# FORTRAN ohjelmoinnin alkeisopas

**L. Lantto**

Teoreettisen fysiikan laitos
Oulun yliopisto
Linnanmaa
90570 Oulu

OPETUSMONISTE 1988

## Tiivistelmä

Tämän alkeisoppaan tarkoituksena on esitellä oleelliset kohdat fortran ohjelmointikielestä puuttumatta liikaa yksityiskohtiin, muodollisiin sääntöihin tai poikkeuksiin. Opas ei ole täydellinen eikä edes kovin tarkka. Siihen on kuitenkin koottu ne asiat fortranista, jotka ovat tärkeitä teoreettisen fysiikan laitoksen kurssin **ATK IV Numeerinen ohjelmointi** kannalta. Huomattava osa ko. kurssista käytetään numeeristen menetelmien esittelemiseen ja niihin liittyvään ohjelmointiharjoitteluun, joten siinä ei ole mahdollisuutta mutta ei myöskään tarvetta kovin yksityiskohtaiseen kielen esittelyyn.

Paras tapa oppia kieli on käyttää sitä, siksi kehoitammekin sinua mahdollisimman pian itse kirjoittamaan ja suorittamaan omia ohjelmia, aluksi suoraan matkimalla ja muuttamalla annettuja esimerkkiohjelmia. Oletamme, että sinulla on aikaisemman perusteella jonkinlainen tietämys ohjelmointiin liittyvista peruskäsitteistä. Emme aio selittää mitään tietokoneista, käyttöjärjestelmistä tai niiden tiedostorakenteista. Keskitymme ohjelmien tekemisen kannalta kielen oleellisiin osiin, joita tässä ovat muuttujat, aritmeettiset lausekkeet, kontrollirakenteet ja aliohjelmat. Opittuasi fortran ohjelmoinnin alkeet voit tarkentaa detaljitietoja käsikirjoista. Käsikirjat eivät kuitenkaan opeta hyviä ohjelmointitapoja. Paremminkin ne vain luettelevat kaikki käytettävissä olevat piirteet, joista jotkut ovat turhia, toiset hyödyttömiä ja eräät jopa haitallisia. Tämä johtuu siitä, että kun uusia piirteitä on sisällytetty kieleen, niin vanhoja tarpeettomiksi käyneitä ei ole samanaikaisesti poistettu. Poistaminen olisikin hankalaa, sillä vanhat ohjelmat lakkaisivat toimimasta. Hyvien ohjelmien kirjoittamiseksi ei kuitenkaan tarvitse opetella kielen huonoja ilmaisuja. Yritämme siis korostaa hyviä tapoja. Se tarkoittaa selkeitä, helposti luettavia ohjelmia, ei mitään knoppologiaa, eikä harvinaisia (vähän käytettyjä) 'makupaloja'. Itse ohjelmointikieli ei ole pääasia, tärkeää on, että kielen avulla voimme opetella numeeriseen laskentaan liittyviä ohjelmointitaitoja, joihin tärkeänä osana kuuluu joukko perusalgoritmeja. Samalla opimme yleisemminkin kehittelemään ongelmien ratkaisemista algoritmisen ajattelun avulla.

# Sisältö

# 1 ENSIMMÄINEN OHJELMA

Ohjelmoinnin oppimiseksi on kirjoitettava ohjelmia ja suoritettava niitä. Ensimmäinen ohjelma voi olla sama kaikilla kielillä: Laadi ohjelma, joka kirjoittaa päätteelle sanat

```
Alku aina hankalaa ...
```

Pienenkin fortran-ohjelman tekemiseksi sinun on osattava seuraavat asiat:

- ohjelman kirjoittaminen tiedostoon (editing)
- ohjelman kääntäminen (compiling)
- ohjelman suoritus eli ajo.

Nämä toimenpiteet riippuvat käyttöjärjestelmästä, ts. ensiksi on tutustuttava käytössä olevaan käyttöjärjestelmään ainakin sen verran, että opitaan muodostamaan tiedostoja teksti-editoria käyttäen. Tässä oppaassa emme käsittele noita asioita, jos tuonnempana on tarvetta viitata käyttöjärjestelmään, niin oletamme sen olevan UNIX. Fortran-kielellä em. ohjelma voidaan kirjoittaa esimerkiksi

```
program junk
print*,'Alku aina hankalaa ...'
end
```

Tässä oppaassa kirjoitamme ohjelmat `pienillä kirjaimilla`, koska niitä on mielestämme miellyttävämpi lukea. Fortran-77 standardi käyttää isoja kirjaimia, mutta useimmat kääntäjät hyväksyvät myös pienet kirjaimet sekä muuttujien nimissä että ohjelmakäskyissä. Isot ja pienet ovat siis samoja (toisin kuin esim. C-kielessä). Ohjelmarivit eivät tarvitse rivinumeroita, mutta rivin teksti ei voi alkaa rivin alusta sarakkeista 1-6. Sarakkeet 1-6 on varattu muuhun tarkoitukseen.

Ensimmäinen ohjelmarivi `program junk` ei tee mitään, se ilmoittaa vain, että kyseessä on pääohjelma ja sen nimi on `junk`. Toisella rivillä oleva print-lause toimii kuten Basicin print-käsky, se tulostaa heittomerkkien välissä olevan tekstin sellaisenaan päätteelle. Kolmannen rivin end ilmoittaa kääntäjälle (**compiler**) ohjelman loppukohdan. Suorita tämä ohjelma omassa ohjelmointiympäristössäsi, se vaatii normaalit toimenpiteet: editoi, käännä, suorita, ja varmista, että ohjelma toimii oikein.

# 2 MUUTTUJANIMET JA -TYYPIT

Ensimmäisessä ohjelmassa ei esiinny yhtään muuttujaa, ainoastaan yksi merkkijonovakio. Toinen ohjelma ratkaisee toisen asteen yhtälön

$$ax^2 + bx + c = 0$$

juuret tuttua kaavaa

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

käyttäen. Oletamme, etta vakiokertoimet $a, b$ ja $c$ ovat reaalisia ja niiden arvot luetaan päätteeltä. Tässä ohjelma:

```
program roots
real a, b, c
real x1, x2
print*,'Anna kertoimet a b ja c'
read*, a, b, c
x1 = (-b + sqrt(b*b-4*a*c))/2*a
x2 = (-b - sqrt(b**2-4*a*c))/(2*a)
print*, 'Juuret ovat:', x1, x2
end
```

Tässä ohjelmassa on muuttujia 5 kpl: `a, b, c, x1, x2`. Fortran-kielen muuttujanimet ovat 1-6 merkkiä pitkiä ja alkavat kirjaimella. Ohjelman toinen ja kolmas rivi ovat ns. määrittelyrivejä, joissa luetellaan ohjelmassa esiintyvien muuttujien nimet ja niiden tyypit. Tavallisia muuttujatyyppejä ovat `real, integer, character ja complex`. Integer- tyyppisen muuttujan arvot voivat olla vain kokonaislukuja ja real- muuttujat ovat ns. liukuvan pilkun desimaalilukuja. Fortranisssa ei ole pakko määritellä kaikkia muuttujia ennen niiden käyttöönottoa. Jos muuttujanimi alkaa kirjaimilla `i, j, k, l, m tai n` niin tyyppi on automaattisesti integer ja kirjaimilla `a-h ja o-z` alkavat muuttujat ovat real-tyyppisiä. Tässä oppaassa emme kuitenkaan käytä tällaista implisiittistä tyyppimäärittelyä vaan määrittelemme eksplisiittisesti kaikki variaabelit ohjelman alussa. Samaa käytäntöä suosittelemme lämpimästi sinullekin.

