8101160 OHJELMIEN TODISTAMINEN

3 ov

Antti Valmari

Tampereen teknillinen korkeakoulu
Ohjelmistotekniikan laitos
PL 553, 33101 TAMPERE

huone TF 209
puh. 3115 4321
sähköposti Antti.Valmari@tut.fi
veppisivu http://www.cs.tut.fi/~ava/

13.1.2005

KURSSIN JÄRJESTELYT

Uutisryhmä: tut.ot.ohjtod

Veppisivu:

http://www.cs.tut.fi/kurssit/8101160/toteutus.2005.html

Suoritus: laskuharjoituksilla

- pistejärjestelmä epämuodollinen

Kirjallisuus: luentomonisteet + laskuharjoitusmoniste

- mikään kirja ei kata kurssin aluetta kokonaan
- aineistoa parannetaan syksyn aikana
- jakelu vepin kautta
- oheislukemiseksi sopivia kirjoja:
  - R. C. Backhouse: *Program Construction and Verification*, Prentice-Hall 1986
  - D. Gries: *The Science of Programming*, Springer-Verlag 1981
  - (D. Gries, F. B. Schneider: *A Logical Approach to Discrete Math*, Springer-Verlag 1993)
  - ((N. Francez: *Program Verification*, Addison-Wesley 1992))

SISÄLLYSLUETTELO

1 JOHDANTO

Tällä kurssilla käsitellään ohjelmanpätkien ja algoritmien matemaattisen täsmällistä

- spesifiointia
- oikeaksi osoittamista

Tällä kurssilla päähuomio on *semantiikan* eli merkityksen matemaattisen täsmällisessä käsittelyssä

- *syntaksin* eli oikeinkirjoitussääntöjen täsmällistä käsittelyä käsitellään muilla kursseilla
  - helpompaa

Ohjelmistotuotantoon liittyvää menetelmää, jossa sekä syntaksi että semantiikka ovat matemaattisen täsmällisiä, kutsutaan *formaaliksi menetelmäksi*

⇒ tällä kurssilla ei varsinaisesti käsitellä formaaleja menetelmiä

- tämän kurssin jälkeen on helppo ymmärtää ja ottaa käyttöön formaaleja menetelmiä

Formaaleja menetelmiä on kehitetty moniin tehtäviin

- vaatimusten asettelu, määrittely, suunnittelu, toteutus, oikeaksi osoittaminen, testaus jne.
- tämän kurssin opit auttavat ennen kaikkea alleviivattuja

Tässä luvussa

- perustellaan oikeaksi osoittamisen tärkeyttä
- luodaan yleissilmäys formaalien menetelmien olemukseen ja nykytilaan
- asetetaan kurssin tavoitteet

1.1 Miksi oikeaksi osoittamista tarvitaan?

Esimerkki

- R. Backhousen oppilas palautti seuraavan Pascal-ohjelman, kun tehtävänä oli verrata merkkijonojen samuutta:

```
onsama := ( jono1.pituus = jono2.pituus );
if onsama then
    for i := 1 to jono1.pituus do
        onsama := jono1.merkki[i] = jono2.merkki[i];
write(onsama)
```

- testaillaanpa …
  - 'korkeakoulu' 'korkeakoulu' ➠ true ·/·
  - 'kurssi' 'kurssi' ➠ true ·/·
  - '' '' ➠ true ·/·
  - 'korkeakoulu' 'kurssi' ➠ false ·/·
  - 'luento' 'kurssi' ➠ false ·/·
- hyvin näkyy toimivan!
- vielä yksi testi:
  - 'toimiva' 'ohjelma' ➠ true ?!?!?!
  - ohjelma tulostaa “true”, jos jonoilla on sama pituus ja sama *viimeinen merkki*
  - opiskelijan vastaus:
    “But it worked last Tuesday!”

Oliko opiskelija testannut ohjelman huolimattomasti?

- hän oli käynyt läpi kolme testisarjaa:
  (a) eripituisia jonoja
  (b) samanpituisia, erilaisia jonoja
  (c) samanlaisia jonoja
  ja saanut aina oikean tuloksen

⇒ testausta ei voi sanoa huolimattomaksi!

Miten paljon ohjelmaa olisi pitänyt testata, että virhe olisi todennäköisesti paljastunut?

- lähtökohta: testaaja ei osaa ennakolta epäillä mitään tiettyä jonoa
  - jos osaisi, virhe paljastuisi heti järkeilemällä
- virhe ilmenee vain tapauksessa (b)
- laskemme virheen havaitsemistodennäköisyyden satunnaisella (b)+(c)-tyypin testiaineistolla
- jos erilaisia merkkejä on m ja jonon pituus on k, on virheen havaitsemistodennäköisyys yhdessä testissä

$$\frac{1}{m} - \frac{1}{m^k} < \frac{1}{m}$$

  ja n testissä alle

$$1-\left(1-\frac{1}{m}\right)^n$$

- olkoon $m = |\{'a', \ldots, 'z'\}| = 26$
  - jotta hav.tod.näk. ≥ 0,5, tarvitaan ≥ 18 testiä
  - jotta hav.tod.näk. ≥ 0,9, tarvitaan ≥ 59 testiä

Huomautuksia

- tuskin kukaan jaksaa testata näin pientä ohjelmanpätkää edes 18 kertaa

- jos ohjelmanpätkä on tarkoitettu käytettäväksi, se ajetaan melko varmasti ainakin 59 kertaa
- isommilla ohjelmilla virheen löytymistodennäköisyys testauksessa ja ilmenemistodennäköisyys tuotantoajossa ovat yleensä pienempiä
- testausajan suhde tuotantoajoihin käytettyyn aikaan on kuitenkin yleensä aika pieni
  ⇒ virheen ilmenemistodennäköisyys tuotantoajossa on paljon suurempi kuin testauksessa
- ohjelman käytös saattaa muuttua, kun ohjelma siirretään testiympäristöstä tuotantokäyttöön
  - eri kone ja käyttöjärjestelmä
  - erilainen kuormitus
- ohjelmien käyttäytymistä ei voi ekstrapoloida luotettavasti
  - vrt. jos silta kestää 10 tonnin kuorman, se kestää kaikki pienemmät kuormat
  ⇒ sillan suunnittelussa voi käyttää varmuuskertoimia: pannaan paksumpi palkki
  - mitä olisi varmuuskerroin loogiselle informaatiolle? puhelinnumeroon 20% lisää?

⇒ *jos ohjelman toiminta varmistetaan vain testaamalla satunnaisella aineistolla, on melkein varmaa, että tuotantoajossa ilmenee virheitä*

- kokenut ohjelmoija näkee yllä olevan virheen suoraan koodia lukemalla, ilman testejä

⇒ järkeily on usein testausta paljon tehokkaampaa

- johtopäätös:

  pelkkä sokea testaaminen ei riitä, ohjelman toimintaa on tutkittava myös järkeilemällä

Vastaväite:

> “huolellinen testaaja ei käytä puhtaasti satunnaista aineistoa, vaan koettaa taata, että ohjelman kaikki haarat testataan”

- usein on vaikeaa tai työlästä pakottaa ohjelma käymään kaikki haarat läpi
  - esim. virhetilanteiden käsittelykoodi
  - ⇒ osa ohjelmakoodia jää helposti kokonaan testaamatta
  - mitä tuloksia olette saaneet kattavuusanalyyseistä?
- Backhousen oppilaskin kävi kaikki haarat läpi
  - jako (a) – (b) – (c)
  - testiaineisto kävi läpi jokaisen koodirivin

⇒ kaikkien haarojen läpikäynti ei takaa kattavuutta

⇒ jotta voitaisiin olla edes jossain määrin varmoja testauksen kattavuudesta, on ohjelman toimintaa tutkittava *myös* järkeilemällä

Vastaväite:

> “matemaattinen täsmällisyys on liian hankalaa ja kallista”

- suurelta osin totta
  - ⇒ sitä kannattaa käyttää tiukasti vain, kun laatu ja luotettavuus ovat erityisen tärkeitä
- suurelta osin vain koulutuskysymys
  - suuri sellainen!

- kurssin tekniikoita voi ja kannattaa soveltaa eri täsmällisyystasoilla tilanteen mukaan
  - hyvin tarkasti kriittisissä ohjelmakohdissa
  - epämuodollisesti normaalissa ohjelmoinnissa
  - matemaatikon tapaan algoritmisuunnittelussa

  jotta osaisi soveltaa sopivan epätäsmällisesti, on osattava soveltaa hyvin täsmällisesti

Joka tapauksessa ohjelmoidessasi järkeilet ainakin vähän

- katselmointikin perustuu järkeilyyn, ja on havaittu hyvin tehokkaaksi
- nyt voit opetella järkeilemään enemmän, tarkemmin ja järjestelmällisemmin

Ohjelmointihaasteita: saatko nämä toimimaan luotettavasti ja nopeasti ilman kurssin keinoja?

- suurimman 0-neliön etsiminen matriisista, jonka alkiot ovat 0:ia ja 1:iä

0	1	1	0	0	0	1
0	0	1	0	1	1	0
1	1	0	0	0	1	1
1	0	0	0	0	1	0
0	1	0	0	0	0	0
0	0	0	1	0	0	0

- alkioiden poisto taulukosta siten, että jäljelle jäävät muodostavat mahdollisimman pitkän aidosti kasvavan jonon
  - [3,1,6,1,9,4,5,8,1,2,4,0] ➠ [–,1,–,–,–,4,5,8,–,–,–,–]

- murtolukulaskimen teko
  - [2/5] * (1[2/3] + 4[1/3]) ➠ 2[2/5]
- nopeimman junayhteyden etsiminen, kun lähtöaika saa olla mikä tahansa
  - ⇒ odotusta lähtö- ja määräasemilla ei lasketa, mutta odotus vaihtoasemilla lasketaan
- resurssin (esim. kirjoitin) jako usealle asiakkaalle siten, että resurssi on joka hetki enintään yhdellä asiakkaalla, ja jokainen vuoroaan pyytänyt asiakas lopulta saa sen

Harjoitustehtäviä

Näiden tehtävien tarkoituksena on näyttää, minkä tyyppisissä ohjelmointiongelmissa ohjelmien oikeaksi todistamisen tekniikoista on apua. Osa näistä tullaan tekemään uudelleen kun konstit on opittu.

1. (a) Seuraava quicksort-algoritmin toteutus on erään C++-oppikirjan esimerkkinä:

```
void qsort( int *ia, int low, int high ) {
   if ( low < high ) {
      int lo = low;
      int hi = high + 1;
      int elem = ia[ low ]; // jakoarvo
      for (;;) {
         while ( ia[ ++lo ] < elem ) ;
         while ( ia[ --hi ] > elem ) ;
         if ( lo < hi )
            swap( ia, lo, hi ); // = vaihda
         else break;
      } // end, for(;;)
      swap( ia, low, hi );
      qsort( ia, low, hi - 1 );
      qsort( ia, hi + 1, high );
   } // end, if ( low < high )
}
```

   Toteutuksessa on virhe, jota kirjan tekijä ei näytä huomanneen. Mikä?

(b) Arvioi sanallisesti mahdollisuutta löytää virhe testaamalla. Arvioi mahdollisuus myös sille, että virhe ei koskaan ilmene ohjelmaa käytettäessä.

2. (a) Ennen seuraavan puolitushakuohjelman alkua taulukolle $A[1…n]$ pätee $A[1] \le A[2] \le \ldots \le A[n]$. Ohjelma yrittää löytää suurimman a siten, että $A[i] < avain$ aina kun $1 \le i < a$. Etsi virhe.

```
a := 1; y := n;
while a < y do
    v := (a+y) div 2;
    if A[v] < avain then a := v+1
    else y := v
    endif
endwhile
```

(b) Arvioi todennäköisyyttä sille, että virhe ilmenee m:ssä testi- tai tuotantoajossa, jos $A[1], \ldots, A[n]$ ovat kaikki eri suuria, ja *avain* on joku niistä.

(c) Kuten (b), mutta nyt kaikki $A[1], \ldots, A[n]$ ja *avain* ovat eri suuria.

3. Löytääkö seuraava ohjelma mahdollisimman pitkän aidosti kasvavan jonon $B[1…k]$, joka on tehty $A[1…n]$:stä alkioita poistamalla? Anna perustelu tai vastaesimerkki.

```
k := 0;
for i := 1 to n do
    h := 1; C[1] := A[i];
    for j := i+1 to n do
        if A[j] > C[h] then h := h+1; C[h] := A[j]
        endif
    endfor
    if h > k then
        k := h;
        for j := 1 to k do B[j] := C[j] endfor
    endif
endfor
```


4. Voidaanko solmun u korkeus (= pisimmän lapsettomaan solmuun vievän polun pituus) selvittää alla olevilla algoritmeilla seuraavasti (a) puussa (b) silmukattomassa suunnatussa graafissa (c) suunnatussa graafissa? Solmun lähtökaaret on linkitetty listaksi *outedges*. Aluksi kaikissa solmuissa $m = -1$. Anna perustelu tai vastaesimerkki.

```
function Korkeus1(u) is
if u^.m = –1 then
    u^.m := 0; a := u^.outedges;
    while a ≠ Nil do
        x := Korkeus1( a^.head ); a := a^.next;
        if x ≥ u^.m then u^.m := x+1 endif
    endwhile
endif
return u^.m
```

```
function Korkeus2(u) is
if u^.m = –1 then
    k := 0; a := u^.outedges;
    while a ≠ Nil do
        x := Korkeus2( a^.head ); a := a^.next;
        if x ≥ k then k := x+1 endif
    endwhile
    u^.m := k
endif
return u^.m
```


1.2 Täsmällisyys ja formaalius

Mitä “formaali” tarkoittaa?

- sanakirja: “formal” = muodollinen, kaavamainen, virallinen
- matematiikassa: merkitykseen nojautumaton
  - formaali laskeminen on vain sääntöjen soveltamista kaavoihin, eräänlainen merkkipeli
  - kaavan sisältöä ei ajatella: säännön mukaan saa laskea, vaikka se johtaisi ilmeisen “väärään” tulokseen
  - edes selvästi “oikeaa” laskua ei saa tehdä, jollei löydy sääntöä, joka antaa siihen luvan

⇒ jotta vältettäisiin älyttömyydet, säännöstö on suunniteltava ja sitä on käytettävä huolella
  - sääntöjen on oltava hyvin täsmällisiä
  - niitä on noudatettava hyvin täsmällisesti
  - säännöt on täytynyt valita huolella

- ohjelmistotuotannon kannalta
  - *formaali* = matemaattisen täsmällinen
  - *formaali menetelmä* = ohjelmistotuotannossa käytettävä menetelmä, jossa kohde ilmaistaan ja siihen kohdistuvat operaatiot toteutetaan matemaattisen täsmällisesti

Formaaleja ohjelmistotuotannon menetelmiä on mm. seuraaviin tehtäviin

- järjestelmän, ohjelmiston, ohjelmanpätkän, tietorakenteen, tiedon esitystavan, protokollan jne. määrittely
- toteutuksen (esim. ohjelman) johtaminen määrittelystä

- ohjelman muuntaminen (esim. tehokkaammaksi) sen merkitystä muuttamatta
- ohjelman (loogisten) ominaisuuksien tutkiminen
- ohjelman osoittaminen oikeaksi
- testiaineiston muodostus
- testaamisen automatisointi

Tällä kurssilla käsitellään ohjelmanpätkien ja algoritmien spesifiointia ja oikeaksi osoittamista

Koska formaalin menetelmän sääntöjä sovelletaan merkitystä ajattelematta, on yksittäisen säännön soveltaminen mekaanista työtä

- vrt. shakkipelin siirron toteuttaminen
- vaatii suurta täsmällisyyttä

⇒ ihmiselle kurjaa, tietokoneelle helppoa

Formaali säännöstö ei määrittele, mitä sääntöä ja mihin on sovellettava seuraavaksi, jotta päästäisiin tavoitteeseen

- vrt. shakkipelin siirron valinta
- vrt. sieventäminen matematiikassa: $a(b+c) = ab + ac$

⇒ säännön valinta jättää tilaa luovuudelle

- vaikea pulma ohjelmistotuotannon formaalien menetelmien automatisoinnille

⇒ kaiken kaikkiaan formaalit menetelmät ovat vaikeita sekä ihmisille että koneille

Kaikkien muiden formaalien menetelmien lähtökohtana on formaali määrittely

- määrittelyyn liittyy kaksi osaa
  - *syntaksi*: mitkä tekstit, kaaviot yms. tarkoittavat ylipäänsä mitään
  - *semantiikka*: mitä syntaktisesti oikeat tekstit ym. tarkoittavat
- kumpikin erikseen voi olla formaali tai epäformaali
  - esim. ohjelmointikielten syntaksi on yleensä määritelty formaalisti ja semantiikka epäformaalisti
  - esim. matemaattisten merkintöjen kohdalla tilanne on yleensä päinvastoin
- tietokoneella käsittely edellyttää, että ainakin syntaksi on määritelty formaalisti (ainakin melkein …)

Syntaksin formaaliin määrittelemiseen keksittiin hyviä keinoja jo 1960-luvulla

- pian löydettiin myös hyviä algoritmeja moniin syntaksin käsittelyyn liittyviin tehtäviin
  - ennen kaikkea jäsentäminen

⇒ pelkkää syntaksin formaalia käsittelyä ei nykyisin kutsuta “formaaliksi menetelmäksi”

- siis termi “formaali menetelmä” vaatii, että sekä syntaksi että semantiikka on määritelty formaalisti
  - markkinamiesten puheet ovat tietysti erikseen
- syntaksin formalismeja käsitellään kursseilla
  - 8101010 Lausekielten toteutustekniikka
  - 8101100 Johdatus tietojenkäsittelyteoriaan → 8100500 Ohjelmistotekn. matem. menetelmät

Semantiikan formaali käsittely on vaikeaa ja innokkaan tutkimuksen kohde

- tämän kurssin kohde on semantiikan käsittely koodi-, algoritmi- ja määrittelytasolla
- rinnakkaisohjelmien semantiikkaa on esim. “8101150 Rinnakkaisten järjestelmien ulkoinen käyttäytyminen”
- spesifikaatiotaso mm. Z-, VDM-seminaareissa

Vaikka semantiikka on tällä kurssilla formaali, ei sitä eikä ohjelmia, määritelmiä jne. esitetä formaalilla syntaksilla

⇒ ei formaalien menetelmien kurssi

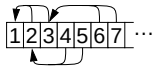
- matemaatikon vapaus
- semantiikan osalta hyvin paljon yhteistä aidosti formaalien menetelmien kanssa

Sana *puoliformaali* (*semiformal*) tarkoittaa samaa kuin epäformaali

- vrt. puolitotuus
- **ei päde** formaali = tietokoneen luettava

Yksinkertainen esimerkki ohjelman formaalista muuntamisesta

- taulukko $A[1, \ldots, n]$ on *keko*, joss $A[\lfloor i/2 \rfloor] \geq A[i]$ aina kun $2 \leq i \leq n$
  - $\lfloor i/2 \rfloor$ on $i/2$ pyöristettynä alas kokonaisluvuksi



- oheinen algoritmi muuntaa A:n keoksi

```
for i := ⌊n/2⌋ downto 1 do
    j := i; x := A[i]
    repeat
        old := j; j := 2·j
        if j < n and A[j+1] > A[j] then
            j := j + 1 endif
        if j ≤ n and A[j] > x then A[old] := A[j] endif
    until j > n or A[j] ≤ x
    A[old] := x
endfor
```

- keon määritelmä ja algoritmi on muutettava niin, että taulukkoa indeksoidaan 0, …, n – 1 kuten C++:ssa
- koetetaanko ensin muuntaa ohjelma ilman alla olevia neuvoja?

