisluokkia, jotka implementoivat suoraan dynaamisen listan. Tällöin luokka sisältää moduuleja, joiden avulla voidaan suorittaa mm. alkion lisäys ja poisto listan mistä kohdasta tahansa, jolloin listan pituus kasvaa tai vähenee vastaavasti. Nämä moduulit ovat siis valmiiksi ohjelmoitu luokkakirjastoissa ja niitä voi siis käyttää suoraan ohjelmoimatta niitä itse! Tässä kuitenkin näytetään miten esim. alkion lisäys listaan voitaisiin tehdä, jos talletusrakenteena käytetään staattista taulukkoa.

## *Viikon tärkeimmät asiat*

Abstraktin tietotyypin ja talletusrakenteen käsitteet, tietue, taulukko, lista ja sen määritelmä, pino ja jono, esimerkit, harjoitustehtävät.

## *Harjoitustehtävät*

1. -2. Tee kurssisivuilla olevan havainnollistusohjelman ViLLE kohdan *Taulukot* harjoitustehtävät.
   ViLLEssä asetetaan taulukkoon alkioita Javamaisesti ja indeksointi alkaa nollasta: esim. jos asetetaan v:={4,8}, niin v[0]=4, v[1]=8 ja v.length=2.
   Ks. ohje tämän oppaan s. 4.

3. Mikä on vektorin v sisältö seuraavien lauseiden suorituksen jälkeen?

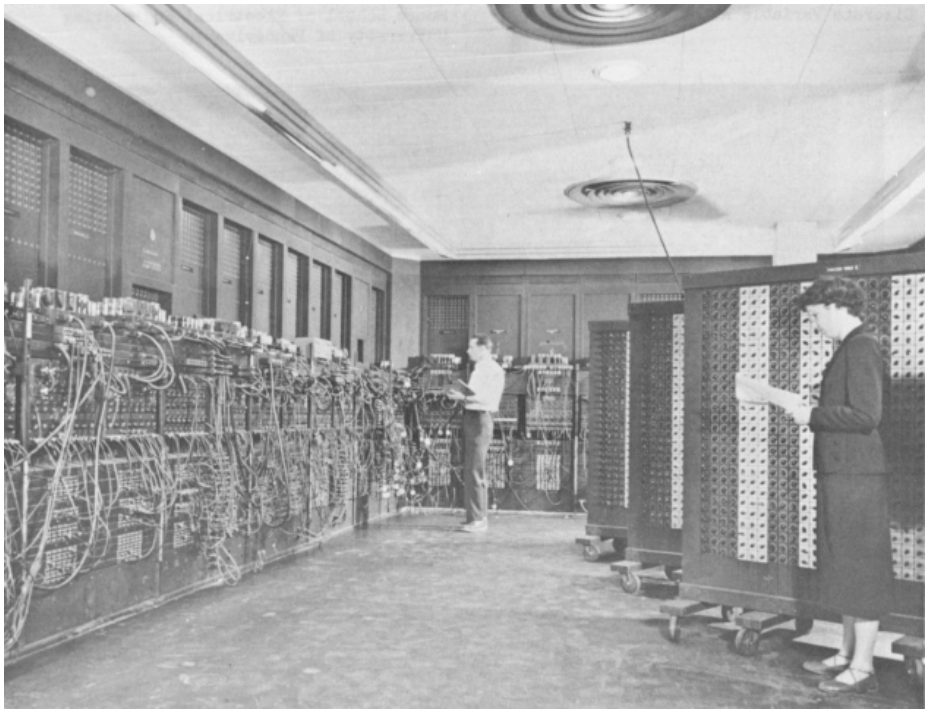
   ```
   v[1] := 2
   FOR i := 2,...,6   DO
       v[ i ] := v[ i − 1] + i
   ENDFOR
   ```

   *Ohje*: Piirrä lokerikko ja kirjoita arvot lokerikon sisälle.

4. Kirjoita moduuli, joka palauttaa n-komponenttisen vektorin pienimmän alkion arvon (pienimpiä voi olla useita, mutta sillä ei ole merkitystä). Toistorakenteena tulee käyttää WHILE-lausetta. Kirjoita myös main-metodi, jossa määritellään kaksi vektoria v1={11,5,44,2,77} ja v2={2,99,-1}, haetaan niiden pienimmät alkiot käyttäen metodiasi ja tulostetaan kyseiset arvot käyttäen lausetta tulosta(x), joka tulostaa lausekkeen x arvon.

5. Havainnollista opintomonisteen s. 61 moduulin *vaihtolajittelu* toimintaa lukulistalla 66, 55, 44, 33, jolloin n=4 ja T[1]=66, T[2]=55, T[3]=44, T[4]=33. Mieti myös miten tämä eroaa opintomonisteen s. 34 viimeisen lajittelumenetelmän ideasta.

6. Oletetaan, että lukuja sisältävä vakiomittainen lista toteutetaan vektorina. Kirjoita moduuli, jolla on parametrina vektori v ja joka palauttaa vektorin, jossa jokainen v:n negatiivinen luku on muutettu nollaksi. Esimerkiksi: jos vektorissa v on luvut 2,-33,-11,33, niin moduuli palauttaa vektorin 2,0,0,33. Toistorakenteena tulee käyttää WHILE-lausetta.
   *Ohje*: Tee toistorakenne, jossa tutkitaan jokainen komponentti ja tehdään v:hen muutos tarvittaessa, joka lopuksi palautetaan.



Voi niitä aikoja: Eniac http://fi.wikipedia.org/wiki/ENIAC