Ohjelmassa esiintyvä `read*` -lause lukee päätteeltä muuttujille arvot. Sijoituslauseissa muuttujien x1 ja x2 arvoiksi sijoitetaan oikealla puolella olevien lausekkeiden arvot, ja lopuksi ohjelma tulostaa x1:n ja x2:n arvot päät-

teelle varustettuna tekstillä `Juuret ovat:` . Sijoituslauseissa esiintyy muuttujien lisäksi funktio `sqrt(...)` , joka on fortraniin kuuluva ns. sisäinen funktio. Se laskee argumenttinaan olevan lausekkeen neliöjuuren. Koska fortran on tieteelliseen laskentaan kehitetty kieli se sisältää runsaasti funktioita `sin(x), cos(x), tan(x), exp(x), log(x),` jne.... Yhtäsuuruusmerkki = on siis fortranin sijoitusoperattori (kuten basicinkin), ja yhteen-, vähennys-, kerto- ja jakolaskuoperaattorit ovat tavanomaiset, potenssiin korotus ilmaistaan `**` -llä.

Ohjelma roots näyttää hyvältä ohjelmalta. Se on helposti luettavissa ja sen syntaksi on oikea, ts. kääntäjä ei ilmoita siitä mitään virheilmoituksia. Siinä on kuitenkin yksi paha vika: se ei toimi oikein. x1:n lausekkeessa jakolasku ei toimi halutulla tavalla, suluissa oleva lauseke tulisi jakaa 2a:lla, mutta nyt se tulee jaettua 2:lla ja kerrottua a:lla. Virheen voi korjata käyttämällä sulkuja kuten x2:n lausekkeessa. Muista olla tarkkana tällaisissa tilanteissa. Toinen asia on, että x1:n ja x2:n lausekkeissa esiintyvät vakiot 4 ja 2 on kirjoitettu kokonaislukutyyppisiksi. Ne on kuitenkin tarkoitettu real-tyyppisiksi, siksi olisi hyvä kirjoittaa ne muodossa 4.0 tai 4., jossa pelkkä desimaalipiste ilmaisee, että kyseessä on real-tyyppi.

Kokonaislukujen jakolaskusta on tiedettävä, että kahden integer-tyyppisen luvun osamäärä on myös kokonaisluku, ts. $8/3 = 2$, $5/7 = 0$, ja $-3/2 = -1$ jne. Real-ja integer-tyyppisten muuttujien kerto- ja jakolaskun tulos on automaattisesti real-tyyppinen, esim. `2*a` ja `a*5` ovat real-tyyppisiä jos `a` on sitä tyyppiä. Tyyppikonversioita varten on olemassa tarkat säännöt, jos olet epävarma tarkista asia käsikirjasta.

Numeeristen muuttujien lisäksi tarvitsemme joskus merkkijonomuuttujia. Niiden määrittelyn yhteydessä ilmoitetaan merkkijonon maksimipituus, esimerkiksi

```
character*80 text
character*10 word
...
text = 'Alku aina hankalaa ...'
word = text(6:9)
...
```

Tässä `text` on merkkimuuttuja, johon sopii korkeintaan 80 merkkiä. Alunperin merkkijono pitää sisällään tyhjiä (blankoja). Sijoitus- tai syöttölauseissa merkkijonoa täytetään vasemmalta alkaen. Osamerkkijonoon viitataan su-

luissa kaksoispisteellä erotetuilla indekseillä, `text(6:9)` tarkoittaa merkkejä `6-9`, joista yo. esimerkissä muodostuu sana `aina`.

Edellä mainittu korjaus roots-ohjelmaan ei tee siitä vieläkään hyvää ohjelmaa. Se voi toimia oikein joillakin kertoimien $a, b$ ja $c$ arvoilla, mutta jos $b^2 - 4ac$ tulee negatiiviseksi, niin ohjelman suoritus pysähtyy virheeseen, koska `sqrt`-funktio ei osaa laskea neliöjuurta negatiivisesta luvusta. Ohjelmoijan on siis osattava varautua myös tämän tyyppisiin tilanteisiin. Ohjelma ei myöskään toimi jos $a = 0$, sillä nollalla jakaminen ei ole sallittua. Vaikeammin havaittava ongelma tulee eteen, jos $a$ on paljon paljon pienempi kuin $b$ ja $c$, sillä ohjelma kyllä toimii, jos em. viat on korjattu, mutta tuloksen numeerinen tarkkuus voi olla kovin huono, eikä ohjelma anna siitä mitään varoitusta.

# 3 KONTROLLIRAKENTEET

Kirjoittakaammme ohjelmasta roots parempi versio. Vakiokertoimien $a, b$ ja $c$ arvoista riippuen yhtälön juuret voivat olla joko reaaliset tai kompleksiset. Tämä on otettava huomioon ohjelmaa kirjoitettaessa, ts. ohjelmaan on rakennettava vaihtoehtoisesti suoritettavia haaroja. Haarautumiskriteerinä on lausekkeen $b^2 - 4ac$ arvo: jos se on negatiivinen niin suoritetaan eri ohjelmalohko kuin tapauksissa, joissa se on positiivinen. Ennen sitä on vielä varmistettava, että kerroin $a$ ei ole nolla, koska se esiintyy jakajana myöhemmissä lausekkeissa. Tässä yksi versio ohjelmasta.

```
C
C Ohjelma laskee toisen asteen yhtälön juuret
C
      program roots2
      real a, b, c, dis
      real x1, x2, z
C
      print*,'Anna kertoimet a b ja c'
      read*, a, b, c
C
      if(a .eq. 0.) then
       x1 = -c/b
       print*,'Yhtälö on lineaarinen, vain yksi juuri: ', x1
```

```
       stop
      endif
C
      dis = b*b - 4.*a*c
      if(dis .lt. 0.) then
       print*,'Juuret ovat kompleksiset'
       x1 = -b/(2.*a)
       z = sqrt(-dis)/(2.*a)
       write(6,'(1x,f7.2,a,f7.2,a)')x1,' + ', z, 'i'
       write(6,'(1x,f7.2,a,f7.2,a)')x1,' - ', z, 'i'
      else
       print*,'Juuret ovat reaaliset'
       dis = sqrt(dis)
       x1 = (-b+dis)/(2.*a)
       x2 = (-b-dis)/(2.*a)
       write(6, '(1x,a,f7.2)')'x1 = ', x1
       write(6, '(1x,a,f7.2)')'x2 = ', x2
      endif
      end
```

Esimerkkiohjelma `roots2` sisältää ehdollisia haarautumisia. Kertoimet $a, b$ ja $c$ luetaan päätteeltä ja sitten tarkastetaan onko $a = 0$. Jos on, tulostetaan yksi juuri ja lopetetaan ohjelman suoritus. Muutoin lasketaan lauseke $b^2 - 4ac$ ja tutkitaan onko se negatiivinen: jos on niin juuret ovat kompleksiset, jos ei niin juuret ovat reaaliset.

## 3.1 Ehdollinen haarautuminen

Ehdollinen haarautuminen toteutetaan siis *if-then-else* konstruktiolla, joka on muotoa

```
if(ehto) then
 true-lohko
else
 false-lohko
endif
```

Tässä `ehto` on jokin looginen lauseke ja `true-lohko` tarkoittaa ohjelmakäskyjä, jotka suoritetaan jos `ehto` on tosi, muutoin suoritetaan `false-lohko`.