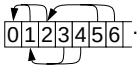
1. keon määritelmän muunto

- liitetään jokaiseen indeksointiin “– 1” muuta muuttamatta
  ⇒ indeksointialue siirtyy halutulla tavalla

  $A[\lfloor i/2 \rfloor - 1] \geq A[i-1]$ aina kun $2 \leq i \leq n$
- korvataan i kaikkialla $(i+1)$:llä
  ⇒ sieventäminen käy mahdolliseksi

  $A[\lfloor (i+1)/2 \rfloor - 1] \geq A[(i+1)-1]$ aina kun $2 \leq (i+1) \leq n$
- sievennetään

  $A[\lfloor (i-1)/2 \rfloor] \geq A[i]$ aina kun $1 \leq i \leq n-1$



2. algoritmin muunto

- liitetään jokaiseen indeksointiin “– 1” muuta muuttamatta

```
for i := ⌊n/2⌋ downto 1 do
    j := i; x := A[i−1]
    repeat
        old := j; j := 2·j
        if j < n and A[j+1−1] > A[j−1] then
            j := j + 1 endif
        if j ≤ n and A[j−1] > x then
            A[old−1] := A[j−1] endif
    until j > n or A[j−1] ≤ x
    A[old−1] := x
endfor
```

- korvataan i, j ja *old* kaikkialla “i+1”:llä jne.
  ⇒ tullaan pääsemään eroon monesta “– 1”:stä
  – syntyy “epäkoodia”, vaan siitä tullaan selviämään

```
for i+1 := ⌊n/2⌋ downto 1 do
    j+1 := i+1; x := A[i+1−1]
    repeat
        old+1 := j+1; j+1 := 2·(j+1)
        if j+1 < n and A[j+1+1−1] > A[j+1−1]
            then j+1 := j+1 + 1 endif
        if j+1 ≤ n and A[j+1−1] > x then
            A[old+1−1] := A[j+1−1] endif
    until j+1 > n or A[j+1−1] ≤ x
    A[old+1−1] := x
endfor
```

- sievennetään
  - sijoituksia käsitellään kuten yhtälöitä
  - $j+1 := 2\cdot(j+1)$ ➠ $j := 2\cdot(j+1)-1 = 2\cdot j+1$
  - $j+1 \leq n$ ➠ $j < n$

```
for i := ⌊n/2⌋ − 1 downto 0 do
    j := i; x := A[i]
    repeat
        old := j; j := 2·j+1
        if j < n−1 and A[j+1] > A[j] then
            j := j + 1 endif
        if j < n and A[j] > x then
            A[old] := A[j] endif
    until j ≥ n or A[j] ≤ x
    A[old] := x
endfor
```

- vertaa lopputulosta alkuperäiseen!

Täsmällisten ja täysin formaalien menetelmien hyötyjä

- määritelmien ym. dokumenttien laatu paranee
  - yksikäsitteisyys
  - tarkkuus
  - ymmärrettävyys — ainakin joskus!
- määritelmistä saadaan riippumattomia toteutuksesta
  - “mitä”-tasolle “miten”-tason sijaan
- jos syntaksikin on formalisoitu, niin dokumentteja voi käsitellä tietokoneella
  - sisäisen johdonmukaisuuden tarkistus
  - simulointi / animointi
  - erilaiset tavalliset ja älykkäät testit
  - oikeaksi todistaminen
- formaalisti johdettu tai oikeaksi osoitettu ohjelma täyttää “varmasti” määritelmänsä
  - pohdittavaksi: kuinka varmasti?
- ohjelmien formaali muuntelu sallii esim. nopeuden kasvattamisen toimintaa vahingossa rikkomatta
- testaus tehostuu ja monipuolistuu

⇒ ohjelmien
  - suunnitteluprosessi helpottuu (tai ei!)
  - huollettavuus paranee
  - virheet vähenevät
- on havaittu, että jo pelkkä täsmällinen määritteleminen vähentää merkittävästi virheitä
- formaalien menetelmien perusideoiden osaaminen parantaa merkittävästi myös epäformaalin ohjelmoinnin laatua

Haittoja

- työläitä
  - tai siltä ainakin alkuvaiheessa tuntuu
  - lisävaiva maksaa itsensä myöhemmin takaisin — tai ei maksa
- vaativat uudenlaista ajattelua ja uusia taitoja
  - vaikea oppia (varsinkin uraantuneiden)
  - ⇒ koulutettua henkilökuntaa niukasti saatavana
- eivät vielä(kään) kypsiä eivätkä vakiintuneita
- työkaluja ym. tukea niukasti saatavana
- osa menetelmistä laskennallisesti vaikeita
  - tietokoneella tehtynä vaatii suunnattomasti laskentakapasiteettia
    - ⇒ riittävän tehokkaita työkaluja ei saatavana
  - ihmisen tekemänä vaatii matemaattista luovuutta
    - ⇒ ei onnistu perusinsinööriltä

Lupaavimpia sovelluskohteita

- turvallisuuskriittiset järjestelmät
  - ydinvoimalan prosessinohjaus, lentokoneen ohjaus, sairaalalaitteet, …

- hankalasti korjattavat järjestelmät
  - avaruusluotaimet, piilastut, puhelinverkon protokollat, …
  - virheen korjaaminen käy hirvittävän kalliiksi
- reaktiiviset järjestelmät
  - esim. mekatroniikan ohjaus, monen käyttäjän järjestelmät, tietoliikenne
  - rinnakkaisuuden hallinta on osoittautunut ihmiselle hyvin hankalaksi
- kirjastoitavat ohjelmanpätkät
  - toimittava monenlaisissa ympäristöissä
    ⇒ vaikea testata

Täysin formaalit menetelmät ovat yleistyneet hyvin hitaasti

- on olemassa yrityksiä, joiden toimiala on formaaleja menetelmiä tukevien työkalujen ja / tai palvelujen myynti
  - Praxis, UK; Meta Software, USA; Formal Systems, UK; Telelogic, Ruotsi; …
- on olemassa merkittäviä käyttäjiä
  - Inmos on ainakin vuodesta 1989 käyttänyt formaaleja menetelmiä piilastujen suunnittelussa
  - Lucent Bell Labs:lla on vahva protokollien automaattiseen analysointiin keskittynyt ryhmä
  - Intel havahtui asiaan Pentium liukulukuvirheen jälkeen, ja on nyt suuri käyttäjä
  - Nokia on osoittanut kiinnostusta rinnakkaisjärjestelmien formaaleihin menetelmiin
  - …

- viranomaisten määräykset saattavat nyt tai tulevaisuudessa suoraan tai epäsuorasti (ankara tuotevastuu) pakottaa formaalien menetelmien käyttöön
  - UK:n puolustusministeriön standardi 00-55 (1991) vaatii formaalien menetelmien käyttöä
- näyttäviä sovelluksia on raportoitu
- kaikesta huolimatta käyttö on vielä tälläkin hetkellä (2005) vähäistä
- alue on laaja, hajanainen ja epäkypsä

Täysin formaalien menetelmien huonon menestyksen syitä

- nippeliyksityiskohtien formalisointi on välttämättöntä
- nippeliyksityiskohtien formalisointi teettää valtavasti työtä, mutta antaa vain vähän vastineeksi
  - periaatteessa tie monipuoliseen automaattiseen käsittelyyn
  - vain yksinkertaisia asioita tekevät automaatit toimivat käytännössä, muiden teho ei riitä
- ohjelmistovirheitä suvaitaan usein aika hyvin
  - kukaan ei huomaa jos joka 10 000. kännykkäpuhelu jää laskuttamatta
  - asiakkaat ovat tottuneet näkyviinkin virheisiin
  - ⇒ virheiden poistosta ei kannata maksaa

Matemaattinen täsmällisyys ilman syntaksin formalisointia on hyvä kompromissi

- tuo suuren osa formaalien menetelmien lupaamasta laadun parannuksesta
- epäolennaisia yksityiskohtia ei tarvitse formalisoida
- formalismin voi valita joustavasti

Vertailu muihin aloihin

- useimmat tekniikan alat hyödyntävät tehokkaasti matematiikkaa tai matemaattisia teorioita
  - rakennustekniikka: lujuuslaskenta, materiaalien ominaisuudet, …
  - elektroniikka: sähköoppi, lämmön virtaus, …
  - …
- tekniikan alat ovat yleensä alkaneet käsityötaitona ja perimätietona
- vähitellen on kehitetty teorioita tärkeiden ilmiöiden ymmärtämiseksi
  - esim. Maxwellin yhtälöt sähkömagnetismissa
- suunnittelun helpottamiseksi on kehitetty erilaisia laskenta- yms. menetelmiä
  - esim. Laplace- ja Fourier-muunnokset
- nykyaikana menetelmiä on toteutettu tietokoneohjelmina, jopa niin, että käyttäjän ei tarvitse ymmärtää menetelmän toimintaa kovin syvällisesti
  - esim. osittaisdifferentiaaliyhtälöiden numeerinen ratkaiseminen
  - esim. elektroniikkapiirien simulointiohjelmistot
  - esim. tilastolliset ohjelmistot

nikkarointi → *tiede* → *automatisointi*

- ohjelmistotuotanto on uskoakseni valmis siirtymään nikkaroinnista tieteeksi, mutta ei vielä valmis automatisoitavaksi

⇒ ohjelmistomatematiikka toimii, formaalit menetelmät eivät

Ohjelmistomatematiikan yleinen osaamistaso on Suomessa heikonlainen verrattuna moniin muihin maihin

1.3 Kurssin tavoite ja sisältö

Edellä mainituista syistä

- ei kannata keskittyä mihinkään yksittäiseen formaaliin menetelmään
- kannattaa keskittyä asioihin, joita löytyy monista formaaleista menetelmistä
  - määrittely logiikalla ja joukko-opilla
  - päättely
- kannattaa säilyttää matematiikan joustavuus
  - kaikkea ei formalisoida

Ohjelmien todistaminen

- edellyttää määrittelyä logiikalla
- edellyttää päättelyä

⇒ harjaannuttaa useimpien formaalien menetelmien perustaitoja

- suhteellisen vakiintunut teoriapohja

⇒ sisältö tuskin heti vanhenee

⇒ kurssi keskittyy siihen

Sisältö

- ohjelman tila ja miten siitä puhutaan
- peruslogiikan ja joukko-opin kertausta
- ohjelmanpätkien ominaisuuksista puhuminen logiikalla ja joukko-opilla
- ohjelmien oikeaksi todistaminen
- todistamisen sovelluksia
- algoritmien todistaminen

2 OHJELMAN MÄÄRITTELY

Tässä luvussa

- opetellaan puhumaan ohjelman tilojen ominaisuuksista tilapredikaateilla
- kootaan ohjelman spesifikaatio tilapredikaateista
- pohditaan, miten tilapredikaatin ja spesifikaation saa sanomaan oikeat asiat

Vaikka sovellusalueena on ohjelman tila, monet esitetyistä asioista pätevät laajemminkin

Kirjallisuutta

- luvun sisältö ei vastaa suoraan mitään kirjaa, mutta seuraavista opuksista saattaa olla hyötyä:
  - R. C. Backhouse: *Program Construction and Verification*, Prentice-Hall 1986
  - D. Gries: *The Science of Programming*, Springer-Verlag 1981
- seuraavassa kirjassa matematiikkaa opetetaan poikkeavalla, tietojenkäsittelijöille sopivalla tavalla:
  - D. Gries, F. B. Schneider: *A Logical Approach to Discrete Math*, Springer-Verlag 1993

2.1 Ohjelman tila

Tässä alaluvussa

- tarkastellaan ohjelman tilan käsitettä
- perustellaan, miksi siitä on järkevää puhua logiikan ja joukko-opin avulla

Määritelmä

ohjelman tila = muuttujien arvot + suorituksen sijainti

Esimerkkejä: alkuarvot, ohjelma ja sen kaikki tilat

- aluksi $x = y = 0$

```
1:  x := 1;
2:  y := 2
3:
```

suoritus	x	y
1	0	0
2	1	0
3	1	2

- aluksi $x = 10$

```
1:  while x > 1 do
2:      x := x div 2
3:  endwhile
4:
```

suoritus	1	2	3	1	2	3	1	2	3	1	4
x	10	10	5	5	5	2	2	2	1	1	1

- aluksi $i = -5$ ja $A[1] = A[2] = A[3] = 0$

```
1: for i := 1 to 3 do
2:     A[i] := 2·i
3: endfor
4:
```

suoritus	1	2	3	1	2	3	1	2	3	1	4
i	–5	1	1	1	2	2	2	3	3	3	??
$A[1]$	0	0	2	2	2	2	2	2	2	2	2
$A[2]$	0	0	0	0	0	4	4	4	4	4	4
$A[3]$	0	0	0	0	0	0	0	0	6	6	6

Tilapredikaatit

- kaikkien muuttujien arvojen luetteleminen on usein työlästä ja tarpeetonta
- haluamme puhua tilojen ominaisuuksista määrittelemättä kaikkien muuttujien arvoja

⇒ otamme käyttöön *tilapredikaatteja*

- sanomme, että lauseke on *looginen lauseke*, jos se on tarkoitettu tulkittavaksi väittämäksi, joka pitää tai ei pidä paikkaansa
- määritelmä

  Tilapredikaatti = ohjelman muuttujien arvoista puhuva looginen lauseke.

- esimerkkejä
  - $x > 0$
  - $y = 2 \cdot x + 1$
  - $\forall\, i\ ;\ 1 \le i \le n\colon A[i] = -1 \vee A[i] \ge x$

Tilapredikaatit ja totuusarvot

- useimmissa ohjelmointikielissä on jokin tapa esittää "kyllä" ja "ei"
  - tarvitaan mm. **if**-lauseen ehdossa
- esimerkkejä
  - Pascal: tyyppi Boolean: **false** = ei, **true** = kyllä
  - C: `int` 0 = ei, muut `int` ovat kyllä
  - Lisp: Nil = ei, muut arvot = kyllä
- ohjelmointikielen lauseketta sanotaan *totuusarvoiseksi*, jos sen tulos on joko "kyllä" tai "ei"
  - C: tulos "kyllä" = 1
  - Lisp: tulos "kyllä" = "T"
- ohjelmointikielen totuusarvoinen lauseke **ei ole** tilapredikaatti
  - totuusarvoinen lauseke on ohjelma*ssa*
  - tilapredikaatti puhuu ohjelma*sta*, mutta on itse sen ulkopuolella

  ⇒ ne asuvat eri maailmoissa
- esimerkki

```
1: löytyi := false; i := 1;
2: while i ≤ n ∧ ¬ löytyi do
3:     if A[i] = avain then löytyi := true
4:     else i := i + 1
5:     endif
6: endwhile
7:
```

  - rivillä 2 "$i \le n \wedge \neg$ *löytyi*" on totuusarvoinen lauseke
  - "$i \le n \wedge$ (*löytyi* = **false**)" on tilapredikaatti, joka pitää paikkansa rivin 3 alussa, mutta ei pidä rivin 7 alussa

- totuusarvoinen lauseke on haluttaessa helppo ja luonteva *tulkita* tilapredikaatiksi
  - lauseke e tilapredikaatiksi tulkittuna tarkoittaa samaa kuin tilapredikaatti "e = kyllä"

  ⇒ kyllä ~ pitää paikkansa, ja
  ei ~ ei pidä paikkaansa
- esimerkki: tilapredikaatin "$i \le n \wedge$ (*löytyi* = **false**)" saa nyt lyhentää muotoon "$i \le n \wedge \neg$ *löytyi*"

⇒ totuusarvoisen lausekkeen ja tilapredikaatin välillä
  - on periaatteellinen ero
  - useimmiten erosta ei tarvitse välittää, vaan totuusarvoisen lausekkeen saa tulkita tilapredikaatin erikoistapaukseksi

Tilapredikaatit ja suorituksen sijainti

- tavallisesti tilapredikaatit eivät puhu suorituksen sijainnista

⇒ tilapredikaatin paikkansapitävyys riippuu siitä, missä kohti ohjelmaa se tarkastetaan