Huomaa, että sana `endif` on välttämätön, se ilmoittaa kääntäjälle `if-then-else` rakenteen loppukohdan. Else-sana ja siihen liittyvä `false-lohko` voidaan myös jättää kokonaan pois, ellei niitä tarvita. Yleisemmin tämä rakenne voi esiintyä muodossa

```
if(ehto_1) then
 lohko_1
else if(ehto_2) then
 lohko_2
...
else if(ehto_n) then
 lohko_n
else
 lohko_default
endif
```

Luonnollisesti kukin ohjelmalohko voi sisältää uusia kontrollirakenteita, if-then-else-lauseita, silmukoita, aliohjelmakutsuja, jne. Sisäkkäisten if-then-else lauseiden tapauksessa on tärkeää, että kutakin alkavaa if-then-else konstruktiota vastaa oikeassa paikassa oleva endif-sana.

## 3.2 Loogiset lausekkeet

If-lauseissa esiintyvät ehdot ovat *loogisia lausekkeita*. Niiden muodostamiseksi tarvitsemme loogisia operattoreita, joita ovat:

| | |
|---|---|
| `.eq.` | yhtä suuri kuin |
| `.ne.` | eri suuri kuin |
| `.gt.` | suurempi kuin |
| `.lt.` | pienempi kuin |
| `.ge.` | suurempi tai yhtä suuri kuin |
| `.le.` | pienempi tai yhtä suuri kuin |

| | |
|---|---|
| `.and.` | ja |
| `.or.` | tai |
| `.not.` | negaatio |

Näiden avulla voidaan tarvittavat ehdot ilmaista. Fortranissa on myös mahdollista käyttää loogisia muuttujia, ts. muuttujia, joiden arvoksi voi sijoittaa loogisen lausekkeen arvon. Arvo voi olla joko tosi (`.true.`) tai epätosi (`.false.`) eli valhe. Esimerkiksi

```
      real x, y, z
      logical t
      ...
      t = x .lt. y .and. x+y+z .gt. 2.0
      if(t) then
        lohko
      endif
```

Useimmiten loogiset lausekkeet voi kuitenkin kirjoittaa suoraan if-lauseisiin. On syytä huomata, että loogisen lausekkeen arvo ei ole numeerinen niin kuin se on esim. basic- ja C-kielissä. Loogisissa lausekkeissa voi esiintyä aritmeettisiä lausekkeita, laskutoimitukset suoritetaan ensin ja sitten vasta vertailut. Jos olet epävarma suoritusjärjestyksestä voit käyttää sulkuja. Loogisen lausekkeen ilmaisu ei aina noudata tarkoin luonnollisen kielen ilmaisua. Esim. ehto *jos x tai y on suurempi kuin z* on kirjoitettava muodossa

```
      if(x .gt. z .or. y .gt. z)
```

eikä

```
      if(x .or. y .gt. z)
```

## 3.3 Silmukat

Yllä esitetyn päätösrakenteen ohella tärkeimpiä kontrollirakenteita ovat silmukat. Seuraavassa esimerkissä ohjelma kysyy käyttäjältä kokonaisluvun `lkm` ja sen jälkeen `lkm` kappaletta lukuja. Ohjelma laskee ja tulostaa lukujen keskiarvon sekä suurimman ja pienimmän syötetyn luvun.

```
      program loop
      integer lkm, i
      real number, min, max, sum, average
C
C  Alkuarvot
      max = -1.0e20
      min = -max
      print*,'Anna lkm'
      read*, lkm
C
```

```
      do 100 i = 1, lkm
       write(6, '(a,i3,a)') 'Syötä ', i,'. luku'
       read(5, *) number
       sum = sum + number
       if(number .gt. max) then
          max = number
         else if(number .lt. min) then
         min = number
       endif
  100 continue
C
      average = sum/lkm
      print*,'Keskiarvo =', average
      print*,'Suurin luku =', max
      print*,'Pienin luku =', min
      end
```

Fortranissa ohjelmasilmukka on siis muotoa

```
      do 200 k = 1, lkm
       ohjelmalohko
  200 continue
```

tai yleisemmin

```
      do 300 z = expr1, expr2, step
       ohjelmalohko
  300 continue
```

Tässä `expr1, expr2`, ja `step` tarkoittavat aritmeettisia lausekkeita tai muuttujia. Silmukkamuuttujan `z` alkuarvo on `expr1` ja jokaisen kierroksen lopussa sitä kasvatetaan `step`:n verran niin kauan kuin se on pienempi tai yhtäsuuri kuin `expr2`. Huomaa, että kierrosten lukumäärä lasketaan silmukan aloitushetkellä, joten silmukan sisällä ei pidä muuttaa esim. `step`:n arvoja, ainakaan se ei vaikuta kierrosten lukumäärään. Jos `expr2` on pienempi kuin `expr1` ja `step` suurempi kuin nolla niin silmukan sisällä olevaa ohjelmalohkoa ei suoriteta lainkaan. Do-silmukka päättyy continue-käskyyn, joka on tyhjä käsky, ts. se ei tee mitään. Sen edessä 'rivinumero' 300 ei tarkoita varsinaista rivinumeroa, se on `viitenumero (label)`, johon viitataan jossain toisaalla

ohjelmassa. Viitenumeron tulee sijaita sarakkeissa 1-5. Tästä syystä viitenumero on ohjelman ulkoasun kannalta huono asia. Valitettavasti fortran-77 standardissa ei ole silmukkaa, joka voitaisiin tehdä ilman viitenumeroita. Tämä puute (töppäys) varmasti on korjattu seuraavassa standardissa (-90). Jo nyt useat fortran-kääntäjät hyväksyvät muotoa

```
do k = 1, lkm
 lohko
enddo
```

olevan silmukan, jossa viitenumero on korvattu enddo-lauseella. Silmukan sisällä oleva ohjelmalohko voi tietenkin sisältää mitä tahansa ohjelmalauseita ja myös sisäkkäisiä silmukoita.

Edellä esitetyn haarautumiskäskyn lisäksi fortranissa on muitakin if-lauseita, mutta suosittelemme, että et opettele niiden käyttöä. If-then-else-lause riittää kaikkiin tilanteisiin. Joskus voi kuitenkin olla mukavampi käyttää yhden rivin if-käskyä, jossa suoritetaan vain yksi ohjelmakäsky ehdon ollessa tosi. Esimerkiksi osa do-silmukassa olevasta ohjelmalohkosta voidaan jättää suorittamatta joillakin kierroksilla, jos jokin ehto tulee todeksi.

```
      do 100 k = 1, lkm
       lohko
       if(ehto) goto 100
         lohko
  100 continue
```

Samantapainen tilanne tulee eteen, kun halutaan lopettaa silmukan pyörittäminen ennenaikaisesti

```
      do 200 k = 1, lkm
       lohko
       if(ehto) goto 201
         lohko
  200 continue
  201 continue
```

Näissä esimerkeissä esiintyy **goto**-lause, jonka käyttöä pitää aina yrittää välttää. Edellisessä tapauksessa se on helposti mahdollista (miten?), mutta koska fortranissa ei ole silmukan keskeytyslausetta (`break`), on keskeytys tehtävä goto-käskyä käyttäen. Goto-hyppy voisi tietenkin tapahtua muuallekin

kuin välittömästi silmukan jälkeen tulevaan lauseeseen. Suosittelemme kuitenkin tällaista rakennetta; nähdessään kaksi peräkkäistä continue-lausetta lukija huomaa heti, että jossain silmukan sisällä voi olla silmukan keskeyttävä break-ehto. Muunlainen menettely johtaa hankalammin luettavaan koodiin eikä siten ole hyvien tapojen mukaista.