- tärkeitä tulkintakohtia
  - "alussa" = juuri ennen ohjelman suorituksen aloittamista
  - "lopussa" = ohjelman lopetettua normaalisti (ei siis esim. ohjelman kaaduttua)
  - "rivillä n" = **joka kerta** kun ohjelma on valmis suorittamaan rivin n
- lopussa tulkittava tilapredikaatti sanoo jotain ohjelman tuloksista
  - tulokset jäävät joihinkin muuttujiin

- alussa tulkittava tilapredikaatti tulkitaan eri tavalla kuin muualla esiintyvät!
  - esittää muuttujissa oleville arvoille asetettua *alkuehtoa*
  - ko. muuttujat sisältävät ko. ohjelmanpätkän syötteet
- usein on kätevää kirjoittaa tilapredikaatti siihen kohti ohjelmaa, jossa se halutaan tulkita
  - aaltosulkeisiin "{" ja "}" (myöhemmin myös sulkeisiin "⟨" ja "⟩")
  - tällöin ei välttämättä tarvita rivinumeroita
- esimerkki

```
if x < 0 then
    { x < 0 }
    x := –x
    { x > 0 }
endif { x ≥ 0 }
```

  - jos **then**-haaraa ei suoriteta, niin lopussa $x \ge 0$
  - jos suoritetaan, niin lopussa $x > 0$

  ⇒ joka tapauksessa lopussa $x \ge 0$

- esimerkki (kaikki muuttujat kokonaislukutyyppiä)

```
 1:  { n ≥ 1 }
 2:  a := 1; y := n;
 3:  { a = 1 ∧ y = n ≥ 1 }  (* seuraus: a ≤ y *)
 4:  while a < y do
 5:      { a < y }
 6:      v := (a+y) div 2;
 7:      { a ≤ v < y }
 8:      if A[v] < avain then
 9:          { v < y }
10:          a := v+1
11:          { a ≤ y }
12:      else
13:          { a ≤ v }
14:          y := v
15:          { a ≤ y }
16:      endif
17:      { a ≤ y }
18:  endwhile
19:  { a = y }
```

  - syötteitä koskeva vaatimus: $n \geq 1$
  - rivi 7 voidaan päätellä rivistä 5 ja keskiarvon ja alaspäin pyöristyksen ominaisuuksista
  - rivit 9 ja 13 seuraavat rivistä 7
  - rivi 17 seuraa riveistä 11 ja 15
  - rivillä 19 $a \geq y$ rivin 4 ehdon vuoksi; lisäksi $a \leq y$ rivien 3 ja 17 vuoksi; näistä yhdessä seuraa $a = y$

Pysyväisväittämät

- *pysyväisväittämä* eli *assertio* on ohjelman sisään tarkastuksen vuoksi kirjoitettu lause, joka kaataa ohjelman, jos annettu ehto ei päde
- pysyväisväittämistä on **erittäin** paljon hyötyä varsinkin monimutkaisten tietorakenteiden ja algoritmien virheiden jäljittämisessä
- C++:

```
#include <cassert>
   ...
assert( i >= 0 && 1 < n );
```

- AV:
  - parametrit lasketaan joka tapauksessa

  ⇒ ei tehokkain mahdollinen ratkaisu, mutta …
  - yksinkertainen ja informatiivinen
  - näin pientä tehohävikkiä kannattaa murehtia vasta kun ohjelmointitaidot ovat hyvät

```
inline void tarkasta(
   bool ehto, const char *v0,
   int i1 = 0, const char *v1 = 0,
   int i2 = 0, const char *v2 = 0
){
   if( !ehto ){
      std::cout.flush();  // kyllä, cout!
      std::cerr << "\n??? Ohjelman "
         "sisäinen virhe:\n??? " << v0;
      if( v1 ){ std::cerr << i1 << v1; }
      if( v2 ){ std::cerr << i2 << v2; }
      std::cerr << std::endl; exit( 1 );
   }
}
```


- käyttöesimerkki

```
 1:  tarkasta( n ≥ 1, “liian pieni n” )
 2:  a := 1; y := n;
 4:  while a < y do
 6:      v := (a+y) div 2;
 8:      if A[v] < avain then
10:          a := v+1
12:      else
14:          y := v
16:      endif
17:      tarkasta( a ≤ y, “liian iso a ”, a, “ ”, y, “” )
18:  endwhile
19:  tarkasta( a = y, “a ≠ y ”, a, “ ”, y, “” )
```

- jos tilapredikaatin totuusarvo on helppo laskea, siitä saa kätevästi pysyväisväittämän
- monimutkaisemmin laskettavan tilapredikaatin tarkastuksesta voi tehdä aliohjelman, jota kutsutaan silloin tällöin
  - esim. pääsilmukan joka sadannella kierroksella
  - virhettä jäljitettäessä kutsujen tiheyttä voi nostaa
  - lopullisesta ohjelmasta kutsun voi poistaa esim. muuttamalla kommentiksi tai `#ifdef`illä
- esimerkki: keko

  $\forall\, i;\ 2 \leq i \leq n\colon A[\lfloor i/2 \rfloor] \geq A[i]$
- esimerkki: kaksisuuntainen linkitetty lista

  $x\text{^}.\text{next}\text{^}.\text{prev} = x \wedge x\text{^}.\text{prev}\text{^}.\text{next} = x$

⇒ tilapredikaattien avulla voi testata ohjelmia

Jatkossa tulemme laskemaan paljon tilapredikaateilla

⇒ kannattaa kerrata predikaateilla laskemisen säännöt eli logiikan perusteet esim. 8100500 prujusta

Ohjelmanpätkien määrittelyssä tarvitaan

- merkintöjä kohdealueen käsitteille
  - esim. järjestämisalgoritmi: “<”
- joukko-oppia tietorakenteista puhumiseen
- logiikkaa määritelmän osien yhdistämiseen
  - propositiologiikka: loogiset operaattorit ja säännöt
  - predikaattilogiikka: kohdealueen muuttujat, kvanttorit

Ohjelmien todistamisessa tarvitaan

- kohdealueen laskulakeja
  - esim.

    jos $a < b$ ja $b < c$, niin $a < c$
- ohjelmointikielen semantiikan teoriaa
  - esim.

    if $x < 0$ **then** $x := -x$ **endif** $\{x \geq 0\}$
- logiikkaa päättelyjen suorittamiseen

Ohjelmointikielen lauseiden semantiikkaa käsitellään luvuissa 3.2 ja 3.3

2.2 Tilapredikaatin kirjoittaminen

Esimerkki: taulukko $A[1\ldots n]$ on järjestyksessä

- miksi seuraava ei kelpaa?

  järjestyksessä1$(A[1\ldots n])$:⇔
  $\forall\, i\ ;\ 1 \le i \le n\colon A[i] \le A[i+1]$

- korjattu versio:

  järjestyksessä$(A[1\ldots n])$:⇔
  $\forall\, i\ ;\ 1 \le i \le n-1\colon A[i] \le A[i+1]$

- minkä lisävaatimuksen seuraava asettaa?

  järjestyksessä2$(A[1\ldots n])$:⇔
  $\forall\, i\ ;\ 1 \le i \le n-1\colon A[i] < A[i+1]$

Määrittelymerkinnät :⇔ ja :=

- tarkoittavat "määritellään tarkoittamaan"
  - :⇔ predikaateille
  - := muille
- määritelmän *symboli* := *lauseke* jälkeen pätee *symboli* = *lauseke*
  - vastaavasti :⇔
- määritelmä tekee muutakin kuin on yhtälö
  - ottaa käyttöön uuden symbolin

⇒ hyvä olla oma, epäsymmetrinen merkintä

- kirjallisuudessa esiintyy mm. $=_{\text{def}}$ ja $\triangleq$
- :⇔ ja := matkivat ohjelmointikielten sijoitusta

Esimerkki: paikan valinta järjestetystä taulukosta

- oletamme, että *järjestyksessä*$(A[1\ldots n])$ ja $n \ge 1$

- haluamme predikaatin *paikka*$(A[1\ldots n], avain, x)$, joka sanoo, että
  - jos *avain* on taulukossa, x osoittaa missä kohti
  - muutoin x osoittaa sitä kohtaa, missä alkion *avain* "pitäisi olla"
- x osoittaa taulukon kohtaa
  ⇒ järkevät x:n arvot ovat välillä $1 \le x \le n$
  ⇒ ei haittaa, vaikka predikaatti olisi muulloin määrittelemätön
- mitä "pitäisi olla" tarkoittaa?
  - esim. *avain* = 7

3	5	6	9	11	12	18	18	22	23
1	2	3	4	5	6	7	8	9	10

    ↑ ↑
    x? x?

  - valinta: "pitäisi olla" ~ "lähinnä suurempi"
- 1. yritys: *paikka1*$(A[1\ldots n], avain, x)$:⇔
  $A[x] = avain \vee (A[x-1] < avain \wedge A[x] > avain)$
  - määrittelemätön, kun $x = 1$ ja $A[1] > avain$
- *paikka2*$(A[1\ldots n], avain, x)$:⇔
  $A[x] = avain$
  $\vee\ (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  - entä jos *avain* = 24?

  ⇒ arvo $x = n+1$ näyttää tarpeelliselta
- johtuuko arvon $x = n+1$ tarve itse tehtävästä vai ratkaisuyrityksen huonoudesta?
- koe: jos $n = 1$, niin
  - tarvitaan 2 vastausvaihtoehtoa "tähän" ja "perään" sen mukaan onko $A[1] \ge avain$
  - arvoalue $1 \le x \le n = 1$ sallii vain yhden
  - havainto yleistyy jokaiselle taulukon koolle

⇒ itse tehtävä vaatii, että arvoalueella on $n+1$ arvoa
  - valitsemme arvoalueen $1 \le x \le n+1$

- *paikka3*$(A[1\ldots n], avain, x)$:⇔
  $A[x] = avain$
  $\vee\ (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee\ (x = n+1 \wedge A[n] < avain)$
  - jos on useita i siten, että $A[i] = avain$, ei määrittele, mikä valitaan
  - pitäisi valita pienin
- *paikka4*$(A[1\ldots n], avain, x)$:⇔
  $(A[x] = avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee\ (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee\ (x = n+1 \wedge A[n] < avain)$
  ⇔
  $(A[x] \ge avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee\ (x = n+1 \wedge A[n] < avain)$
- koska $n \ge 1$, niin *paikka4* ⇔
  $(A[x] \ge avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee\ (x = n+1 \wedge A[x-1] < avain)$
  ⇔
  $(x = n+1 \vee A[x] \ge avain) \wedge (x = 1 \vee A[x-1] < avain)$
  ⇔: *paikka5*$(A[1\ldots n], avain, x)$
  - nyt on jo aika selkeä!
  - nimen vaihto oli tarpeen, koska sievennys perustui oletukseen $n \ge 1$
- osilla on selvä tulkinta:
  - $x = n+1 \vee A[x] \ge avain$ ⇔ x on tarpeeksi suuri
  - $x = 1 \vee A[x-1] < avain$ ⇔ x ei ole liian suuri
- valitsemme: *paikka*$(A[1\ldots n], avain, x)$:⇔
  paikka5$(A[1\ldots n], avain, x)$ ⇔
  $(x = n+1 \vee A[x] \ge avain) \wedge (x = 1 \vee A[x-1] < avain)$

Havaintoja

- predikaattia ei aina ole tarkoitettu käytettäväksi kaikilla argumenttien arvoyhdistelmillä
  - muiden arvoyhdistelmien huomioon otto tekee predikaatista usein turhan monimutkaisen

⇒ kannattaa tehdä päätös, mille alueelle se tarkoitetaan, ja unohtaa muut arvot
  - esimerkissä $n > 0$ ja $1 \le x \le n+1$

- jos käyttöalue on K, predikaatit P_1 ja P_2 ovat "samanveroiset", joss
  $K \Rightarrow P_1 \leftrightarrow P_2$
- jos haluaa kaikkialla määritellyn tilapredikaatin, käyttöalueen ulkopuolisten arvojen tuottaman totuusarvon voi kiinnittää jälkeen päin
  - esim. "**false**" kirjoittamalla $1 \le x \le n+1 > 1 \wedge P(x)$
  - haluttaessa rajauksen voi lopuksi sieventää $P(x)$:n "sisälle"

⇒ $P(x)$:n suunnittelussa ei kannata ottaa taakkaa käyttöalueen ulkopuolisista arvoista
  - käyttöalue muistettava kertoa, jos ≠ **true** !

- yritys kirjoittaa tilapredikaatti nostaa usein esiin kysymyksiä, joita ei aikaisemmin ole huomattu
  - esim. mitä "pitäisi olla" tarkkaan ottaen tarkoittaa?
  - esim. mikä x valitaan, jos useita vaihtoehtoja?
  - täsmällisyyden etu ja haitta!
- predikaatin käyttöalueelle osuvien määrittelemättömien tilanteiden poisto saattaa vaatia vaivaa ja huolellisuutta

- formalisointi saattaa paljastaa katteettomia ennakko-oletuksia
  - esim. “arvoalue $1 \le x \le n$ riittää”
  - tällaiset oletukset ovat kiusallinen virhelähde!
- formalisointi saattaa paljastaa monikäsitteisyyttä
  - monikäsitteisyys ei aina ole haitaksi!
  - silti on hyvä tietää, onko monikäsitteisyyttä

$\Rightarrow$ ylläolevat selittävät sitä käytännön havaintoa, että formaali määrittely parantaa merkittävästi lopputuotteen laatua ja lyhentää kehitysaikaa
  - tosin hinta on korkea!

- tulosta kannattaa yrittää sieventää tai muuntaa erilaisiin muotoihin
  - yksinkertaisempi predikaatti on usein parempi predikaatti
  - sievennetyt muodot saattavat tukea intuitiota alkuperäistä paremmin
  - sievennetyt muodot saattavat olla tulkittavissa toisenlaisesta intuitiosta

  $\Rightarrow$ usko predikaatin “oikein” oloon vahvistuu
- sieventämisen luotettavuus ei edellytä intuitiota
  - lakeja voi (ja kannattaa) soveltaa täysin mekaanisesti
  - olennaista, jotta lopputuloksen intuitiivisuus todella lisäisi uskoa predikaatin “oikein” oloon
- hyvin suunniteltu predikaatti on joskus kelvollinen alkuperäisen alueen ulkopuolellakin
  - esim. *paikka* “toimii” myös kun $n = 0$

Predikaatin kirjoittaminen muistuttaa monessa suhteessa ohjelmointia

- intuitiivisen idean koodausta kankealle kielelle
  - joskus ajatus joudutaan esittämään hyvin epäsuorasti
- ei yleensä onnistu ensimmäisellä yrityksellä
- erikoistapauksia on mietittävä
- on otettava huomioon semanttisia rajoituksia tyyliin “nollalla ei saa jakaa”
- “syötteiden” sallittu alue sovittava
- kannattaa pyrkiä “kauniiseen” tulokseen

Erojakin on

- predikaateilla voi laskea, ohjelmilla normaalisti ei

  $\Rightarrow$ paljon enemmän sieventämismahdollisuuksia
- tehokkuusnäkökohdista ei tarvitse välittää

$\Rightarrow$ paremmat mahdollisuudet saavuttaa ymmärrettävä lopputulos
  - ymmärrettävyydestä ei tarvitse tinkiä muiden asioiden hyväksi
  - tietenkään väittämän sisältöä ei saa muuttaa ymmärrettävyyden vuoksi!
- keinokokoelma on erilainen
  - esim. laskujärjestystä ei voi ohjata
  - esim. ohjelmoinnissa ei ole niin voimakkaita operaattoreita kuin “$\exists$”
- predikaatteja ei voi koeajaa!

- (on olemassa eräs lukuteorian kaava $\varphi(x)$ siten, että **ei voi** olla olemassa algoritmia, joka ottaa syötteekseen x ja vastaa aina ja vieläpä oikein kysymykseen, päteekö $\varphi(x)$)

Iso kysymys:

Mistä voi olla varma, että tilapredikaatti on oikein?

- viime kädessä vastaus on: **ei mistään!**
- voidaan asettaa joitakin muodollisia, yleispäteviä oikeellisuuden kriteereitä
  - tilapredikaatin arvo ei saa riippua $\perp$:n ominaisuuksista, jos argumentit ovat sallitulla alueella
  - on epäilyttävää, jos tilapredikaatti tuottaa aina **false** tai aina **true**, kun argumentit ovat laillisia
  - vrt. ohjelmat
- tällaiset eivät kuitenkaan kerro, onko tilapredikaatin *merkitys* se, mitä haluttiin
- tilapredikaatin merkityksen todistaminen oikeaksi vaatisi, että käytettävissä olisi formaalissa muodossa tieto siitä, mikä on “oikea merkitys”

$\Rightarrow$ tarvittaisiin toinen formaali kaava
  - mistä voi tietää, että se on oikein?