Esimerkkiohjelmien `loop` ja `roots2` avulla olemme esitelleet do-silmukan sekä if-then-else lauseen. Nämä ovat fortranin tärkeimmät kontrollirakenteet. Valitettavasti fortran standardi ei tunne ns. while-silmukkaa, jossa silmukan sisällä oleva ohjelman osa suoritetaan toistuvasti niin kauan kuin jokin annettu ehto on tosi eikä niinkuin do-silmukassa, jossa toistojen lukumäärä lasketaan silmukan aloitushetkellä. Ei-standardin mukainen while- silmukka on esim. Vax:in VMS-fortranissa muotoa

```
do while(ehto)
 lohko
enddo
```

Standardi fortranissa tällainen rakenne on muodostettava goto-käskyä käyttäen, esim. seuraavasti

```
C While-silmukka
  111 continue
      if(ehto) then
       lohko
       goto 111
      endif
```

Tämä ja edellämainittu break-tilanne ovatkin ainoat, joissa goto-käskyn käyttö on 'sallittua'.

Ohjelman roots2 alussa olevat kolme riviä ovat ns. kommenttirivejä, jotka erottuvat ohjelmalauseista siten, että rivin ensimmäisessä sarakkeessa on C-kirjain (tai `*`). Mikä tahansa ohjelmarivi voidaan 'kommentoida' pois lisäämällä rivin alkuun c-kirjain. Ohjelman luettavuutta helpottavia kommenttilauseita tulisi käyttää pihistelemättä, toisaalta triviaalien kommenttien lisääminen on turhaa.

Edellä olevissa ohjelmissa esiintyy useita tulostuslauseita, jotka ovat muotoa

```
write(6,'(format-koodit)') muuttujaluettelo
```

Tässä numero 6 on ns. standarditulostusyksikön eli päätteen laitenumero. Tulostus voisi tapahtua myös tiedostoon, jolloin olisi käytettävä jotain muuta numeroa kuin 6 tai 5, joka on standardisyöttölaitteen (pääte) laitenumero. Format-koodit määrittelevät tulostuksen ulkoasun. Esimerkiksi lause

```
write(6,'(f7.2, a, e12.5, a)') x1, 'tekstiä', x2, 'tekstiä'
```

tulostaa muuttujan `x1` arvon 7 merkkiä pitkään kenttään 2:lla desimaalilla, ascii-tekstiä, muuttujan `x2` arvon 12 merkin mittaiseen kenttään 5:llä desimaalilla eksponenttiesityksessä, ja lopuksi taas ascii-tekstiä. Kaikkiaan format-koodeja on runsaasti, mutta muutamallakin tulee toimeen ainakin alkuun. Format-koodit voi myös kirjoittaa erikseen ns. format-lauseeseen, jolloin koodien paikalle write-käskyyn tulee ko. lauseen viitenumero.

Ohjelma `roots2` näyttää suhteellisen hyvältä ohjelmalta, tosin se ei toimi jos esim. a = 0 ja b = 0, ja joissakin tapauksissa se voi antaa epätarkan tuloksen. Ohjelman parantelemisen sijasta viisainta olisikin itse algoritmin tarkasteleminen. Nimittäin toisen asteen yhtälön juuret voit laskea myös vähemmän tunnettua, mutta turvallisempaa reseptiä käyttäen:

$$\begin{aligned} q &\equiv -\frac{1}{2}\left(b + sign(b)\sqrt{b^2 - 4ac}\right) , \\ x1 &= \frac{q}{a} , \\ x2 &= \frac{c}{q} . \end{aligned}$$

Tässä $sign(b) = +1$,jos $b > 0$ ja $-1$ jos $b < 0$.

Vielä yksi ohjelmien ulkoasuun liityvä asia. Fortranissa on harmillinen reikäkorttiaikakaudelta peräisin oleva rajoitus: ohjelmalauseiden tulee sijaita sarakkeissa 7-72. Jos jokin rivi ulottu yli 72:n sarakkeen, niin standardin mukainen fortran kääntäjä jättää sarakkeissa 73,74,... olevat merkit kokonaan huomiomatta. Tällainen möhläys tapahtuu helposti jos kääntäjä tulkitsee rivien alussa olevat näkymättömät tabulaattorimerkit 8:ksi välilyönniksi, paremmat kääntäjät kyllä käsittelevät tällaisen tilanteen fiksummasti, jolloin vältytään hankalasti havaittavilta virheiltä.

# 4 TAULUKOT

Hyödyllisiä ohjelmia ei voi kirjoittaa pelkästään yksittäisiä muuttujia käyttäen, tarvitsemme indeksoituja muuttujia eli taulukoita. Niissä voi käyttää

yhtä tai useampaa indeksiä. Tällaiset muuttujat on aina määriteltävä ennen käyttöönottoa. Esimerkiksi määrittely

```
real vector(5), matrix(5,5)
```

varaa 5 muistipaikkaa muuttujille `vector(1),..., vector(5)` ja $5 \times 5$ muistipaikkaa muuttujille `matrix(1,1), matrix(2,1),..., matrix(5,1), matrix(1,2), matrix(2,2),...,matrix(5,2), matrix(1,3),...,matrix(5,5)`. Kaksi-indeksisen taulukon alkiot sijoittuvat tietokoneen muistiin peräkkäin siten, että 1. indeksi kasvaa nopeammin. Dynaaminen tilavaraus ei ole fortranissa mahdollista, ts. taulukoiden määrittelyssä niiden koko on ilmaistava vakioiden avulla, muuttujia ei voi siihen tarkoitukseen käyttää.

Esimerkkinä taulukoiden käytöstä tarkastelemme kahden vektorin skalaarituloa:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

Tämä voitaisiin laskea seuraavan kaltaisella ohjelmalla.

```
      program scalar
      real a1, a2, a3, b1, b2, b3, scalar
C
      print*,'Anna a:n ja b:n komponentit'
      read*, a1, a2, a3, b1, b2, b3
      scalar = a1*b1 + a2*b2 + a3*b3
      print*,'Skalaaritulo =', scalar
      end
```

Ohjelma kyllä toimii, mutta usein haluamme kirjoittaa tämän tyyppisen ohjelman yleisemmässä muodossa siten, että se laskee skalaaritulon vaikka vektorien komponenttien lukumäärä, tässä 3, ei olisi etukäteen tiedossa. Tällöin on melkein pakko käyttää taulukoita. Olettakaamme, että komponenttien lukumäärä voi olla väliltä 1-100.

```
      program scalar
      real a(100), b(100), scalar
      integer lkm, i
C
      print*,'Anna komponenttien lkm (kork. 100)'
      read(5,*) lkm
      print*,'Anna a:n komponentit'
```

```
      read(5,*)(a(i), i = 1, lkm)
      print*,' ja b:n komponentit'
      read(5,*)(b(i), i = 1,lkm)
C
      scalar = 0.0
      do 100 i = 1, lkm
        scalar = scalar + a(i)*b(i)
  100 continue
C
      print*,'Skalaaritulo =', scalar
      end
```

Ohjelma lukee ensin kaikki `a`-vektorin komponentit taulukkoon `a`, ja sitten `b`:n komponentit taulukkoon `b`. Huomaa lukulauseessa käytetty ns. implisiittinen silmukka: `read(5,*) (a(i), i=1, lkm)`. Tässä esimerkissä haluamme korostaa sitä, että käyttämällä indeksoituja muuttujia voimme kirjoittaa ohjelmat yleisessä muodossa; ylärajan 100 muuttaminen tarvittaessa suuremmaksi on triviaalia. Itse skalaaritulon laskeminen on riippumaton komponenttien lukumäärästä tai siitä tavasta, jolla komponenttien arvot tulevat ohjelmaan. Niinpä olisi paikallaan kirjoittaa funktioaliohjelma skalaaritulon laskemiseksi. Aliohjelma voidaan laatia siten, että se on muista ohjelman osista täysin riippumaton ohjelmayksikkö. Ennenkuin ryhdymme tarkastelemaan aliohjelmia kirjoitamme vielä yhden esimerkin kaksiulotteisten taulukoiden käytöstä.