$\Rightarrow$ ketjun viimeistä formaalia kaavaa φ ei voi todistaa oikeaksi, koska sen oikeellisuuden mitta ei ole käytettävissä formaalissa muodossa
  - jos olisi, φ ei olisikaan ketjun viimeinen

$\Rightarrow$ usko ketjun viimeisen kaavan “oikeellisuuteen” voi perustua vain intuitioon

- ohjelman spesifikaatiossa olevan tilapredikaatin tehtävä on olla osa ohjelman oikeaksi todistamisessa käytettävää oikeellisuuden mittaa

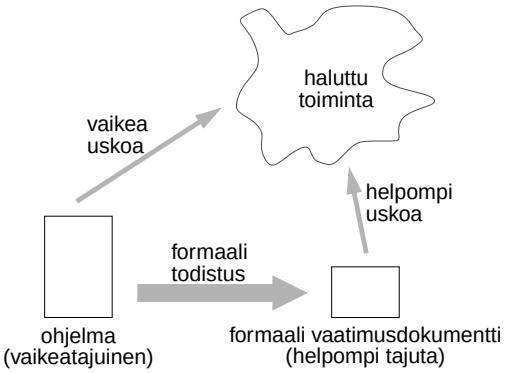
$\Rightarrow$ se on yleensä tarkoitettu ketjun viimeiseksi kaavaksi

$\Rightarrow$ sen oikeellisuutta ei voi todistaa
  - ts. ei voida todistaa sitä, että sen *merkitys* on haluttu
  - kokonaan eri asia on, että usein voidaan osoittaa, että se *on voimassa* jossain tilassa

Seurauksia

- kysymys: Jos ei voida olla varmoja, että ohjelman todistuksessa käytettävä tilapredikaatti on oikein, miten sitten voidaan olla varmoja, että ohjelma on oikein?
- vastaus: ei mitenkään!
  - ei voida todistaa, että ohjelma toimii niin kuin halutaan
  - voidaan todistaa, että ohjelma toimii esim. tilapredikaateista muodostetun formaalin vaatimusmäärittelyn mukaan
  - onko vaatimusmäärittely oikein, jää uskon varaan
- kysymys: Mitä hyötyä ohjelmien todistamisesta sitten on?

- vastaus: Sillä voidaan olennaisesti “pienentää” sitä askelta, joka on vain uskottava.



- seurauksia
  - tilapredikaatin tulee olla mahdollisimman helppotajuinen (tietenkin sillä rajoituksella, että sen tulee silti sanoa oikea asia)
  - tilapredikaatteja kannattaa “testata”

Keinoja lisätä uskoa tilapredikaatin oikeellisuuteen

- alla olevissa esimerkeissä käytetään predikaattia $paikka(A[1\ldots n], avain, x) \Leftrightarrow$
  $(x = n{+}1 \vee A[x] \geq avain) \wedge (x = 1 \vee A[x{-}1] < avain)$

- tarkasta, että predikaatin arvo ei voi riippua $\perp$:n ominaisuuksista, jos argumentit ovat sallitulla alueella
  - esim. *paikka*:n ainoat vaaralliset operaatiot ovat $A[x]$ kun $x = n{+}1$ ja $A[x{-}1]$ kun $x = 1$
  - tällöin ympäristö määrää tuloksen
  - (mitä jos $n = 0$?)
- testaa sijoittamalla osalle argumenteista arvot, sieventämällä kaavaa ja varmistamalla, että lopputulos vastaa intuitiota
  - erityisesti kannattaa testata rajatapauksilla
  - esim. kun $n = 0$, niin $paikka \Leftrightarrow (x = 1) \vee$ **?** $\cdot/\cdot$
  - esim. kun $n = 1$, niin $1 \leq x \leq 2$, joten $paikka$
    $\Leftrightarrow (x = 2 \vee A[x] \geq avain) \wedge (x = 1 \vee A[x{-}1] < avain)$
    $\Leftrightarrow (x = 2 \vee A[1] \geq avain) \wedge (x = 1 \vee A[1] < avain)$
    $\Leftrightarrow (A[1] < avain \rightarrow x = 2) \wedge (A[1] \geq avain \rightarrow x = 1)$
    täsmää, koska luettelee oikeat vastaukset testin “$A[1] < avain$” tuloksen funktiona
- voit myös testata sijoittamalla kaikille argumenteille arvot, ja tarkastamalla lopputulos
  - kannattaa testata sekä **false**- että **true**-tapauksilla
  - usein työläyteensä nähden tehoton testaustapa
  - $\Rightarrow$ yleensä on parempi testata sieventämällä
- sievennä kaava toiseen muotoon, ja tarkasta, että tulos vastaa intuitiota
- esimerkki: $paikka(A[1\ldots n], avain, x) \Leftrightarrow$
  $x = 1 \wedge x = n{+}1$
  $\vee \quad (x = n{+}1 \wedge A[n] < avain)$
  $\vee \quad (x = 1 \wedge A[1] \geq avain)$
  $\vee \quad (A[x] \geq avain \wedge A[x{-}1] < avain)$
  - $x = 1 = n{+}1$ käsittelee oikein tapauksen $n = 0$
  - $x = n{+}1 \wedge A[n] < avain$ on oikea tapa käsitellä taulukon oikea reuna, jos $n > 0$

  - $x = 1 \wedge A[1] \geq avain$ on oikea tapa käsitellä vasen reuna, jos $n > 0$
  - $A[x] \geq avain \wedge A[x{-}1] < avain$ on oikein keskellä taulukkoa (määrittelemätön reunoissa, mutta silloin jokin kumpi edeltävistä tuottaa **true**)

Kannattaa käyttää hyväksi sitä, että tilapredikaateilla voi laskea.

Esimerkki: taulukon $A[1\ldots n]$ kaikki alkiot ovat eri suuria

- 1. yritys: $erisuuria1(A[1\ldots n]) :\Leftrightarrow$
  $\forall\, i, j \in \{1,\ldots,n\}: A[i] \neq A[j]$
- testataan rajatapauksella $n = 1$
  - silloin $erisuuria1(A[1\ldots n]) \Leftrightarrow A[1] \neq A[1]$
  hupsis!
- $erisuuria2(A[1\ldots n]) :\Leftrightarrow$
  $\forall\, i, j \in \{1,\ldots,n\}: (i \neq j \rightarrow A[i] \neq A[j])$
  - toimii rajatapauksissa $n = 0$ ja $n = 1$
  - ei indeksoi A:ta laittomasti
- intuition mukaan tulisi päteä
  $järjestyksessä2(A[1\ldots n]) \Leftrightarrow$
  $järjestyksessä(A[1\ldots n]) \wedge erisuuria2(A[1\ldots n])$
  tarkastetaanpa!
  - “$\Leftarrow$”: helppo
  - “$järj.2(A[1\ldots n]) \Rightarrow järj.(A[1\ldots n])$”: välitön
  - “$järj.2(A[1\ldots n]) \Rightarrow eris.2(A[1\ldots n])$”: kun $i < j$, niin $1 \leq i < n$, joten $A[i] < A[i{+}1] < \ldots < A[j]$, joten $A[i] \neq A[j]$; vastaavasti, kun $j < i$; tapaus $i = j$ välitön

$\Rightarrow$ uskomme, että *erisuuria2* on oikein
$erisuuria(A[1\ldots n]) :\Leftrightarrow erisuuria2(A[1\ldots n])$

Esimerkki tilapredikaatin kirjoittamisesta joukkojen avulla: samat alkiot

- $samat\text{-}alk(A[1\ldots n], B[1\ldots n])$ ~ taulukoissa A ja B on samat alkiot
- 1. yritys: $samat\text{-}alk1(A[1\ldots n], B[1\ldots n]) :\Leftrightarrow$
  $\forall\, i\,;\, 1 \leq i \leq n: \exists\, j\,;\, 1 \leq j \leq n: A[i] = B[j]$
  - sallii $A = [1,1]$, $B = [1,2]$
- $samat\text{-}alk2(A, B) :\Leftrightarrow$
  $samat\text{-}alk1(A, B) \wedge samat\text{-}alk1(B, A)$
  - sallii $A = [1,1,2]$, $B = [1,2,2]$

$\Rightarrow$ näyttää tarpeelliselta laskea, kuinka monta kutakin alkiota on

$\Rightarrow$ otamme käyttöön apumerkinnän:
$määrä(x, A[1\ldots n]) := |\{\, i \mid 1 \leq i \leq n \wedge A[i] = x \,\}|$

- (“:=” tarkoittaa tässä “määritellään tarkoittamaan”)
- $samat\text{-}alk3(A, B) :\Leftrightarrow$
  $\forall\, i\,;\, 1 \leq i \leq n: määrä(A[i], A) = määrä(A[i], B)$
- *samat-alk3* on epäsymmetrinen A:n ja B:n suhteen
  $\Rightarrow$ voiko se olla oikein?
  - vastaoletus: symmetrinen kaava ei päde
  $\Rightarrow \exists\, k\,;\, 1 \leq k \leq n: määrä(B[k], A) \neq määrä(B[k], B)$
  - jos $määrä(B[k], A) > 0$, niin $\exists\, i\,;\, 1 \leq i \leq n: B[k] = A[i]$, joten $samat\text{-}alk3 \Rightarrow määrä(B[k], A) = määrä(B[k], B)$
  $\Rightarrow määrä(B[k], A) = 0 \Rightarrow määrä(B[k], B) > 0$
  $\Rightarrow$ B:ssä on enemmän alkioita kuin A:ssa
  $\Rightarrow$ ristiriita
  $\Rightarrow$ vastaoletus on väärin
  $\Rightarrow$ epäsymmetria on vain näennäistä

$\Rightarrow$ hyväksymme $samat\text{-}alk(A, B) :\Leftrightarrow samat\text{-}alk3(A, B)$

Tilapredikaatin muotoileminen on joskus vaikeaa

- logiikan kieli on kankea
  - matemaatikkojen suunnittelema
  - ei suunniteltu isojen väittämien esittämiseen
  - ei suunniteltu tietokoneen luettavaksi
- toisaalta logiikka sallii omien merkintöjen lisäilyn
  ⇒ saa luvan riittää meille
  - omia merkintätapoja saa tällä kurssilla ottaa tarvittaessa käyttöön, mutta ne on määriteltävä (paitsi jos merkitys on ilmeinen)
  - esim. nyt kun *järjestyksessä* on määritelty, sitä voidaan käyttää muiden predikaattien osana
- uusia tietoabstraktioita käytettäessä on usein tarpeen ottaa käyttöön ko. abstraktion käsittelyyn sopivia merkintöjä
  - esim. jono, pino, graafi
- tilapredikaatin kirjoitusvaikeuksien syynä on usein se, että väittämän tarkkaa sisältöä kaikissa tilanteissa ei ole aikaisemmin mietitty
  - siis predikaatin vaikeuden syynä on usein se, että itse **asia** on vaikea
  - ilman väittämän formalisointia sen aukot jäävät yleensä huomaamatta
    ⇒ kiusallisia virheitä
- formalismin käytön ensimmäinen ja usein suurin hyöty tulee siitä, että formalismi pakottaa miettimään asian tarkasti

2.3 Ohjelman spesifikaatio

Ohjelman oikeellisuus

- on osoittautunut hyödylliseksi jakaa ohjelman oikeellisuusvaatimus kahteen osaan:
  - *osittainen oikeellisuus* (*partial correctness*)
  - *pysähtyvyys*
- osittainen oikeellisuus =
  ohjelma ei saa tehdä mitään väärää
  - ei saa antaa väärää vastausta
  - ei saa pysähdyttyään jäädä väärään tilaan
  - ei saa tehdä mitään kiellettyä (nollalla jako, taulukon indeksointi ohi rajojen tms.)
- pysähtyvyys = ohjelman on lopulta pysähdyttävä
- seuraava ohjelma on osittain oikea, olipa spesifikaatio mikä tahansa:

```
while true do
    (* älä vain tee mitään! *)
endwhile
```

- *täysi oikeellisuus* (*total correctness*) =
  osittainen oikeellisuus + pysähtyvyys
- pysähtyvyyden vaatimus on sama kaikille ohjelmille
- osittaisen oikeellisuuden vaatimus on tapauskohtainen
  - esim. järjestämisohjelman lopussa taulukko sisältää alkuperäiset alkiot suuruusjärjestyksessä
  - esim. etsimisohjelman lopussa i osoittaa kysyttyä tietuetta

⇒ tarvitsemme esitystavan, jolla ohjelmalle asetettava osittaisen oikeellisuuden vaatimus voidaan ilmaista

Osittaisen oikeellisuuden spesifikaatio

- ohjelman osittaisen oikeellisuuden vaatimus voidaan ilmoittaa muodossa

  {*alkuehto*} *ohjelma* {*loppuehto*}

  missä *alkuehto* (*precondition*) ja *loppuehto* (*postcondition*) ovat tilapredikaatteja
- esimerkki: potenssiin korotus

  { $n \geq 0$ }
  x := 1;
  for i := 1 **to** n **do**
  $x := x \cdot a$
  endfor
  { $x = a^n$ }
- esimerkki: alustus

  { **true** }
  for i := 1 **to** n **do**
  $A[i] := 0$
  endfor
  { $\forall\, i$; $1 \leq i \leq n$: $A[i] = 0$ }
- tarvittaessa otetaan käyttöön apusymboleita edustamaan arvoja, jotka eivät muutu ohjelman suorituksen aikana
  - eivät edusta ohjelman muuttujia, vaan esim. niiden alkuarvoja
  - esitetään usein alaindeksillä 0, esim. x_0, $A_0[i]$
  - ns. *haamumuuttujat* (*ghost variables*)
- esim. muuttujien arvojen vaihto

  { $x = x_0 \wedge y = y_0$ }
  $t := x$; $x := y$; $y := t$
  { $x = y_0 \wedge y = x_0$ }

- siis: tällä kurssilla merkintä

  {*alkuehto*} *ohjelma* {*loppuehto*}

  tarkoittaa seuraavaa:

  jos *ohjelma*:a käynnistettäessä *alkuehto* pätee,
  niin *ohjelma*:n pysähdyttyä *loppuehto* pätee
- **varoitus!** merkitys vaihtelee hieman eri kirjoittajilla
  - meidän valintamme on sama kuin Backhousen
  - esim. Gries esittää saman muodossa
    alkuehto {*ohjelma*} *loppuehto*

Minkä muuttujien arvoja saa muuttaa?

- seuraava ohjelma selvästikin täyttää muodollisen spesifikaationsa!

  { $n \geq 0$ }
  x := 1; a := 1; n := 1
  { $x = a^n$ }
- muuttujien arvojen muuttaminen voidaan kieltää vaatimalla, että niiden arvot lopussa ovat samat kuin alussa:

  { $n = n_0 \geq 0 \wedge a = a_0$ }
  …
  { $x = a^n \wedge n = n_0 \wedge a = a_0$ }
- työlästä ongelman yleisyyteen nähden

⇒ ratkaisu: jaetaan muuttujat kahteen ryhmään sen mukaan, saako ohjelma muuttaa niiden arvoja
  - *kiinteät* muuttujat
  - tavalliset muuttujat

- miten suhtautua seuraavaan?

  $\{ n \geq 0 \}$
  $apu := n$; $n := 0$;
  … (* tässä ei kosketa *apu*:un *)
  $n := apu$
  $\{ x = a^n \}$

- ratkaisu
  - on vaikea (joskus mahdoton) tarkastaa, muuttaako ohjelma todella jonkin muuttujan arvoa
  - on helppo tarkastaa, onko ohjelmassa k.o. muuttujaan kohdistuvia sijoituslauseita

  ⇒ **päätös:** kiinteään muuttujaan ei saa edes yrittää sijoittaa
  - (useiden saantipolkujen ongelma)
- kiinteät muuttujat ovat eri asia kuin haamumuuttujat
  - haamumuuttujat eivät esiinny ohjelmassa, ainoastaan tilapredikaateissa
- syöte-, apu- ja tulosmuuttujat
  - syötteet annetaan syötemuuttujissa
  - tulokset palautetaan tulosmuuttujissa
  - apumuuttujat ovat välituloksia varten
- jako kiinteisiin ja tavallisiin muuttujiin on eri asia kuin jako syöte-, apu- ja tulosmuuttujiin
  - kiinteät muuttujat ovat yleensä syötemuuttujia
  - kaikki syötemuuttujat eivät ole kiinteitä: esim. taulukon järjestäminen paikallaan
- yleensä jako tavallisiin ja kiinteisiin muuttujiin oletetaan tunnetuksi ja jätetään mainitsematta

  ⇒ ryhmille ei edes ole vakiintuneita nimiä

Osittainen oikeellisuus ja loppuun pääsy

- seuraava ohjelma on osittain oikea, olkoot P ja Q mitä tahansa:

  $\{P\}$
  while true do
  endwhile
  $\{Q\}$

⇒ spesifikaatio ei ole täydellinen, ellei vaadita, että ohjelma pääsee loppuun

- yleensä ohjelmien halutaan pääsevän loppuun
  - tärkeä poikkeus: reaktiiviset ohjelmat (ei käsitellä tällä kurssilla)
- ohjelman loppuun pääsemisen todistaminen vaatii usein huomattavaa lisävaivaa
  - esim.

  $x := 1$; $y := 1$; $z := 1$; $n := 3$;
  while $x^n + y^n \neq z^n$ **do**
  if $y > 1$ **then** $x := x+1$; $y := y-1$
  elsif $z > 1$ **then** $z := z-1$; $y := x+1$; $x := 1$
  elsif $n > 3$ **then** $n := n-1$; $z := x+1$; $x := 1$
  else $n := x+3$; $x := 1$
  endif
  endwhile
  $\{ x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3 \wedge x^n + y^n = z^n \}$

⇒ osittainen oikeellisuus ja loppuun pääsy todistetaan yleensä erikseen

⇒ käsittelemme paljon välivaiheita, joissa loppuun pääsyä ei vaadita

⇒ tällä kurssilla on oltava tarkkana siitä, milloin loppuun pääsy vaaditaan ja milloin ei

Loppuun pääsyn spesifiointi

- ohjelman loppuun pääsy vaaditaan (tällä kurssilla) asettamalla tilapredikaatit kulmasulkeisiin:

  $\langle x = x_0 \wedge y = y_0 \rangle$
  $t := x$; $x := y$; $y := t$
  $\langle x = y_0 \wedge y = x_0 \rangle$

- siis: tällä kurssilla merkintä

  $\langle alkuehto \rangle$ *ohjelma* $\langle loppuehto \rangle$

  tarkoittaa seuraavaa:

  jos *ohjelma*:a käynnistettäessä *alkuehto* pätee,
  niin *ohjelma* pääsee loppuun, ja siellä *loppuehto* pätee
- **varoitus!** tämäkään merkintä ei ole vakiintunut
  - joillakin kirjoittajilla $\{P\}$ *ohjelma* $\{Q\}$ sisältää loppuun pääsyn vaatimuksen (esim. Gries)
  - meidän tapamme: N. Francez: *Program Verification*

⇒ pelkkä loppuun pääsemisen vaatimus voidaa esittää seuraavasti:

$\langle alkuehto \rangle$ *ohjelma* $\langle$**true**$\rangle$

Yleisoletuksia

- jotta esimerkeissä, tenttivastauksissa yms. ei tarvitsisi luetella suurta joukkoa yleismääritelmiä, oletamme alla olevat, ellei toisin sanota
- muuttujat saavat arvonsa joukosta $\boldsymbol{Z}$
- merkintä $A[a…y]$ tarkoittaa taulukkoa, jonka indeksialue on a, $a+1$, …, y
  - a ja y ovat kiinteitä
  - $y \geq a-1$
  - moniulotteiset taulukot esim. $A[1…n, 1…m]$

- kun taulukon rajat on kerran annettu, samaa taulukon nimeä saa käyttää ilman rajoja
- haamumuuttujan tyyppi, koko jne. määräytyvät vastaavan ohjelmamuuttujan vastaavista
  - esim. jos $A[1…n]$, niin $A = A_0$ tarkoittaa, että A_0:n indeksialue on 1, …, n, ja $\forall\ i\ ;\ 1 \leq i \leq n$: $A_0[i] = A[i]$

Spesifiointiesimerkki: taulukon $A[1…n]$ järjestäminen

- 1. yritys:

  $\langle$ **true** $\rangle$
  järjestä
  $\langle$ *järjestyksessä*(A) $\rangle$

- pulma: sallii

  $\langle$ **true** $\rangle$
  for $i := 1$ **to** n **do** $A[i] := 0$ **endfor**
  $\langle$ *järjestyksessä*(A) $\rangle$

- ratkaisu:

  $\langle A = A_0 \rangle$
  järjestä
  $\langle$ *järjestyksessä*$(A) \wedge$ *samat-alk*(A, A_0) $\rangle$

Kuinka vakava alkuperäisen spesifikaation puute on?

- usein järjestämisalgoritmin ainoa taulukkoa muuttava operaatio on kahden alkion vaihto

⇒ *samat-alk*(A, A_0) on helppo nähdä voimassa olevaksi

⇒ todistuksen pääpaino keskittyy väittämän *järjestyksessä*(A) todistamiseen

- joskus *samat-alk*(A, A_0) ei kuitenkaan ole ilmeinen
  - alla olevassa esimerkissä *samat-alk*(B, A_0)

    COUNTING-SORT(A[1…n], B[1…n], M)

    { ∀ i ; 1 ≤ i ≤ n: 0 ≤ A[i] ≤ M }
    for k := 0 **to** M **do** C[k] := 0 **endfor**
    for i := 1 **to** n **do**
    C[A[i].*avain*] := C[A[i].*avain*] + 1
    endfor
    (* nyt C[k] tietää kuinka monen alkion avain = k *)
    for k := 1 **to** M **do** C[k] := C[k] + C[k–1] **endfor**
    (* nyt C[k] tietää kuinka monen alkion avain ≤ k *)
    for i := n **downto** 1 **do**
    B[C[A[i].*avain*]] := A[i];
    C[A[i].*avain*] := C[A[i].*avain*] – 1
    endfor

⇒ virheen vakavuus riippuu spesifikaation käyttötarkoituksesta
  - jos se annetaan ilkeämieliselle lukijalle (amerikkalainen lakimies), sen on oltava oikein
  - kun sitä käytetään tilanteessa, jossa puuttuva osa on kaikkien mielestä silmin nähden totta, virhe ei ole vakava
- vrt. spesifikaatio
  “saat jälkiruokaa sitten kun lautanen on tyhjä”
- on parempi esittää vaatimus *samat-alk*(A, A_0) vaikka epäformaalisti kuin jättää se kokonaan pois

Esimerkki: puolitushaku spesifikaatioineen

- A[1…n] ja *avain* kiinteitä

  ⟨ ∀ i ; 1 ≤ i ≤ n–1: A[i] ≤ A[i+1] ⟩
  a := 1; y := n+1;
  while a < y **do**
  v := (a+y) div 2;
  if A[v] < *avain* **then** a := v+1
  else y := v
  endif
  endwhile
  ⟨ (a = n+1 ∨ A[a] ≥ *avain*) ∧
  (a = 1 ∨ A[a–1] < *avain*) ⟩

Esimerkki: binääripotenssi spesifikaatioineen

- a ja n kiinteitä
- x, a ja b jotain lukutyyppiä, esim. $\boldsymbol{R}$

  ⟨ n ≥ 0 ⟩
  i := n; b := a; x := 1;
  while i > 0 **do**
  if i mod 2 = 1 **then** x := b·x **endif**;
  i := i div 2; b := b·b
  endwhile
  ⟨ x = a^n ⟩

Esimerkki: suurimman 0-neliön etsiminen matriisista, jonka alkiot ovat 0:ia ja 1:iä

- aputilapredikaatti:
  nollaneliö(i, j, k, A[1…n, 1…m]) :⇔
  i+k–1 ≤ n ∧ j+k–1 ≤ m ∧
  ∀ h ; i ≤ h ≤ i+k–1: ∀ l ; j ≤ l ≤ j+k–1: A[h, l] = 0
  - käyttöalue: 1 ≤ i ≤ n, 1 ≤ j ≤ m, k ≥ 0
  - miksi tarvitaan ehto i+k–1 ≤ n ∧ j+k–1 ≤ m ?
- A kiinteä
- m ja n ovat kiinteitä ilman eri mainintaa yleisoletuksen vuoksi
- vastauksen
  - se nurkka, jonka koordinaatit ovat pienimmät on kohdassa (i, j)
  - koko on k

  ⟨ ∀ i ; 1 ≤ i ≤ n: ∀ j ; 1 ≤ j ≤ m: A[i, j] ∈ {0,1} ⟩
  etsi_isoin_nollaneliö
  ⟨ 1 ≤ i ≤ n ∧ 1 ≤ j ≤ m ∧ k ≥ 0 ∧
  nollaneliö(i, j, k, A[1…n, 1…m]) ∧
  ∀ i ; 1 ≤ i ≤ n: ∀ j ; 1 ≤ j ≤ m:
  ∀ h ; h > k: ¬*nollaneliö*(i, j, h, A[1…n, 1…m]))
  ⟩

Esimerkki: opintojakson välittömien ja välillisten esitietojen läpikäynti

- kaikkien opintojaksojen joukko on *Kurssit*
- opintojakson k välittömät esitiedot ovat joukko
  välitt-esit(k)
- otamme käyttöön merkintöjä (k, g ∈ *Kurssit*)
  - $k \to g$:⇔ g ∈ *välitt-esit*(k)
  - $k \to^+ g$:⇔ ∃ n ∈ $\boldsymbol{N}$: n > 0 ∧ ∃ k_0, k_1, …, k_n:
    $k = k_0 \wedge k_n = g \wedge \forall\ i\ ;\ 1 \le i \le n\colon k_{i-1} \to k_i$
- algoritmi spesifikaatioineen
  - *Kurssit*, *välitt-esit* ja k_0 kiinteitä
  - muuttujien k, g ja k_0 tyyppi on *Kurssit*
  - *kesken*, *löydetyt* ja *välitt-esit* ovat tyyppiä $2^{Kurssit}$, siis joukko kursseja

  kesken := {k_0}; *löydetyt* := Ø
  while *kesken* ≠ Ø **do**
  valitse-jokin k ∈ *kesken*
  forall g ∈ *välitt-esit*(k) **do**
  if g ∉ *löydetyt* **then**
  löydetyt := *löydetyt* ∪ {g}
  if $g \neq k_0$ **then** *kesken* := *kesken*∪{g} **endif**
  endif
  endfor
  kesken := *kesken* – {k}
  endwhile
  ⟨ *löydetyt* = { k ∈ *Kurssit* | $k_0 \to^+ k$ } ⟩
- johtopäätös: monimutkaisten tai abstraktien rakenteiden parissa puuhattaessa on usein tarpeen ottaa käyttöön alakohtaisia merkintöjä
- (vastaavasti todistuksissa saatetaan joutua hyödyntämään alakohtaisia tuloksia)

3 OHJELMIEN OIKEAKSI OSOITTAMINEN

Tässä luvussa tarkastellaan keinoja osoittaa, että pienehkö ohjelmanpätkä täyttää spesifikaationsa

⇒ joudutaan puhumaan ohjelmointikielen lauseiden semantiikasta

Keinoista on hyötyä ennen kaikkea pienehkön, toimintaperiaatteeltaan hankalan ohjelmanpätkän suunnitteluun ja tarkistamiseen

- etsintä- ja selaustehtävät
- järjestäminen
- koodimuunnokset
- aritmeettiset tehtävät
- hankalan tietorakenteen käsittely
- merkkijonojen käsittely
- …

Saman kaltaisiin ideoihin perustuu myös

- isompien ohjelmien
- rinnakkaisten ohjelmien
- abstraktien spesifikaatioiden
- monimutkaisten algoritmien

oikeaksi osoittaminen

3.1 Yleisperiaate

Ehdotus todistuskeinoksi: symbolinen suoritus

- käydään ohjelma läpi lause kerrallaan
- esitetään ohjelman tila joka välissä tilapredikaatilla

Esimerkki: muuttujien arvojen vaihto

- osoitettava, että seuraava ohjelmanpätkä vaihtaa muuttujien x ja y arvot

  $t := x$; $x := y$; $y := t$

- suoraviivainen ratkaisu
  - annetaan x:n ja y:n alkuarvoille nimet x_0 ja y_0
  - suoritetaan ohjelma symbolisesti pitäen kirjaa muuttujien arvoista

    $\langle\ x = x_0 \wedge y = y_0\ \rangle$
    $t := x;$
    $\langle\ x = x_0 \wedge y = y_0 \wedge t = x_0\ \rangle$
    $x := y;$
    $\langle\ x = y_0 \wedge y = y_0 \wedge t = x_0\ \rangle$
    $y := t$
    $\langle\ x = y_0 \wedge y = x_0 \wedge t = x_0\ \rangle$

- koska lopussa $x = y_0$ ja $y = x_0$, ohjelma vaihtaa x:n ja y:n arvot

⇒ ohjelman pystyi osoittamaan oikeaksi symbolisella suorituksella

Hankalampi esimerkki: taulukosta etsiminen

- tehtävän asettelu
  - on annettu taulukko $A[1\ldots n]$ ($n \geq 1$) ja alkio x
  - löydettävä pienin i, jolle $A[i] = x$
  - jos sellaista ei ole, on palautettava $i = 0$
  - A ja x kiinteitä
- (täyden) oikeellisuuden vaatimus

  $\langle\ n \geq 1\ \rangle$
  etsintä
  $\langle\ 1 \leq i \leq n \wedge A[i] = x \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\vee\ \ i = 0 \wedge \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x\ \rangle$

- oikeaksi osoitettava ohjelma

  (* *etsintä* *)
  $i := 1$;
  while $i < n \wedge A[i] \neq x$ **do**
  $i := i+1$
  endwhile;
  if $A[i] \neq x$ **then** $i := 0$ **endif**

- yritetään suorittaa *etsintä* symbolisesti:

  $\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0\ \rangle$
  $i := 1$;
  $\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0 \wedge i = 1\ \rangle$
  while $i < n \wedge A[i] \neq x$ **do**
  $\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0 \wedge i = ?? \wedge A[??] \neq x_0\ \rangle$
  …
  ei tule mitään!

⇒ symbolinen suoritus ei yleensä onnistu, jos ohjelmassa on silmukoita

⇒ tarvitaan jokin muu tapa järkeillä ohjelman toiminnasta

Parempi todistuskeino

- kirjoita spesifikaation osittaisen oikeellisuuden vaatimus muodossa

  {*alkuehto*} *ohjelma* {*loppuehto*}

- koristele ohjelma **sopivasti valituilla** tilapredikaateilla siten, että ohjelman alussa on {*alkuehto*} ja lopussa {*loppuehto*}
- osoita kukin predikaatti voimassa olevaksi käyttäen hyväksi edellistä predikaattia ja välissä suoritettavia lauseita
  - silmukoille on erikoissääntöjä, joihin palataan

  ⇒ ohjelma antaa lopussa oikean tuloksen, jos se pääsee sinne asti
- predikaattien avulla osoita, että koskaan ei tehdä mitään kiellettyä
  - nollalla jako, taulukon indeksin ylivuoto, …

  ⇒ ohjelma ei pysähdy ennen loppua
- osoita, että jokaisesta silmukasta tullaan lopulta pois
  - keinoihin palataan

  ⇒ ohjelma pysähtyy

⇒ ohjelma ajaa loppuun, pysähtyy sinne ja antaa oikean tuloksen

Esimerkki: *etsintä* koristeltuna tilapredikaateilla

- alla olevat tilapredikaatit on valittu tietäen, miten *etsintä* toimii
- jatkossa osa niistä johdetaan muista
- jatkossa silmukka käsitellään hieman toisin

$\{ n \geq 1 \}$
$i := 1;$
$\{ n \geq 1 \wedge i = 1 \}$
while $i < n \wedge A[i] \neq x$ **do**
  $\{ 1 \leq i < n \wedge \forall j\ ; 1 \leq j \leq i\text{: } A[j] \neq x \}$
  $i := i+1$
  $\{ 1 < i \leq n \wedge \forall j\ ; 1 \leq j \leq i-1\text{: } A[j] \neq x \}$
endwhile;
$\{ 1 \leq i \leq n \wedge \forall j\ ; 1 \leq j \leq i-1\text{: } A[j] \neq x \wedge$
  $(i \geq n \vee A[i] = x) \}$
if $A[i] \neq x$ **then**
  $\{ n \geq 1 \wedge i = n \wedge \forall j\ ; 1 \leq j \leq i\text{: } A[j] \neq x \}$
  $i := 0$
  $\{ n \geq 1 \wedge i = 0 \wedge \forall j\ ; 1 \leq j \leq n\text{: } A[j] \neq x \}$
(* **else**
  $\{ 1 \leq i \leq n \wedge \forall j\ ; 1 \leq j \leq i-1\text{: } A[j] \neq x \wedge$
    $A[i] = x \}$ *)
endif
$\{ 1 \leq i \leq n \wedge A[i] = x \wedge \forall j\ ; 1 \leq j \leq i-1\text{: } A[j] \neq x$
  $\vee\ i = 0 \wedge \forall j\ ; 1 \leq j \leq n\text{: } A[j] \neq x \}$

Tilapredikaattien valinta

- tilapredikaattien ei tarvitse määritellä tilaa täydellisesti
- niiden on oltava **tarpeeksi vahvoja**, niin että ohjelma voidaan osoittaa niiden avulla oikeaksi

- ne **eivät** saa olla **liian vahvoja**
  - eivät saa väittää epätosia asioita
  - paikkansa pitävän, vaikeasti todistettavan, koko ohjelman todistuksen kannalta turhan ominaisuuden todistaminen on hyödytöntä
  - liika vahvuus voi estää tilapredikaatin sieventämisen helppoon muotoon

$\Rightarrow$ valinta vaatii näkemystä ohjelman toiminnasta
  - mikä on totta missäkin kohti
  - mitkä totuudet ovat olennaisia

- kohta opimme laskutekniikoita, joilla suuri osa tilapredikaateista voidaan johtaa muista
- silti kaksi asiaa jää todistajan luovuuden varaan:
  - tilapredikaattien heikentäminen sopivasti tarvittaessa
  - silmukan käsittelyssä tarvittavien tilapredikaattien valinta

Miten tulkitaan ohjelman keskellä olevat kulmasulkeisiin asetetut tilapredikaatit?