Oletamme, että tehtävänä on laskea kahden matriisin $A$ ja $B$ tulomatriisi $C$. Tulomatriisihan muodostetaan siten, että sen alkiot $C_{ij}$ saadaan matriisin $A$ $i$:nnen vaakarivin ja matriisin $B$ $j$:nnen pystyrivin skalaaritulona, ts.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Matriisin $A$ vaakavektorien komponenttien lukumäärän on siis oltava sama kuin $B$-matriisin pystyvektorin komponenttien lukumäärä. Oletamme, että matriisien $A$ ja $B$ komponentit on tallennettu vaakariveittäin tiedostoihin A_matr.dat ja B_matr.dat ja tulostamme myös $C$:n komponentit tiedostoon C_matr.dat. Tässä yksi versio ohjelmasta:

```
C  Esimerkki 2-ulotteisten taulukoiden käytöstä
      program matprod
```

```
      real a(100,100), b(100,100), c(100,100)
      integer  ma, na, mb, nb
      integer i, j, k
      character*10 afile, bfile, cfile
C
      afile = 'A\_matr.dat'
      bfile = 'B\_matr.dat'
      cfile = 'C\_matr.dat'
C
      open(11, FILE = afile)
      open(12, FILE = bfile)
C   Oletetaan, että matriisien vaaka- ja pystyvektorien lukumäärät
C   ovat tiedostojen ensimmäisillä riveillä.
C
      read(11,*) ma, na
      read(11,*) ((a(i,j), j = 1,na), i = 1,ma)
      read(12,*) mb, nb
      read(12,*) ((b(i,j), j = 1,nb), i = 1,mb)
      close(11)
      close(12)
C
      if(na .ne. mb) then
       print*,'na ja mb erisuuret:', na, mb
      stop
      endif
C
C   Lasketaan matriisien tulo c = ab
C
      do 100 i = 1, ma
        do 200 j = 1, nb
          c(i,j) = 0.0
          do 300 k = 1, na
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
  300   continue
  200 continue
  100 continue
C
C Tallennetaan c-matriisi
```

```
C
      open(13, FILE = cfile)
      write(13, '(1x,2i4)') ma, nb
      do 400 i = 1, ma
       write(13, '(1x, 100e12.4)') (c(i,j), j = 1, nb)
  400 continue
C
      close(13)
      stop ' Loppu '
      end
```

Ohjelman alussa oleva määrittely

```
real a(100,100), b(100,100), c(100,100)
```

merkitsee sitä, että kullekin muuttujista `a, b ja c` varataan muistista $100 \times 100$ peräkkäistä muistipaikkaa, joihin voidaan viitata indeksien avulla. Voimme siis käsitellä matriiseja, joiden alkioiden lukumäärä 1-10000. Uutena piirteenä tässä on myös se, että ohjelman syöttötiedot tulevat tiedostoista ja myös tulostiedot kirjoitetaan tiedostoon. Tiedoston avauskäskyssä tiedostoon liitetään viitenumero, jota sitten käytetään luku- ja kirjoituslauseissa päätteen laitenumeroiden 5 ja 6 sijasta, muutoin syöttö ja tulostus tapahtuu kuten päätteelle. Tiedostojen käsittelyyn palaamme hieman tuonnempana. Huomaa, että tiedostojen nimet on sijoitettu merkkijono-muuttujiin `afile, bfile ja cfile`. Merkkitiedon tyyppimäärittely tapahtuu `character*`-ilmaisulla, jossa tähden perässä oleva luku ilmoittaa merkkijonon (maksimi)pituuden.

Matriisien alkioiden lukemisessa tiedostosta on edellä käytetty implisiittistä silmukkaa: `read(11,*) (a(i,j), j = 1,na), i = 1,ma)`. Tässä sisemmän silmukan indeksi j juoksee nopeammin, matriisit oletetaan tallennetun vaakariveittäin. Itse matriisitulon laskeminen, joka tarvitsee kolme sisäkkäistä silmukkaa, muodostaa vain pienen osan koko ohjelmasta. Loput on syöttöä ja tulostusta. Input-tiedosto A_matr.dat voidaan helposti muodostaa editorin avulla, sen sisältö voisi olla vaikka seuraavan näköinen:

```
4  6
2.1  -1.3   9.7   1.1   6.3   2.0
0.5   3.2  -2.8  -3.3   2.2   6.5
9.3  12.1  -1.6   7.7  -2.5  14.0
0.1  -1.5  11.1  -3.6  18.1  -7.9
```

Ensimmäisellä rivillä olevat luvut 4 ja 6 ilmoittavat, että matriisissa on 4 vaakariviä ja 6 pystyriviä. Tiedostossa B_matr.dat tulisi olla vastaavasti 6 vaakariviä ja korkeintaan sata pystyriviä. Luvut voi erottaa toisistaan vähintään yhdellä välilyönnillä tai pilkulla. Huomaa, että pientenkin matriisien tapauksessa ohjelma varaa keskusmuistia kolmelle $100 \times 100$-matriisille.

Tässäkin ohjelmassa matriisitulon muodostaminen on muusta ohjelmasta riippumaton kokonaisuus, joten olisi sopivaa laskea se aliohjelmassa. Seuraavassa siirrymmekin tarkastelemaan aliohjelmia, joita on kahta tyyppiä: `function-` ja `subroutine-` aliohjelmat.

# 5 ALIOHJELMAT

Käytännössä ohjelmat laaditaan siten, että niissä on yksi pääohjelma ja useita aliohjelmia. Annettu tehtävä jaetaan sopiviin osatehtäviin, jotka voidaan suorittaa enemmän tai vähemmän toisistaan riippumattomalla tavalla. Aliohjelmat voivat kätkeä sisälleen yksityiskohtia, joiden tarkasteleminen on epäoleellista kokonaisuuden kannalta. Erittäin tärkeää on myös se, että voimme käyttää valmiita (so. muiden tekemiä) aliohjelmia osana omaa ohjelmaamme. Aliohjelmien sujuva ja järkevä käyttö onkin yksi niitä ohjelmointitaitoja, joita haluamme tässä oppaassa erityisesti korostaa.

Tyyppiesimerkkejä aliohjelmien käytöstä saamme edellä tarkastelluista ohjelmista erottamalla vektorien skalaaritulon tai matriisien kertolaskun aliohjelmiksi. N-ulotteisten vektoreiden skalaaritulon laskemisessa käytetty algoritmi ei riipu siitä miten itse vektorit on muodostettu eikä komponenttien lukumäärästä. Vastaavasti matriisien kertolasku tehdään aina samalla tavalla riippumatta muista ohjelman osista. Skalaaritulo on yksi ainoa luku, joten se on kätevää laskea *function*-aliohjelmalla, jolloin sitä voidaan käyttää samalla tavalla kuin mitä tahansa fortranin sisäistä funktiota (sqrt(x), sin(x),...). Sen argumenttina tulee olla kaksi vektoria sekä niiden komponenttien lukumäärä. Tällainen funktio kirjoitetaan seuraavalla tavalla:

```
      function scalar(x, y, n)
      integer n
      real scalar, x(n), y(n)
      integer i
C
      scalar = 0.0
      do 100 i = 1, n
```

```
      scalar = scalar + x(i)*y(i)
  100 continue
      return
      end
```

Function-aliohjelma alkaa `function`-sanalla, jota seuraa funktion nimi ja sen perässä suluissa funktion (muodolliset) argumentit. Senjälkeen ilmoitetaan funktion ja sen argumenttien tyypit, määritellään sisäiset muuttujat ja kirjoitetaan itse ohjelma. Kaikki tämä voidaan sijoittaa omaan erilliseen tiedostoonsa, kääntää erikseen, ja myös testata erillään sen lopullisesta käyttöympäristöstä. Testausta varten on kuitenkin laadittava pääohjelma. Pääohjelmassa, kun vektorit on muodostettu, skalaaritulon arvo voidaan esim. tulostaa käskyllä

```
print*,'Skalaaritulo = ', scalar(X, Y, N)
```

ja muutoinkin funktiota voi käyttää missä tahansa lausekkeissa samalla tavoin kuin muuttujia tai sisäisiä funktioita. On huomattava, että pääohjelmassa (kutsuvassa ohjelmassa) esiintyvät muuttujanimet `X, Y, N` eivät tarvitse olla samoja kuin aliohjelmassa käytetyt `x, y, n`, ainoastaan niiden tyypit, lukumäärä ja järjestys on oltava sama. Aliohjelmassa voi myös esiintyä sisäisiä (paikallisia, lokaalisia) muuttujia, joiden nimet ovat samoja kuin kutsuvassa ohjelmassa mutta merkitys ei ole sama. Function-aliohjelman sisällä funktion nimi esiintyy kuten tavallinen muuttuja ja sille on asetettava arvo ennen `return`-lausetta, jolla aliohjelmasta palataan pääohjelmaan. Tärkeää on huomata, että aliohjelman argumentteina olevien taulukoiden määrittely

```
real x(n), y(n)
```

ei merkitse nyt muistin tilavarausta, vaan ainoastaan ilmoittaa kääntäjälle, että kyseessä on 1-ulotteinen taulukko, jonka komponenttien lukumäärä voi olla mitä hyvänsä. Indeksi n on ns. *dummy*-indeksi, tässä se voitaisiin korvata esim. `1`-llä tai `*`-llä. Funktion argumentteina voi siis olla yksittäisiä muuttujia sekä yksi- tai useampiulotteisia taulukoita. Oleellista on, että argumenttien välityksellä aliohjelmaan tulee kaikki tarvittava tieto funktion arvon laskemiseksi.

Function-aliohjelmaa käytetään silloin kun aliohjelman tulee laskea yksi ainoa arvo, funktion arvo. *Subroutine*-aliohjelmaa on käytettävä, kun halutaan laskea useampia kuin yksi arvo. Matriisien kertolaskun tapauksessa on

laskettava tulomatriisin kaikkien alkioiden arvot. Kun ne lasketaan aliohjelmassa, niin vastaava pääohjelma voi olla sellainen, että se hoitaa vain tarvittavat määrittelyt ja matriisien alkioiden syötön ja tulostuksen. Esim.

```
C Pääohjelma
      real A(100,100), B(100,100), C(100,100)
      integer MA, NA, MB, NB
C
      luetaan matriisit A ja B
C
      call matpro(A, B, C, MA, NA, NB, 100)
C
      tulostetaan matriisi C
C
      end
```

Aliohjelmaan siirtyminen tapahtuu siis `call`-käskyllä ja sieltä palattaessa aliohjelman tulos on taulukossa `C` ja pääohjelman suoritus jatkuu call-lausetta seuraavasta ohjelmakäskystä.

Subroutine-ohjelma voidaan jälleen kirjoittaa haluttaessa erilliseen tiedostoon. Matriisien kertolaskuohjelma on seuraavan kaltainen:

```
C  Matriisien kertolasku
      subroutine matpro(a, b, c, ma, na, nb, MAX)
      integer ma, na, nb, MAX
      real a(MAX,na), b(MAX,na), c(MAX,na)
      integer i, j, k
C
      do 300 i = 1, ma
       do 200 j = 1, nb
         c(i,j) = 0.0
         do 100 k = 1, na
           c(i,j) = c(i,j) + a(i,k)*b(k,j)
 100     continue
 200   continue
 300  continue
      return
      end
```

`Subroutine`-aliohjelman koodi alkaa sanalla `subroutine`, jota seuraa aliohjelman nimi. Nimen ei tarvitse vastata mitään muuttujatyyppiä. Suluissa nimen jäljessä ovat aliohjelman parametrit (muodolliset argumentit), joiden välityksellä kulkee kaikki tieto aliohjelmaan ja sieltä takaisin kutsuvaan ohjelmaan. Muodollisten argumenttien paikoille kutsuvassa ohjelmassa sijoitetaan **todelliset muuttujat**, joiden tyyppien, järjestyksen ja lukumäärän on jälleen vastattava aliohjelman määrittelyjä. Aliohjelmassa määrittely

```
real a(MAX,na), b(MAX,na), c(MAX,na)
```

ei nytkään merkitse tilan varaamista, mutta kaksiulotteisen taulukon tapauksessa ensimmäinen dimensio `MAX` ei ole dummy-indeksi, vaan sen arvon on oltava sama kuin pääohjelman tilavarausmäärittelyssä esiintyvä ensimmäisen dimension lukuarvo, tässä 100. Toinen indeksi, tässä `na`, on edelleen dummy. Tämä johtuu siitä, että fortranissa 2-ulotteiset taulukot ovat muistissa peräkkäisissä muistipaikoissa siten, että ensimmäinen indeksi kasvaa nopeammin. Vastaavasti 3-ulotteisen taulukon tapauksessa aliohjelmaan olisi välitettävä tieto taulukon kahden ensimmäisen dimension todellisista arvoista. Itse asiassa aliohjelmien parametrit tarkoittavat vastaavien muuttujien muistiosoitteita, taulukon nimen välityksellä aliohjelman käyttöön tulee sen ensimmäisen alkion osoite. Matriisin muiden alkioiden muistiosoitteiden laskemiseksi on tiedettävä taulukon ensimmäisen alkion osoite sekä ensimmäinen dimensioluku.

Aliohjelman sisällä sen muodollisia argumentteja voi käyttää kuten tavallisia muuttujia. On vain huomattava, että jos niiden arvoa muutetaan, niin kutsuvan ohjelman vastaava muuttuja saa uuden arvon aliohjelman suorituksen aikana. Tällä tavalla parametrit toimivat tiedonvälittäjinä sekä kutsuvasta ohjelmasta aliohjelmaan että päinvastoin. Luonnollisesti aliohjelmassa itsessään voi olla joukko sisäisiä muuttujia, jotka eivät mitenkään vaikuta kutsuvan ohjelman tai muiden aliohjelmien muuttujiin.

Kun aliohjelmat tehdään riippumattomiksi ohjelmayksiköiksi niistä voidaan muodostaa aliohjelmakirjastoja. Suuri osa tieteellisesta laskennasta tapahtuukin juuri valmiiden ohjelmakirjastojen avulla. Kirjastoista löytyy sopivia aliohjelmia esimerkiksi seuraavantyyppisiä tehtäviä varten:

- lineaarisen yhtälöryhmän ratkaiseminen
- funktioden integrointi
- lajittelu

- funktioiden nollakohtien ja ääriarvojen etsiminen
- tilastolliset menetelmät
- differentiaaliyhtälön ratkaiseminen

Ohjelmoijan tehtävänä onkin usein näiden valmiiden komponenttien yhdistely toimivaksi kokonaisuudeksi. Tämä edellyttää kuitenkin ohjelmointitaidon lisäksi numeeristen menetelmien tuntemusta.