$\langle x \neq 0 \rangle$
$a := 1;$
$\langle x \neq 0 \wedge a = 1 \rangle$
if $x < 0$ **then**
  $\langle x < 0 \wedge a = 1 \rangle\ x := -x\ \langle x > 0 \wedge a = 1 \rangle$
endif
$\langle x > 0 \wedge a = 1 \rangle$

- ohjelman kohdan l *etenemisen vertailukohta* on
  - ohjelman alku, jos l ei ole minkään rakenteellisen lauseen sisällä
  - muutoin se on lähinnä ympäröivän rakenteellisen lauseen sen haaran alku, jossa l on

- esim. alla P_0 sijaitsee kaikkien P_i etenemisen vertailukohdassa, ja vastaavasti Q, R ja S

$\langle P_0 \rangle\ a := 1; \langle P_1 \rangle\ y := n+1; \langle P_2 \rangle$
while $a < y$ **do**
  $\langle Q_0 \rangle\ v := (a+y)$ div $2; \langle Q_1 \rangle$
  if $A[v] <$ *avain* **then** $\langle R_0 \rangle\ a := v+1\ \langle R_1 \rangle$
  else $\langle S_0 \rangle\ y := v\ \langle S_1 \rangle$
  endif $\langle Q_2 \rangle$
endwhile
$\langle P_3 \rangle$

- tällä kurssilla käytettävä tulkinta:
  kohdassa l sijaitseva merkintä $\langle P \rangle$ väittää, että jos alkuehto päti ohjelmaa käynnistettäessä, niin
  - P pätee aina kun suoritus on kohdassa l
  - jos suoritus on nyt l:n etenemisen vertailukohdassa, niin se on lopulta kohdassa l

Yhteenveto tilapredikaattien tulkinnoista eri kohdissa

- ohjelman alussa sijaitseva merkintä $\{P\}$ tai $\langle P \rangle$ määrittelee, että ohjelman alkuehto on P
- jos ohjelman alkuehtoa ei ole määritelty, se on **true**
- kohdassa l sijaitseva merkintä $\{P\}$ väittää, että P pätee aina kun suoritus on kohdassa l, edellyttäen että alkuehto päti ohjelmaa käynnistettäessä
- kohdassa l sijaitseva merkintä $\langle$**true**$\rangle$ väittää, että jos suoritus on l:n etenemisen vertailukohdassa, niin se on lopulta kohdassa l, edellyttäen että alkuehto päti ohjelmaa käynnistettäessä
- merkintä $\langle P \rangle$ väittää saman kuin merkinnät $\{P\}$ ja $\langle$**true**$\rangle$ yhdessä

Ohjelman koristelua koskevia yleisiä lakeja

- olkoot
  - S nollan tai useamman peräkkäisen lauseen jono
  - P ja Q tilapredikaatteja
- jos $\langle P \rangle\ S\ \langle Q \rangle$
  niin $\{P\}\ S\ \{Q\}$
- jos $\{P\}\ S\ \{Q\}$, $P_v \Rightarrow P$ ja $Q \Rightarrow Q_h$
  niin $\{P_v\}\ S\ \{Q_h\}$
  - muistisääntö: v ~ vahvempi, h ~ heikompi
  - perustelu: jos tila ennen S:n suoritusta täyttää P_v:n, niin se täyttää myös P:n
  - jos tila S:n suorituksen jälkeen täyttää Q:n, niin se täyttää myös Q_h:n
- jos $\langle P \rangle\ S\ \langle Q \rangle$, $P_v \Rightarrow P$ ja $Q \Rightarrow Q_h$
  niin $\langle P_v \rangle\ S\ \langle Q_h \rangle$
  - perustelu: kuten äsken
- jos $\{P\}\ S\ \{Q_1\}$ ja $\{P\}\ S\ \{Q_2\}$
  niin $\{P\}\ S\ \{Q_1 \wedge Q_2\}$
  - perustelu: jos tila S:n suorituksen jälkeen täyttää Q_1:n ja Q_2:n, niin se täyttää $(Q_1 \wedge Q_2)$:n
- jos $\langle P \rangle\ S\ \langle Q_1 \rangle$ ja $\langle P \rangle\ S\ \langle Q_2 \rangle$
  niin $\langle P \rangle\ S\ \langle Q_1 \wedge Q_2 \rangle$
- jopa
  jos $\langle P \rangle\ S\ \langle Q_1 \rangle$ ja $\{P\}\ S\ \{Q_2\}$
  niin $\langle P \rangle\ S\ \langle Q_1 \wedge Q_2 \rangle$
  - riittää, että perille pääsy on taattu yhden kerran

Harjoitustehtäviä

1. Suorita symbolisesti seuraava ohjelma. Mitä se tekee (olettaen, että ei satu ylivuotoja tms.)?

   $x := x+y$; $y := x-y$; $x := x-y$

2. Tehtävänä on osoittaa, että seuraava ohjelma sijoittaa z:aan suuremman muuttujien x ja y arvoista.

   if $x > y$ **then** $z := x$
   else $z := y$
   endif

(a) Kirjoita tilapredikaatti, joka sanoo seuraavat asiat:
- z:n arvo on jompi kumpi x:n ja y:n alkuarvoista
- z:n arvo on vähintään x:n alkuarvo
- z:n arvo on vähintään y:n alkuarvo.

(b) Suorita ohjelma symbolisesti.

(c) Osoita, että (a)-kohdassa kirjoittamasi predikaatti pätee ohjelman lopputilassa.

3.2 Lauseiden semantiikka

Mihin perustuen voidaan osoittaa, että jokin predikaatti on voimassa jossain kohti ohjelmaa?

- ohje oli: “käyttäen hyväksi edellistä predikaattia ja välissä suoritettavia lauseita”
- miten välissä olevia lauseita voi käyttää hyväksi?

$\Rightarrow$ tarvitaan päättelysääntöjä muotoa
$\{P\}$ *lause* $\{Q\}$ ja / tai $\langle P\rangle$ *lause* $\langle Q\rangle$
jokaiselle lausetyypille

- tällaisten päättelysääntöjen täytyy jotenkin nojautua lauseiden merkitykseen

Usein ajatellaan, että lauseen *merkitys* (eli *semantiikka*) on tapa, jolla se muuttaa ohjelman tilaa

- ohjelman tila =
  - tieto suorituksen sijainnista +
  - luettelo muuttujien arvoista
- esim. jos alussa $A = \langle 3,1,6,2,1,1,1\rangle$ (jolloin $n = 7$), $x = 2$ ja $i = -95$, niin etsintäohjelman tila eri askelmäärien jälkeen on
  0: sij = 1, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = -95$
  1: sij = 2, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = 1$
  2: sij = 3, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = 1$
  3: sij = 2, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = 2$
  7: sij = 2, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = 4$
  8: sij = 4, $A = \langle 3,1,6,2,1,1,1\rangle$, $n = 7$, $x = 2$, $i = 4$

```
1:  i := 1;
2:  while i < n ∧ A[i] ≠ x do
3:      i := i+1
    endwhile;
4:  …
```


Tällainen tapa esittää lauseen merkitys sopii huonosti ohjelmien todistamisessa tarvittavaan päättelyyn

- jos kaikille muuttujille annetaan konkreettinen arvo, niin voidaan käsitellä vain yhtä tapausta kerrallaan
- jos muuttujien arvot esitetään symbolisesti, niin kyseessä on symbolinen suoritus
  $\Rightarrow$ ei onnistu

Ohjelmien todistamista varten on kehitetty toinen tapa esittää lauseen merkitys:

heikoin esiehto (*weakest precondition*)

- olkoon S lause (tai ohjelmanpätkä) ja Q tilaa koskeva predikaatti
- $wp(S,Q)$ (eli Q:n heikoin esiehto S:n suhteen) on heikoin tilaa koskeva ehto, joka riittää takaamaan, että k.o. tilassa käynnistetty S:n suoritus pysähtyy, ja lopputilassa pätee Q
- toisin sanoen:
  $P \Rightarrow wp(S,Q)$
  jos ja vain jos
  $\langle P\rangle\ S\ \langle Q\rangle$
- $wp(S,Q)$ on siis funktio, joka ottaa lauseen ja sen jälkeistä tilaa kuvaavan predikaatin, ja tuottaa lausetta edeltävää tilaa kuvaavan predikaatin
  - ns. *predikaattimuunnin* (*predicate transformer*)

Esimerkkejä ($x \in \boldsymbol{R}$, $n \in \boldsymbol{Z}$)

- $wp(\ x := x-2,\ x > 0\ ) \Leftrightarrow x > 2$
- $wp(\ \textbf{if}\ x < 0\ \textbf{then}\ x := -x\ \textbf{endif},\ x > 0\ ) \Leftrightarrow x \neq 0$
- $wp(\ \textbf{while}\ n > 0\ \textbf{do}\ n := n-1\ \textbf{endwhile},\ n = 0\ ) \Leftrightarrow n \geq 0$

- $wp(\ x := 0,\ x = 0\ ) \Leftrightarrow$ **true**
- $wp(\ x := 1,\ x = 0\ ) \Leftrightarrow$ **false**
- $wp(\ \textbf{while}\ n \neq 0\ \textbf{do}\ n := n+1\ \textbf{endwhile},\ \textbf{true}\ ) \Leftrightarrow n \leq 0$

Annamme kohta *wp*-merkityksen joukolle tavallisimpia lauseita

- todellisten ohjelmointikielten lauseilla on usein monimutkaisia sivuvaikutuksia
  - esim. C: `x = ++i;`
  - esim. Pascal:

```
function plusplus( var y : integer ) : integer;
begin plusplus := y; y := y+1 end;
        ...
x := plusplus(i)
```

$\Rightarrow$ todellisten kielten verifiointiteorioista tulee sekavia

$\Rightarrow$ rajoitumme yksinkertaistettuihin lauseisiin, joilla ei ole sivuvaikutuksia

- teorian soveltajan on osattava tunnistaa sivuvaikutukset ym. poikkeukset teorian oletuksista, ja otettava ne huomioon!

Yksinkertaisen sijoituslauseen *wp*-semantiikka

- olkoon annettu predikaatti Q sekä sijoituslause S muotoa
  $x :=$ *lauseke*
  jossa
  - *lauseke*:n laskeminen ei aiheuta sivuvaikutuksia
  - x on yksittäinen muuttuja (ei esim. taulukon alkio)

- olkoon $\Uparrow lauseke$ tilaa koskeva predikaatti, joka pätee täsmälleen silloin, kun *lauseke*:n laskeminen varmasti onnistuu
  - ts. ei esiinny nollalla jakoa tms.

  $\Rightarrow \langle P \rangle\ x := lauseke\ \langle \mathbf{true} \rangle$ jos ja vain jos $P \Rightarrow \Uparrow lauseke$
- tila S:n suorituksen jälkeen on muuten sama kuin ennen, paitsi sijainti ja x:n arvo ovat muuttuneet

$\Rightarrow$ jos jälkitilassa pätee predikaatti Q, niin alkutilassa päti Q siten muutettuna, että jokainen x:n esiintymä on korvattu *lauseke*:lla

- tällä kurssilla merkintä $Q[x \leftarrow lauseke]$ tarkoittaa predikaattia, joka saadaan korvaamalla jokainen x:n vapaa esiintymä Q:ssa *lauseke*:eella
  - ns. *tekstikorvaus* (*textual substitution*)
  - (muut kirjoittajat käyttävät yleensä merkintää $Q^x_{lauseke}$, G-S myös $Q[x := lauseke]$ ja Backhouse $Q(x)$ vs. $Q(lauseke)$)
- siis

  jos $\langle P \rangle\ x := lauseke\ \langle Q \rangle$,

  niin $P \Rightarrow \Uparrow lauseke$ ja $P \Rightarrow Q[x \leftarrow lauseke]$
- toisaalta $\{\ Q[x \leftarrow lauseke]\ \}\ x := lauseke\ \{Q\}$
- siis $\langle P \rangle\ x := lauseke\ \langle Q \rangle$ jos ja vain jos $P \Rightarrow \Uparrow lauseke$ ja $P \Rightarrow Q[x \leftarrow lauseke]$ eli $P \Rightarrow \Uparrow lauseke \wedge Q[x \leftarrow lauseke]$
- siis

  $wp(x := lauseke, Q) \Leftrightarrow \Uparrow lauseke \wedge Q[x \leftarrow lauseke]$

Esimerkkejä

- $wp(\ x := x{-}2,\ x > 0\ ) \Leftrightarrow$ $\mathbf{true} \wedge (x > 0)[x \leftarrow x{-}2] \Leftrightarrow x{-}2 > 0 \Leftrightarrow x > 2$
- $wp(\ y := x{-}y,\ x = x_0\ ) \Leftrightarrow$ $\mathbf{true} \wedge (x = x_0)[y \leftarrow x{-}y] \Leftrightarrow x = x_0$
- $wp(\ y := x{-}y,\ y = x_0\ ) \Leftrightarrow$ $\mathbf{true} \wedge (y = x_0)[y \leftarrow x{-}y] \Leftrightarrow x{-}y = x_0$
- $wp(\ y := {}^8\!/_x,\ y > 4\ ) \Leftrightarrow$ $x \neq 0 \wedge (y > 4)[y \leftarrow {}^8\!/_x] \Leftrightarrow x \neq 0 \wedge {}^8\!/_x > 4 \Leftrightarrow$ $x \neq 0 \wedge (\ 0 \le x < 2 \vee 0 \ge x > 2\ ) \Leftrightarrow 0 < x < 2$
- $wp(\ x := 0,\ x = 0\ ) \Leftrightarrow$ $\mathbf{true} \wedge (x = 0)[x \leftarrow 0] \Leftrightarrow 0 = 0 \Leftrightarrow \mathbf{true}$
- $wp(\ x := 1,\ x = 0\ ) \Leftrightarrow$ $\mathbf{true} \wedge (x = 0)[x \leftarrow 1] \Leftrightarrow 1 = 0 \Leftrightarrow \mathbf{false}$

wp:n määritelmän nojalla saadaan yksinkertaiselle sijoituslauseelle kaksi päättelysääntöä

- $\{\ Q[x \leftarrow lauseke]\ \}\ x := lauseke\ \{Q\}$
- $\langle\ \Uparrow lauseke \wedge Q[x \leftarrow lauseke]\ \rangle\ x := lauseke\ \langle Q \rangle$

Edellä esitetyt esimerkit voidaan käydä läpi myös päättelysääntömuodossa:

- $\langle x > 2 \rangle$ $x := x{-}2$ $\langle x > 0 \rangle$
- $\langle x = x_0 \rangle$ $y := x{-}y$ $\langle x = x_0 \rangle$
- $\langle x{-}y = x_0 \rangle$ $y := x{-}y$ $\langle y = x_0 \rangle$
- $\langle 0 < x < 2 \rangle$ $y := {}^8\!/_x$ $\langle y > 4 \rangle$
- $\langle \mathbf{true} \rangle$ $x := 0$ $\langle x = 0 \rangle$
- $\langle \mathbf{false} \rangle$ $x := 1$ $\langle x = 0 \rangle$

Varoitus! Edellä olevat pätevät vain, jos sijoituksen kohteena on yksittäinen muuttuja

- esimerkki: kaava antaisi

  $wp(\ A[i] := 5,\ A[i] = 5 \wedge A[j] = 0 \wedge i = j\ )$

  $\Leftrightarrow\ 5 = 5 \wedge A[j] = 0 \wedge i = j \Leftrightarrow A[j] = 0 \wedge i = j,$

  joten saisimme virheellisesti

  $\langle\ A[j] = 0 \wedge i = j\ \rangle$

  $\quad A[i] := 5$

  $\langle\ A[i] = 5 \wedge A[j] = 0 \wedge i = j\ \rangle$

skip-lauseen *wp*-semantiikka ja päättelysäännöt

- joskus on kätevä käyttää lausetta, joka ei tee mitään
  - **skip**-lause
- koska **skip** ei muuta tilaa, niin
  - $wp(\mathbf{skip}, Q) \Leftrightarrow Q$
  - $\langle P \rangle\ \mathbf{skip}\ \langle P \rangle$

Peräkkäin kytkettyjen lauseiden *wp*-semantiikka

- jos $\langle P \rangle\ S_1; S_2\ \langle Q \rangle$,

  niin $\langle P \rangle\ S_1\ \langle\ wp(S_2, Q)\ \rangle$,

  joten $P \Rightarrow wp(\ S_1,\ wp(S_2, Q)\ )$
- jos $P \Rightarrow wp(\ S_1,\ wp(S_2, Q)\ )$,

  niin $\langle P \rangle\ S_1\ \langle\ wp(S_2, Q)\ \rangle\ S_2\ \langle Q \rangle$,

  joten $\langle P \rangle\ S_1; S_2\ \langle Q \rangle$
- siis $wp(S_1; S_2, Q) \Leftrightarrow wp(\ S_1,\ wp(S_2, Q)\ )$

Esimerkki: laskettava $wp(S, y = x_0 \wedge x = y_0)$, kun S on

$x := x{+}y;\ y := x{-}y;\ x := x{-}y$

- edetään takaperin
- viimeinen lause:

  $wp(x := x{-}y,\ y = x_0 \wedge x = y_0) \Leftrightarrow y = x_0 \wedge x{-}y = y_0$
- kaksi viimeistä lausetta:

  $wp(\ y := x{-}y;\ x := x{-}y,\ y = x_0 \wedge x = y_0\ ) \Leftrightarrow$

  $wp(\ y := x{-}y;\ wp(x := x{-}y,\ y = x_0 \wedge x = y_0)\ ) \Leftrightarrow$

  $wp(\ y := x{-}y;\ y = x_0 \wedge x{-}y = y_0\ ) \Leftrightarrow$

  $x{-}y = x_0 \wedge x{-}(x{-}y) = y_0 \Leftrightarrow x{-}y = x_0 \wedge y = y_0$
- koko ohjelma:

  $wp(\ x := x{+}y;\ y := x{-}y;\ x := x{-}y,\ y = x_0 \wedge x = y_0\ ) \Leftrightarrow$

  $wp(\ x := x{+}y;\ wp(y := x{-}y;\ x := x{-}y,\ y = x_0 \wedge x = y_0)) \Leftrightarrow$

  $wp(\ x := x{+}y,\ x{-}y = x_0 \wedge y = y_0\ ) \Leftrightarrow$

  $x{+}y{-}y = x_0 \wedge y = y_0 \Leftrightarrow x = x_0 \wedge y = y_0$
- vastaus on $x = x_0 \wedge y = y_0$
  - siis ohjelma vaihtaa x:n ja y:n arvot

Vastaavat peräkkäisten lauseiden päättelysäännöt

- jos $\{P\}\ S_1\ \{Q\}$ ja $\{Q\}\ S_2\ \{R\}$

  niin $\{P\}\ S_1; S_2\ \{R\}$
- jos $\langle P \rangle\ S_1\ \langle Q \rangle$ ja $\langle Q \rangle\ S_2\ \langle R \rangle$

  niin $\langle P \rangle\ S_1; S_2\ \langle R \rangle$

Edellinen esimerkki päättelysäännöillä

$\langle x = x_0 \wedge y = y_0 \rangle$ (* viimeiseksi tämä *)

$\quad x := x{+}y;$ (* huom! $(x{+}y) - y = x$ *)

$\langle x{-}y = x_0 \wedge y = y_0 \rangle$ (* kolmanneksi tämä *)

$\quad y := x{-}y;$ (* huom! $x - (x{-}y) = y$ *)

$\langle y = x_0 \wedge x{-}y = y_0 \rangle$ (* toiseksi tämä *)

$\quad x := x{-}y$

$\langle y = x_0 \wedge x = y_0 \rangle$ (* laskut aloitetaan tästä *)