# 6 TIEDOSTOT

Usein ohjelmien syöttö- tai tulostiedot säilytetään tiedostoissa. Tiedostotyyppejä voi olla useita, mutta tavallisesti tarvitsemme ns. *peräkkäis-* ja *suorasaantitiedostoja*. Peräkkäistiedostosta luetaan ja sinne kirjoitetaan peräkkäisessä järjestyksessä rivi (tietue) kerrallaan alusta alkaen, esim. kolmas rivi voidaan lukea vasta kun kaksi edellistä on luettu. Sitävastoin suorasaantitiedostosta voi lukea tietueen kerrallaan mielivaltaisessa järjestyksessä. Edellä esillä olleessa matriisien kertolaskuohjelmassa matriisin alkiot luettiin tiedostosta. Tiedoston avauskäskyssä levyllä olevaan tiedostoon liitetään laitenumero, jota sitten käytetään luku- ja kirjoituskäskyissä päätettä vastaavien laitenumeroiden 5 ja 6 sijasta. Peräkkäistiedoston avauskäsky on yksinkertaisimmillaan muotoa:

```
open(8, FILE = filename)
```

joka liittää numeron 8 tiedostoon *filename*, joka voi olla joko suoraan tiedoston nimi lainausmerkeissä tai merkkimuuttuja, jonka sisältönä on haluttu tiedostonimi. Tämän jälkeen tiedostosta lukeminen tapahtuu käskyllä

```
read(8, *)  muuttujat
```

ja kirjoituskäsky on

```
write(8, '(format koodit)') muuttujat
```

Tiedoston voi kelata alkuun

```
rewind(8)
```

käskyllä ja

```
backspace(8)
```

kelaa tiedostoa taaksepäin yhden tietueen tai rivin verran. Lopuksi tiedoston voi sulkea

```
close(8)
```

lauseella. Ilman close-käskyäkin tiedostot kyllä suljetaan automaattisesti ohjelman suorituksen lopussa käyttöjärjestelmän toimesta.

Peräkkäistiedoston avaamisen voi usein myös jättää käyttöjärjestelmän tehtäväksi. Jos nimittäin ohjelmassa kirjoitetaan laitteelle, jonka numero on eri kuin 5 tai 6, esim.

```
write(12, *) a, b, c
```

vaikka mitään tiedostoa ei ole avattu niin järjestelmä muodostaa automaattisesti levytiedoston ja antaa sille jonkin nimen. Nimi riippuu käyttöjärjestelmästä, UNIX:ssa em. kirjoituskäskyä vastaava nimi olisi `fort.12`. Vastaavasti jos ohjelmassa esiintyy lukukäsky

```
read(9, *) x, y, z
```

niin lukeminen tapahtuu tiedostosta `fort.9`, sen pitäisi tietenkin olla olemassa (oletushakemistossa).

Usein ohjelmissa esiintyy tilanne, että halutaan tallentaa levylle välituloksia tms. tietoja, jotka myöhemmin luetaan takaisin saman tai jonkin toisen ohjelman käyttöön. Tällöin kannattaa tallentaa tiedot formatoimatta eli sellaisina kuin ne esiintyvät ohjelman suorituksen aikana keskusmuistissa. Kirjoituskäskyssä ei tarvita mitään format-koodeja,

```
write(11) a, b, c
```

kirjoittaa muuttujien `a, b, c` arvot tiedostoon 11, josta ne voidaan myöhemmin lukea takaisin lauseella

```
read(11) a, b, c
```

Tässä `a, b, c` voivat olla yksittäisiä muuttujia tai yksi- tai useampi- ulotteisia taulukoita, jos luku- ja kirjoituskäskyt esiintyvät eri ohjelmissa on huolehdittava, että muuttujien tyypit lukukäskyssä vastaavat tarkasti kirjoituslauseessa esiintyviä muttujia.

Suorasaantitiedosto on aina eksplisiittisesti avattava ja avauksen yhteydessä ilmoitettava yhden tietueen (rivin) pituus.

```
      open(7, FILE = 'nimi', ACCESS = 'dir', RECL = 80)
```

avaa suorasaantitiedoston (ACCESS = 'dir'), jonka tietueiden pituus on 80 merkkiä. Nyt tiedostoon kirjoittaminen tapahtuu lauseella

```
      write(7, REC=38) {\it muuttujat}
```

jossa `REC=38` ilmoittaa, että kirjoitetaan 38. tietue tiedostoon, johon numero 7 on liitetty. Vastaavasti lukukäskyssä on ilmoitettava monesko tietue on kysymyksessä. Suorasaantitiedoston oletusmuoto on formatoimaton, joten em. kirjoitus- ja lukulauseissa ei tarvita format-koodeja.

Tiedostojen avaamisen yhteydessä voi esiintyä myös muita määreitä, tavallisia ovat:

```
      UNIT = n  (  n on laitenumero, eri kuin 5 tai 6)
      FILE = tiedostonimi
      ACCES = 'direct' tai 'sequential'
      FORM = 'formatted' tai 'unformatted'
      STATUS = 'old' tai 'new' tai 'unknown'
      RECL = l tietueen pituus (record length)
      ERR = 111 ( 111 on ohjelmalauseen viitenumero)
      IOSTAT = ios  (I*O status)
```

Näiden merkitys on selostettu käsikirjoissa. Usein tiedostoista luettaessa voi olla hyödyllistä varautua tiedoston loppumiseen, se voidaan tehdä lisäämällä lukukäskyyn määre END = *viitenumero* tai ERR = *viitenumero* tai IOSTAT = *ios*, jossa *ios* on kokonaislukumuuttuja. Esim.

```
      read(10, *, IOSTAT = ios) a, b, c
      if(ios .lt. 0) then
C
      tiedoston loppumisen käsittely
      else if(ios .gt. 0)
       lukuvirheen käsittely
      else
       normaali käsittely
      end if
```

# 7 GLOBAALIT MUUTTUJAT

Suositeltavin tapa välittää tietoja aliohjelmien välillä on käyttää aliohjelman argumentteja. Joskus kuitenkin voi olla tarpeen määritellä *'globaaleja'* muuttujia, joihin päästään käsiksi mistä tahansa ohjelmayksiköstä. Globaalit muuttujat sijoitetaan ns. *common*-alueille. Common alueet on ilmoitettva muuttujamäärittelyjen yhteydessä.

```
real alpha, beta, gamma(100)
real x, y, z
common /nimi1/ alpha, beta
common /nimi2/ gamma(100)
...
```

Tässä määritellään kaksi eri common-aluetta, joille annetaan nimet `nimi1` ja `nimi2`, ja niihin sijoitetaan vastaavasti muuttujat `alpha, beta` ja taulukko `gamma`. Aliohjelmissa, joissa halutaan päästä käsiksi common-alueilla oleviin muuttujiin, on myös ilmoitettava halutun common-alueen nimi.

```
subroutine ali1(X, Y, Z)
real X, Y, Z
common /nimi1/ a, b
...
```

Tässä aliohjelmassa voidaan viitata pääohjelman muuttujiin `alpha, beta` nimillä `a, b`, ts. eri ohjelmien globaalien muuttujien nimet eivät tarvitse olla samoja kunhan vain niiden järjestys, tyypit, ja lukumäärä common-lauseissa ovat samat.

Common-alueiden käyttö näyttää kätevältä tavalta välittää tietoa aliohjelmien välillä. Kuitenkin tällaisen menettelyn haittapuolenana on, että se tekee aliohjelmat toisistaan riippuvaisiksi, mikä puolestaan vaikeuttaa ohjelmakokonaisuuden kehittelyä ja ylläpitoa. Sen vuoksi suosittelemme common-alueiden käytön suhteen pidättyvyyttä, useimmiten niitä ei tarvita.

# 8 FORMAT-KOODIT

Koska kaikki tieto tietokoneen muistissa on binäärimuodossa ja päätteellä se esitetään tavallisten ascii-merkkien avulla, on suoritettava konversio sisäisen ja ulkoisen esitysmuodon välillä. Tähän käytetään format-koodeja. Numeerisen tiedon syötössä käytämme yleensä ns. vapaata muotoilua, esim.

```
read(5, *) a, b, c
```

Tässä * format-koodien paikalla ilmaisee, että kyseessä on vapaa 'formaatti'. Tällöin muuttujien a, b, c arvot voidaan syöttää päätteeltä peräkkäin pilkuilla tai välilyönnillä toisistaan erotettuna. Merkkitiedon syötössä voi myös käyttää vapaata muotoilua, mutta silloin input-teksti on sijoitettava heittomerkkien väliin. Muutoin on käytettävä format-koodeja. Esim.

```
character*80 text
...
read(5, '(a80)') text
...
```

Tässä format-koodi on sijoitettu lukukäskyyn heittomerkkien ja sulkujen väliin. Koodi a80 tarkoittaa, että luetaan korkeintaan 80 merkkiä ascii- tekstiä.

Tulostuksen yhteydessä format-koodien käyttö on useimmiten välttämätöntä, koska print*- tai write(6,*)-lauseiden oletusmuotoilu ei ole riittävä. Format-koodeilla ilmoitetaan jokaisen tulostettavan muuttujan ulkoasun tyyppi ja maksimipituus merkkeinä. Esim.

```
write(6, '(i3, f7.2, e12.5)') k, x, y
```

tulostaa muuttujan k arvon koodilla i3 sekä muuttujien x ja y arvot vastaavasti koodeilla f7.2 ja e12.5. Koodi i3 tarkoittaa kokonaislukua, jossa on korkeintaan 3 merkkiä, siis 3-numeroinen positiivinen luku tai miinusmerkki ja 2 numeroa. Koodi f7.2 merkitsee desimaalilukua 7:n merkin mittaisessa kentässä kahdella desimaalilla, desimaalipiste ja mahdollinen etumerkki on myös huomioitava. Muuttujaa y vastaava koodi e12.5 tarkoittaa esitystä, jossa 12 merkkiä pitkään kenttään tulostetaan viidellä desimaalilla eksponenttimuodossa: jos y = 123.45 niin sen arvo tulostuu muodossa 0.12345e+03, jossa eksponenttiosa vie 4 merkkiä ja desimaalipisteelle sekä etumerkille on varattava yhteensä 2 merkkiä. Jos siis tulostetaan negatiivinen luku 5:n merkitsevän numeron tarkuudella, niin tässä muodossa sille on varattava tilaa vähintään 12 merkkiä. Eksponenttiesityksestä on kuitenkin se etu, että sillä voi tulostaa lukuja joiden suuruusluokka ei ole etukäteen tiedossa.

# 9 SISÄISET FUNKTIOT

Koska fortran on nimenomaan laskentaan tarkoitettu ohjelmointikieli, se sisältää melkoisen joukon matemaattisia funktioita, joita kutsutaan kielen si-

säisiksi funktioiksi. Oheiseen luetteloon olemme koonneet niistä tavallisimmat, argumenttina oleva $e$ tarkoittaa jotain lauseketta ja selitysosa kuvaa

| | |
|---|---|
| abs($e$) | itseisarvo |
| aint($e$) | kokonaisosa reaalilukuna |
| int($e$) | kokonaisosa kokonaislukuna |
| real($e$) | konvertoi real-tyyppiseksi |
| dble($e$) | konvertoi real*8-tyyppiseksi (kaksoistarkkuus) |
| cmplx($e1, e2$) | konvertoi complex-tyyppiseksi |
| | |
| mod($e1, e2$) | $e1$ modulo $e2$ |
| max($e1, e2$) | argumenttien maksimi (väh. 2 arg.) |
| min($e1, e2$) | argumenttien minimi (väh. 2 arg.) |
| sqrt($e$) | neliöjuuri |

funktion merkitystä.

| | |
|---|---|
| sin($e$) | sinifunktio |
| cos($e$) | kosinifunktio |
| tan($e$) | tangenttifunktio |
| asin($e$) | sinin käänteisfunktio |
| acos($e$) | kosinin käänteisfunktio |
| atan($e$) | tangentin käänteisfunktio |
| | |
| exp($e$) | eksponenttifunktio |
| log($e$) | luonnollinen logaritmi |
| log10($e$) | kymmenjärjestelmän logaritmi |
| sinh($e$) | hyperbolinen sinifunktio |
| cosh($e$) | hyperbolinen kosinifunktio |
| tanh($e$) | hyperbolinen tangenttifunktio |

Merkkitiedon manipulointia varten on tarjolla ainakin seuraavat standardifunktiot

| | |
|---|---|
| len($s$) | merkkijonon s (määritelty) pituus |
| char($i$) | ascii-koodia $i$ vastaava ascii-merkki |
| ichar($s$) | merkkijonon 1.:n merkin ascii-koodi |
| index($s1, s2$) | alimerkkijonon $s2$ alkamiskohta $s1$:ssä |

Useimmat fortranit sisältävät näiden lisäksi enemmän tai vähemmän standardina olevia matemaattisia funktioita ja merkkifunktioita.

# 10 LOPUKSI

Kuten aluksi totesimme tämä opas ei edes yritä selostaa fortranin kaikkia piirteitä. Käsittelemättä ovat jääneet ainakin seuraavat lauseet:

```
aritmetic if
assigned goto
computed goto
assign
block data
dimension
equivalence
external
implicit
intrinsic
namelist
parameter
save
pause
stop
```

Näistä kolme ensimmäistä on sellaisia lauseita, joita ei kannata käyttää jos haluaa kirjoittaa luettavaa ohjelmaa. Dimension-lauseella määritellään taulukoita, esimerkiksi `dimension a(100)` tarkoittaa samaa kuin `real a(100)`. Equivalence ja external ovat myös määrittelylauseita, edellistä tarvitaan kun halutaan käyttää useampia muuttujanimiä viittaamaan samaan muistipaikkaan ja jälkimmäistä tarvitaan jos aliohjelman argumenttina on toisen aliohjelman nimi. Parameter-lause määrittelee symbolisia vakioita, joita on kätevä käyttää esim. taulukkomäärittelyissä. Pause pitää tauon ja stop lopettaa ohjelman suorituksen.

Nämä ja muutamat muutkin asiat ehtii oppia kunhan ensin oppii ohjelmoinnin perusteet, joihin hyvin oleellisesti kuuluu aliohjelmien ja taulukoiden sujuva käyttö ja tietenkin selkeät ohjelmarakenteet. On syytä muistaa, että ohjelman ulkoasu ja luettavuus ei ole pelkästään esteettinen asia, selkeän ohjelman kirjoittaminen pakottaa myös ajattelemaan selkeästi.