If-lauseen perusmuodon *wp*-semantiikka

- **if**-lauseen perusmuoto on
  if B **then** S_1 **else** S_2 **endif**
- B:n laskennan täytyy onnistua: $\Uparrow B$
- jos valitaan **then**-haara, niin S_1:n esiehdon täytyy toteutua: $B \Rightarrow wp(S_1, Q)$
  – sama ilman implikaationuolta: $\neg B \vee wp(S_1, Q)$
- jos valitaan **else**-haara, niin S_2:n esiehdon täytyy toteutua: $\neg B \Rightarrow wp(S_2, Q)$
  – sama ilman implikaationuolta: $B \vee wp(S_2, Q)$

$\Rightarrow$ kaiken kaikkiaan
$wp($ **if** B **then** S_1 **else** S_2 **endif**, Q $) \Leftrightarrow$
$\Uparrow B \wedge (\neg B \vee wp(S_1, Q)) \wedge (B \vee wp(S_2, Q))$

- tälle on myös vaihtoehtoinen muoto, joka on joskus kätevämpi laskuissa:
  $wp($ **if** B **then** S_1 **else** S_2 **endif**, Q $) \Leftrightarrow$
  $\Uparrow B \wedge (B \wedge wp(S_1, Q) \vee \neg B \wedge wp(S_2, Q))$

Esimerkki: laskettava
$wp($ **if** $x > y$ **then** $d := x - y$ **else** $d := y - x$ **endif**, $d > 0$ $)$

- $wp(d := x-y,\ d > 0) \Leftrightarrow x - y > 0 \Leftrightarrow x > y$
- $wp(d := y-x,\ d > 0) \Leftrightarrow y - x > 0 \Leftrightarrow x < y$

$\Rightarrow$ vastaus $\Leftrightarrow$
true $\wedge\ (\neg(x > y) \vee x > y) \wedge (x > y \vee x < y) \Leftrightarrow x \neq y$

If-lauseen perusmuodon päättelysäännöt

- jos $\{P \wedge B\}\ S_1\ \{Q\}$ ja $\{P \wedge \neg B\}\ S_2\ \{Q\}$
  niin
  $\{P\}$
  if B **then** S_1 **else** S_2 **endif**
  $\{Q\}$

- jos $P \Rightarrow \Uparrow B$, $\langle P \wedge B \rangle\ S_1\ \langle Q \rangle$ ja
  $\langle P \wedge \neg B \rangle\ S_2\ \langle Q \rangle$
  niin
  $\langle P \rangle$
  if B **then** S_1 **else** S_2 **endif**
  $\langle Q \rangle$

Edellinen esimerkki päättelysäännöillä

$\langle x \neq y \rangle$
if $x > y$ **then** $d := x - y$ **else** $d := y - x$ **endif**
$\langle d > 0 \rangle$

- sama kaikki välivaiheet näyttäen

$\langle x \neq y \rangle$
if $x > y$ **then**
  $\langle x > y \rangle$
  $d := x - y$
  $\langle d > 0 \rangle$
else
  $\langle x \leq y \wedge x \neq y \rangle$ (* eli $x < y$ *)
  $d := y - x$
  $\langle d > 0 \rangle$
endif
$\langle d > 0 \rangle$

If-lause ilman **else**-haaraa

- **if** B **then** S **endif** on sama kuin
  if B **then** S **else skip endif**

$\Rightarrow$ voimme laskea
$wp($ **if** B **then** S **endif**, Q $) \Leftrightarrow$
$wp($ **if** B **then** S **else skip endif**, Q $) \Leftrightarrow$
$\Uparrow B \wedge (\neg B \vee wp(S, Q)) \wedge (B \vee wp($**skip**, $Q))) \Leftrightarrow$
$\Uparrow B \wedge (\neg B \vee wp(S, Q)) \wedge (B \vee Q)$

- siis
  $wp($ **if** B **then** S **endif**, Q $) \Leftrightarrow$
  $\Uparrow B \wedge (\neg B \vee wp(S, Q)) \wedge (B \vee Q)$
- vaihtoehtoinen muoto
  $wp($ **if** B **then** S **endif**, Q $) \Leftrightarrow$
  $\Uparrow B \wedge (B \wedge wp(S, Q) \vee \neg B \wedge Q)$
- samalla tavalla saadaan päättelysäännöt:
  – jos $\{P \wedge B\}\ S\ \{Q\}$ ja $P \wedge \neg B \Rightarrow Q$, niin
    $\{P\}$
    if B **then** S **endif**
    $\{Q\}$
  – jos $P \Rightarrow \Uparrow B$, $\langle P \wedge B \rangle\ S\ \langle Q \rangle$ ja $P \wedge \neg B \Rightarrow Q$
    niin
    $\langle P \rangle$
    if B **then** S **endif**
    $\langle Q \rangle$

Esimerkki

- osoitettava (A indeksoidaan $1 \ldots n$)
  $\langle\ 1 \leq i \leq n \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\wedge\ (\ i \geq n \vee A[i] = x\ )\ \rangle$
  if $A[i] \neq x$ **then** $i := 0$ **endif**
  $\langle\ 1 \leq i \leq n \wedge A[i] = x \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\vee\ i = 0 \wedge \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x\ \rangle$
- tilapredikaatit ovat seuraavat:
  – $P \Leftrightarrow 1 \leq i \leq n \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\wedge\ (\ i \geq n \vee A[i] = x\ )$
  – $Q \Leftrightarrow 1 \leq i \leq n \wedge A[i] = x \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\vee\ i = 0 \wedge \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x$
  – $B \Leftrightarrow A[i] \neq x$
- $P \Rightarrow 1 \leq i \leq n \Rightarrow \Uparrow B$ ·/·

- $wp(i := 0,\ Q) \Leftrightarrow$
  $1 \leq 0 \leq n \wedge \ldots \vee\ 0 = 0 \wedge \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x$
  $\Leftrightarrow \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x$
  $P \wedge B \Rightarrow 1 \leq i \leq n \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x \wedge$
  $i \geq n \wedge A[i] \neq x$
  $\Rightarrow i = n \wedge \forall\, j\ ;\ 1 \leq j \leq n \colon A[j] \neq x$
  $\Rightarrow wp(i := 0,\ Q)$
  joten $\langle P \wedge B \rangle\ i := 0\ \langle Q \rangle$
- $P \wedge \neg B \Leftrightarrow$
  $1 \leq i \leq n \wedge \forall\, j\ ;\ 1 \leq j \leq i-1 \colon A[j] \neq x \wedge A[i] = x$
  $\Rightarrow Q$ ·/·

Epädeterministinen **if**-lause

- ohjelmointikielten teoreetikot käyttävät usein epädeterminististä **if**-lausetta
  if $B_1 \rightarrow S_1$
  [] $B_2 \rightarrow S_2$
  …
  [] $B_n \rightarrow S_n$
  fi
- toiminta
  – lauseen suorituksen alkaessa jokaisen B_i täytyy olla laskettavissa
  – jos mikään B_i ei toteudu, syntyy virhetilanne
  – muussa tapauksessa valitaan **mielivaltaisesti** jokin haara siten, että sen B_i toteutuu, ja suoritetaan vastaava S_i
- esimerkki
  if $x \geq 0 \rightarrow S_1$
  [] $x = 0 \rightarrow S_2$
  [] $x \leq 0 \rightarrow S_3$
  fi

- tavavalliset **if**-lauseet voidaan toteuttaa epädeterministisellä **if**:llä:
  if B **then** S_1 **else** S_2 **endif** on sama kuin
  if $B \to S_1$ [] $\neg B \to S_2$ **fi**
- *wp*-semantiikka
  $wp($ **if** $B_1 \to S_1$ [] … [] $B_n \to S_n$ **fi**, Q $) \Leftrightarrow$
  $\Uparrow B_1 \wedge \ldots \wedge \Uparrow B_n \wedge (B_1 \vee \ldots \vee B_n) \wedge$
  $(\neg B_1 \vee wp(S_1, Q)) \wedge \ldots \wedge (\neg B_n \vee wp(S_n, Q))$
- päättelysäännöt
  – jos $\{P \wedge B_i\}\ S_i\ \{Q\}$ kaikilla $i \in \{1, \ldots, n\}$, niin
  $\{P\}$
  if $B_1 \to S_1$ [] … [] $B_n \to S_n$ **fi**
  $\{Q\}$
  – jos $P \Rightarrow \Uparrow B_i$ ja $P \Rightarrow B_1 \vee \ldots \vee B_n$ ja
  $\langle P \wedge B_i \rangle\ S_i\ \langle Q \rangle$ kaikilla $i \in \{1, \ldots, n\}$,
  niin
  $\langle P \rangle$
  if $B_1 \to S_1$ [] … [] $B_n \to S_n$ **fi**
  $\langle Q \rangle$

Taulukkoon sijoittaminen

- edellä näimme, että kaava $wp(x := lauseke, Q) \Leftrightarrow \Uparrow lauseke \wedge Q[x \leftarrow lauseke]$ pätee vain, jos x on tavallinen muuttuja
- syy: tekstikorvaus ei ota huomioon, että taulukon alkioon $A[i]$ voidaan viitata muullakin nimellä
  – esim. $A[1]$ (jos $i = 1$) tai $A[j-3]$ (jos $j = i+3$)
- ratkaisu: ajatellaan, että
  – sijoituksen kohteena on **koko taulukko**, esim.
  $A[2] := 3$ ➠ $A[1\ldots3] := \langle A[1], 3, A[3] \rangle$
  – tekstikorvaus kohdistuu **taulukkomuuttujan nimeen ilman indeksiä**

$\Rightarrow$ tarvitaan merkintä sijoituksessa
$A[i] := e$, i laillisella alueella
muuttuneelle taulukolle:

$A([i] \leftarrow e)$ on taulukko, jolle pätee

$$A([i] \leftarrow e)[k] = \begin{cases} e, & \text{jos } k = i \\ A[k], & \text{muutoin} \end{cases}$$

- taulukkoon sijoituksen heikoin esiehto voidaan nyt laskea jakamalla tarvittaessa kaava osiin seuraavan esimerkin tapaan:
  $wp(A[i] := e, Q(A[i], A[j])) \Leftrightarrow$
  $\Uparrow e \wedge 1 \le i \le n \wedge Q(A([i] \leftarrow e)[i], A([i] \leftarrow e)[j]) \Leftrightarrow$
  $\Uparrow e \wedge 1 \le i \le n \wedge (\ j = i \wedge Q(e, e)$
  $\qquad\qquad \vee\ \ j \neq i \wedge Q(e, A[j])\ )$

Esimerkkejä (olettaen, että indeksit sallitulla alueella)

- $wp(A[i] := 2, A[j] = 0) \Leftrightarrow A([i] \leftarrow 2)[j] = 0 \Leftrightarrow$
  $j = i \wedge 2 = 0 \ \vee\ j \neq i \wedge A[j] = 0 \Leftrightarrow j \neq i \wedge A[j] = 0$
- $wp(\ A[i] := 5, A[i] = 5 \wedge A[j] = 0 \wedge i = j\ ) \Leftrightarrow$
  $A([i] \leftarrow 5)[i] = 5 \wedge A([i] \leftarrow 5)[j] = 0 \wedge i = j \Leftrightarrow$
  $5 = 5 \wedge (j = i \wedge 5 = 0 \vee j \neq i \wedge A[j] = 0) \wedge i = j \Leftrightarrow$
  false
- $wp(A[A[i]+1] := 2, A[j] = 0) \Leftrightarrow$
  $A([A[i]+1] \leftarrow 2)[j] = 0 \Leftrightarrow$
  $j = A[i]+1 \wedge 2 = 0 \ \vee\ j \neq A[i]+1 \wedge A[j] = 0 \Leftrightarrow$
  $j \neq A[i]+1 \wedge A[j] = 0$
- $wp(A[i] := A[i]-A[j], A[i] = y_0 \wedge A[j] = x_0) \Leftrightarrow$
  $\quad j = i \wedge A[i]-A[j] = x_0 = y_0$
  $\quad \vee\ \ j \neq i \wedge A[i]-A[j] = y_0 \wedge A[j] = x_0 \Leftrightarrow$
  $j = i \wedge x_0 = y_0 = 0 \ \vee\ j \neq i \wedge A[i] = x_0+y_0 \wedge A[j] = x_0$

- koko ohjelma
  $A[i] := A[i]+A[j]$; $A[j] := A[i]-A[j]$; $A[i] := A[i]-A[j]$
  $\langle\ \ j = i \wedge x_0 = y_0 = 0\ \vee$
  $\quad j \neq i \wedge A[i] = x_0 \wedge A[j] = y_0\ \rangle$
  $A[i] := A[i]+A[j]$;
  $\langle\ \ j = i \wedge x_0 = y_0 = 0\ \vee$
  $\quad j \neq i \wedge A[i] = x_0+y_0 \wedge A[j] = y_0\ \rangle$
  $A[j] := A[i]-A[j]$;
  $\langle\ \ j = i \wedge x_0 = y_0 = 0\ \vee$
  $\quad j \neq i \wedge A[i] = x_0+y_0 \wedge A[j] = x_0\ \rangle$
  $A[i] := A[i]-A[j]$
  $\langle\ A[i] = y_0 \wedge A[j] = x_0\ \rangle$
- havaintoja viimeisestä esimerkistä
  – ohjelma ei laskekaan väärin ihan aina kun $j = i$
  – tapaus $j = i$ teetätti olennaisesti lisätyötä vain viimeisen sijoituslauseen kohdalla

$\Rightarrow$ taulukkoon sijoittamisen sääntö
– varmistaa, että erikoistapaukset eivät jää ottamatta huomioon
– usein ei ole niin työläs kuin miltä näyttää

Heikoimmille esiehdoille pätee hyödyllisiä lakeja:

- *pois suljetun ihmeen laki* (*law of the excluded miracle*):
  $wp(S,$ **false**$) \Leftrightarrow$ **false**
  – ts. ei ole olemassa tilaa, jossa käynnistetty S:n suoritus päättyy, ja lopputila on mahdoton
- yleensä $wp(S,$ **true**$) \neq$ **true**, koska *wp* takaa, että lauseen suoritus onnistuu
  – esim. jos $i \in \boldsymbol{Z}$ ja S on
  while $i \neq 0$ **do** $i := i-1$ **endwhile**
  niin $wp(S,$ **true**$) \Leftrightarrow i \ge 0$
- jos $P \Rightarrow Q$, niin $wp(S,P) \Rightarrow wp(S,Q)$

- $wp(S, P \wedge Q) \Leftrightarrow wp(S,P) \wedge wp(S,Q)$
- $wp(S, P \vee Q) \Leftarrow wp(S,P) \vee wp(S,Q)$
- $wp(S, P \vee Q) \not\Rightarrow wp(S,P) \vee wp(S,Q)$
  – vastaesimerkki:
  jos S sijoittaa x:ään satunnaisluvun 0 tai 1
  $x :=$ random(0, 1)
  niin
  $wp(S, x = 0 \vee x = 1) \Leftrightarrow$ **true** $\not\Rightarrow$
  $wp(S, x = 0) \vee wp(S, x = 1) \Leftrightarrow$
  false $\vee$ **false** $\Leftrightarrow$ **false**
  – toinen tapa sijoittaa x:ään 0 tai 1:
  if true $\to x := 0$ [] **true** $\to x := 1$ **fi**
- S on *deterministinen* (*deterministic*), jos ja vain jos kaikille tiloille t seuraava pätee kun S on käynnistetty tilassa t:
  jos S voi pysähtyä ja tuottaa lopputilan t',
  niin S varmasti pysähtyy ja tuottaa lopputilan t'
  – ts. jos S voi tehdä jotain, se varmasti tekee sen
  $\Rightarrow$ S:n toiminnalla on vain yksi mahdollinen lopputulos
- jos S on deterministinen, niin
  $wp(S, P \vee Q) \Leftrightarrow wp(S,P) \vee wp(S,Q)$

Harjoitustehtäviä

1. Laske ja sievennä seuraavat heikoimmat esiehdot.

(a) $wp(\ x := 0,\ x = y\ )$

(b) $wp(\ x := 3{\cdot}x,\ x \neq 0\ )$

(c) $wp(\ x := x{-}y,\ y > 0\ )$

(d) $wp(\ x := x{-}y,\ x > 0\ )$

(e) $wp(\ x := x{+}a;\ n := n{+}1,\ x = n{\cdot}a\ )$

(f) $wp(\ x := x{\cdot}a;\ n := n{+}1,\ x = a^n \wedge n > 0\ )$ (muista: $0^0 = 1$)

2. Laske ja sievennä seuraavat heikoimmat esiehdot.

(a) $wp($ **if** $i > j$ **then** $j := j{+}1$ **else** $i := i{+}1$ **endif**, $i \geq j\ )$

(b) $wp($ **if** $A[i] < x$ **then** $i := i{+}1$ **endif**,
$\forall\ j\ ;\ 1 \leq j < i\colon A[j] < x\ )$ (määritelty $A[1\ldots n]$)

3. Olkoon $Q \Leftrightarrow x{\cdot}b^i = a^n$.

(a) Laske $wp(b := b{\cdot}b,\ Q)$.

(b) Laske $wp(i := i$ div $2;\ b := b{\cdot}b,\ Q)$.

(c) Laske $wp(S, Q)$, kun S on
if i mod 2 = 1 **then** $x := b{\cdot}x$ **endif**;
$i := i$ div 2; $b := b{\cdot}b$.

4. Osoita alla oleva ohjelma oikeaksi, kun A:n indeksialue on $A[1\ldots n]$, ja
$P \Leftrightarrow \forall\ j\ ;\ 1 \leq j < i\colon A[j] \leq max \wedge \exists\ j\ ;\ 1 \leq j < i\colon A[j] = max$.

$\langle\ P \wedge i \leq n\ \rangle$
if $A[i] > max$ **then**
  $max := A[i]$
endif;
$i := i + 1$
$\langle\ P\ \rangle$

5. Laske ja sievennä seuraavat heikoimmat esiehdot olettaen, että A on $A[1\ldots n]$.

(a) $wp(\ A[i] := 5,\ \exists\ j\ ;\ i \leq j \leq n\colon A[i] \leq A[j]\ )$

(b) $wp(\ A[i] := 5,\ \exists\ j\ ;\ i \leq j \leq n\colon A[i] < A[j]\ )$

(c) $wp(\ A[i] := 5,\ \forall\ j\ ;\ i \leq j \leq n\colon A[i] \leq A[j]\ )$

(d) $wp(\ A[i] := 5,\ \forall\ j\ ;\ 1 \leq j \leq n\colon A[j] = A_0[j]\ )$

(e) $wp(\ A[i] := 5,\ A[A[i]] = 0\ )$

(f) $wp(\ A[i] := i,\ A[A[i]] = i\ )$

(g) $wp(\ A[i] := A[i{-}1] + A[i],\ A[i] = \sum_{j=1}^{i-1} A[j]\ )$

(h) $wp(\ A[i] := A[i{-}1] + A[i],\ A[i] = \sum_{j=1}^{i} A_0[j]\ )$

6. A on $A[1\ldots n]$. Laske, millä mahdollisimman heikolla esiehdolla seuraava ohjelma vaihtaa $A[i]$:n ja $A[j]$:n arvot.

$t := A[i];\ A[i] := A[j];\ A[j] := t$

7. Mikä on $wp($**while true do skip endwhile**$,\ Q)$?

8. Mitä seuraava ohjelma tekee? Todista, että se tekee sen. Millä tavalla ohjelma on epädeterminististinen?

if $x \leq y \wedge x \leq z \rightarrow t := x$
[] $y \leq x \wedge y \leq z \rightarrow t := y$
[] $z \leq x \wedge z \leq y \rightarrow t := z$
fi

9. Osoita, että $wp($**if**$,\ Q)$:lle annetut kaksi lauseketta ovat yhtäpitävät, ts.

$\Uparrow B \wedge (\neg B \vee wp(S_1, Q)) \wedge (B \vee wp(S_2, Q)) \Leftrightarrow$
$\Uparrow B \wedge (B \wedge wp(S_1, Q) \vee \neg B \wedge wp(S_2, Q))$

3.3 Silmukkalauseiden käsittely

Silmukkalauseen wp-semantiikka

- silmukkalauseen perusmuoto on
  while B **do** S **endwhile**
  missä
  – B on jokin ehto
  – S on lause (tai lausejono)
- yksinkertaisuuden vuoksi oletamme, että B on aina laskettavissa (ts. $\Uparrow B \Leftrightarrow$ **true**)
- heikoin ehto sille, että silmukkalause suoritetaan 0 kertaa ja tulos toteuttaa Q:n on
  $P_0 \Leftrightarrow \neg B \wedge Q$
- heikoin ehto sille, että silmukkalause suoritetaan 1 kerran ja tulos toteuttaa Q:n on
  $P_1 \Leftrightarrow B \wedge wp(S, \neg B \wedge Q) \Leftrightarrow B \wedge wp(S, P_0)$
- vastaavasti kahdelle suorituskerralle saadaan
  $P_2 \Leftrightarrow B \wedge wp(S, P_1)$
  ja n:lle
  $P_n \Leftrightarrow B \wedge wp(S, P_{n-1})$
- jos silmukkalauseen suoritus onnistuu ja tulos toteuttaa Q:n, niin silmukka kierretään n kertaa jollekin äärelliselle n ja alussa P_n pätee; samoin, jos tällainen n on olemassa, niin silmukkalauseen suoritus onnistuu ja tulos toteuttaa Q:n

$\Rightarrow$ $wp($**while** B **do** S **endwhile**$,\ Q) \Leftrightarrow \exists\ n\colon P_n$

Silmukkalauseen wp-semantiikkaa on hankala (tai mahdoton) käyttää laskuissa

- esimerkki: mitä on $wp(S,$ **true**$)$, kun S on:

```
while x ≠ 1 do
    if x mod 2 = 0 then x := x div 2
    else x := 3·x+1 endif
endwhile
```

- ratkeavuuden teoria
  $\Rightarrow$ on tilanteita, joissa ei ole edes periaatteessa mahdollista selvittää, päteekö
  $P \Rightarrow wp($**while**, **true**$)$

$\Rightarrow$ silmukkalauseiden käsittelyyn käytetään toista keinoa
– osittainen oikeellisuus: *invariantti*
– pysähtyvyys: *ylärajafunktio* (*bound function*)

Silmukkalauseen osittaisen oikeellisuuden todistaminen invarianttien avulla

- väittämä I on silmukkalauseen
  while B **do** S **endwhile**
  invariantti (*invariant*), jos ja vain jos
  $\{I \wedge B\}\ S\ \{I\}$
- jos I on silmukkalauseen invariantti, niin
  $\{I\}$ **while** B **do** S **endwhile** $\{I \wedge \neg B\}$
- väittämä muotoa
  $\{P\}$ **while** B **do** S **endwhile** $\{Q\}$
  voidaan siis osoittaa seuraavasti:
  1. keksi jokin sopiva invarianttikandidaatti I
  2. osoita, että $P \Rightarrow I$
  3. osoita, että $\{I \wedge B\}\ S\ \{I\}$
  4. osoita, että $I \wedge \neg B \Rightarrow Q$
- vielä ei tarvitse osoittaa, että $\Uparrow B$

Esimerkki: osoitettava (n, $i \in \boldsymbol{Z}$, a, $x \in \boldsymbol{R}$, oleta $a \neq 0$)

$\{n \geq 0\}$
$x := 1$; $i := 0$;
while $i < n$ **do**
  $x := x \cdot a$; $i := i+1$
endwhile
$\{x = a^n\}$

- ratkaisun alku
  $\{n \geq 0\}$
  $x := 1$; $i := 0$;
  $\{x = 1 \wedge i = 0 \wedge n \geq 0\}$
- silmukan käsittely
  1. invarianttikandidaatti: $x = a^i \wedge i \leq n$
  2. $x = 1 \wedge i = 0 \wedge n \geq 0 \;\Rightarrow\; x = a^i \wedge i \leq n$ $\cdot/\cdot$
  3. $wp(x := x \cdot a;\, i := i+1,\, x = a^i \wedge i \leq n) \;\Leftrightarrow$
     $wp(x := x \cdot a,\, x = a^{i+1} \wedge i+1 \leq n) \;\Leftrightarrow$
     $x \cdot a = a^{i+1} \wedge i+1 \leq n \;\Leftrightarrow\; x = a^i \wedge i < n$
     joten
     $\{x = a^i \wedge i \leq n \wedge i < n\}$
     $x := x \cdot a$; $i := i+1$
     $\{x = a^i \wedge i \leq n\}$
  4. $x = a^i \wedge i \leq n \wedge \neg(i < n) \;\Leftrightarrow\; x = a^i \wedge i = n \;\Rightarrow$
     $x = a^n$
- yhteen koottuna
  $\{n \geq 0\}$
  $x := 1$; $i := 0$;
  $\{x = 1 \wedge i = 0 \wedge n \geq 0\}$
  $\{$**inv:** $x = a^i \wedge i \leq n\}$
  while $i < n$ **do**
    $x := x \cdot a$; $i := i+1$
  endwhile
  $\{x = a^n\}$

Silmukkalauseen pysähtymisen todistaminen ylärajafunktion avulla

- *ylärajafunktio* (*bound function*, *variant*) on (tavallisesti kokonaislukuarvoinen) funktio, jolle
  – on raja (usein 0), jolla ja jota pienemmillä arvoilla ei lähdetä uudelle kierrokselle
  – arvo pienenee silmukan jokaisella kierroksella ainakin yhden yksikön verran

$\Rightarrow$ silmukka ei voi pyöriä loputtomiin

- erityisesti, jos
  – ylärajafunktion raja on 0
  – silmukkaan tultaessa ylärajafunktion arvo on n
  niin silmukka voi pyöriä korkeintaan n kierrosta
- ylärajafunktion ominaisuuksia todistettaessa käytetään tavallisesti hyväksi osittaisen oikeellisuuden todistamista varten valittua invarianttia
  $\Rightarrow$ saattaa olla tarpeen valita vahvempi invariantti kuin muuten tarvittaisiin
- siis: invariantilla I varustetun silmukan
  while B **do** S **endwhile**
  pysähtyminen voidaan todistaa seuraavasti:
  1. keksi jokin sopiva ylärajafunktio yr: $Tilat \to \boldsymbol{Z}$
  2. osoita, että $I \;\Rightarrow\; \Uparrow B$
  3. osoita, että $I \wedge B \;\Rightarrow\; yr > 0$
  4. osoita, että $\langle I \wedge B \wedge yr = yr_0 \rangle\, S\, \langle yr < yr_0 \rangle$, missä yr_0 on mielivaltainen kokonaisluku

Esimerkki: edellisen ohjelman pysähtymisen todistaminen

- sijoituslauseiden lausekkeet ovat aina laskettavissa
  $\Rightarrow$ ne onnistuvat ja pysähtyvät

$\langle n \geq 0 \rangle$
$x := 1$; $i := 0$;
$\langle x = 1 \wedge i = 0 \wedge n \geq 0 \rangle$
$\{$**inv:** $x = a^i \wedge i \leq n\}$
while $i < n$ **do**
  $x := x \cdot a$; $i := i+1$
endwhile
$\{x = a^n\}$

- silmukan pysähtymisen osoittaminen
  1. ylärajafunktio: $n-i$
  2. $i < n$ on aina laskettavissa
  3. $x = a^i \wedge i \leq n \wedge i < n \;\Rightarrow\; i < n \;\Leftrightarrow\; n-i > 0$
  4. $wp(x := x \cdot a;\, i := i+1,\, n-i < yr_0) \;\Leftrightarrow$
     $n-(i+1) < yr_0 \;\Leftrightarrow\; n-i \leq yr_0 \;\Leftarrow\; n-i = yr_0$,
     joten
     $\langle x = a^i \wedge i \leq n \wedge i < n \wedge n-i = yr_0 \rangle$
     $x := x \cdot a$; $i := i+1$
     $\langle n-i < yr_0 \rangle$

Esimerkki: etsintäohjelman osoittaminen oikeaksi

- osoitettava täysin oikeaksi seuraava ohjelma, kun $n \geq 1$, ja A ja x kiinteitä:

$\langle\, n \geq 1 \,\rangle$
$i := 1$;
while $i < n \wedge A[i] \neq x$ **do** $i := i+1$ **endwhile**;
if $A[i] \neq x$ **then** $i := 0$ **endif**
$\langle\, 1 \leq i \leq n \wedge A[i] = x \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
$\quad \vee\, i = 0 \wedge \forall\, j\, ;\, 1 \leq j \leq n \colon A[j] \neq x \,\rangle$

- sij. lauseen $wp \;\Rightarrow\; \langle\, n \geq 1 \,\rangle\; i := 1\; \langle\, n \geq 1 \wedge i = 1 \,\rangle$
- silmukkainvariantti todistuksineen:
  1. $I \;:\Leftrightarrow\; 1 \leq i \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
     (ts. i on sallitulla alueella, ja sitä edeltävissä ei ole etsittyä)
  2. $n \geq 1 \wedge i = 1 \;\Rightarrow\; I$ $\cdot/\cdot$
  3. $wp(i := i+1,\, I) \;\Leftrightarrow$
     $1 \leq i+1 \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i \colon A[j] \neq x \;\Leftrightarrow$
     $0 \leq i < n \wedge \forall\, j\, ;\, 1 \leq j \leq i \colon A[j] \neq x$
     toisaalta $I \wedge B \;\Leftrightarrow\; 1 \leq i \leq n \wedge$
     $\forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x \wedge i < n \wedge A[i] \neq x$
     $\Leftrightarrow\; 1 \leq i < n \wedge \forall\, j\, ;\, 1 \leq j \leq i \colon A[j] \neq x$
     $\Rightarrow\; wp(i := i+1,\, I)$
     joten
     $\{\, I \wedge i < n \wedge A[i] \neq x \,\}\; i := i+1\; \{\, I \,\}$
  4. $I \wedge \neg B \;\Leftrightarrow\; 1 \leq i \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
     $\wedge\, (i \geq n \vee A[i] = x)$

$\Rightarrow$ $\{\, n \geq 1 \wedge i = 1 \,\}$
while $i < n \wedge A[i] \neq x$ **do** $i := i+1$ **endwhile**
$\{\, 1 \leq i \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
$\quad \wedge\, (\, i \geq n \vee A[i] = x \,)\,\}$

- ylärajafunktio todistuksineen:
  1. $yr = n-i$
  2. $I \;\Rightarrow\; 1 \leq i \leq n$, joten $I \;\Rightarrow\; \Uparrow(\, i < n \wedge A[i] \neq x \,)$
  3. $I \wedge i < n \wedge A[i] \neq x \;\Rightarrow\; n-i > 0$ $\cdot/\cdot$
  4. $wp(i := i+1,\, n-i < yr_0) \;\Leftrightarrow\; n-(i+1) < yr_0 \;\Leftrightarrow$
     $n-i \leq yr_0$, joten
     $\langle I \wedge B \wedge n-i = yr_0 \rangle\; i := i+1\; \langle n-i < yr_0 \rangle$

$\Rightarrow$ silmukka pysähtyy

- nyt on osoitettu

  $\langle\, n \geq 1 \,\rangle$ $i := 1$;
  while $i < n \wedge A[i] \neq x$ **do** $i := i+1$ **endwhile**;
  $\langle\, 1 \leq i \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\wedge\, (\, i \geq n \vee A[i] = x\,)\, \rangle$

- loppuosa osoitettiin edellisessä alaluvussa

  $\langle\, 1 \leq i \leq n \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\wedge\, (\, i \geq n \vee A[i] = x\,)\, \rangle$
  if $A[i] \neq x$ **then** $i := 0$ **endif**
  $\langle\, 1 \leq i \leq n \wedge A[i] = x \wedge \forall\, j\, ;\, 1 \leq j \leq i-1 \colon A[j] \neq x$
  $\vee\, i = 0 \wedge \forall\, j\, ;\, 1 \leq j \leq n \colon A[j] \neq x\, \rangle$

Vaiheiden yhdistäminen

- edellä osoitimme erikseen silmukan osittaisen oikeellisuuden ja pysähtymisen
- käytännössä on usein kätevää tehdä ne yhtäaikaa

$\Rightarrow$ väittämä muotoa
$\langle P \rangle$ **while** B **do** S **endwhile** $\langle Q \rangle$
voidaan osoittaa seuraavasti:

1. keksi jokin sopiva invarianttikandidaatti I
2. keksi jokin sopiva ylärajafunktiokandidaatti yr
3. osoita, että $P \Rightarrow I$
4. osoita, että $I \Rightarrow \Uparrow B$
5. osoita, että $I \wedge B \Rightarrow yr > 0$
6. osoita, että $\langle I \wedge B \wedge yr = yr_0 \rangle\ S\ \langle I \wedge yr < yr_0 \rangle$, missä yr_0 on mielivaltainen kokonaisluku
7. osoita, että $I \wedge \neg B \Rightarrow Q$

Esimerkki: osoitetaan, että puolitushaku lopettaa, ja lopussa etsintävälin a, …, y leveys on 1

$\langle$ **true** $\rangle$
$a := 1$; $y := n+1$;
while $a < y$ **do**
  $v := (a+y)$ div 2;
  if $A[v] <$ *avain* **then** $a := v+1$
  else $y := v$
  endif
endwhile
$\langle\, 1 \leq a = y \leq n+1 \,\rangle$

- $A[1 \ldots n]$ ja *avain* kiinteitä
- alustukset on helppo käsitellä:

  $\langle$ **true** $\rangle$
  $a := 1$; $y := n+1$;
  $\langle\, a = 1 \wedge y = n+1 \,\rangle$
  while $a < y$ **do**
    $v := (a+y)$ div 2;
    if $A[v] <$ *avain* **then** $a := v+1$
    else $y := v$
    endif
  endwhile
  $\langle\, 1 \leq a = y \leq n+1 \,\rangle$

- silmukan todistamisessa tarvittavat tilapredikaatit:
  – $P \Leftrightarrow a = 1 \wedge y = n+1$
  – $Q \Leftrightarrow 1 \leq a = y \leq n+1$
  – $B \Leftrightarrow a < y$

1. invariantti:
   – etsintäväli on aluksi 1, …, $n+1$ ja sen tulisi kaveta yhden levyiseksi
   $\Rightarrow$ valitsemme $I \Leftrightarrow 1 \leq a \leq y \leq n+1$

2. ylärajafunktio:
   – välin a, …, y tulisi kaveta joka kierroksella
   $\Rightarrow$ valitsemme $yr = y - a$
3. $P \Rightarrow I$:
   – yleisoletuksien takia $n \geq 0$, joten $n+1 \geq 1$
   – siis $P \Rightarrow 1 = a \leq y = n+1 \Rightarrow I$ ·/·
4. $I \Rightarrow \Uparrow B$:
   – ehto $a < y$ on aina laskettavissa
5. $I \wedge B \Rightarrow yr > 0$:
   $I \wedge B \Rightarrow B \Leftrightarrow a < y \Rightarrow yr = y - a > 0$ ·/·
6. $\langle I \wedge B \wedge yr = yr_0 \rangle\ S\ \langle I \wedge yr < yr_0 \rangle$:
   työläs, osoitetaan kohta
7. $I \wedge \neg B \Rightarrow Q$:
   $I \wedge \neg B \Rightarrow 1 \leq a \leq y \leq n+1 \wedge a \geq y$
   $\Rightarrow 1 \leq a = y \leq n+1 \Leftrightarrow Q$ ·/·

Esimerkin kohdan 6 todistus

- **if**-lauseen haaroista saadaan
  – $wp(a := v+1,\ I \wedge yr < yr_0) \Leftrightarrow$
    $1 \leq v+1 \leq y \leq n+1 \wedge y-(v+1) < yr_0$
  – $wp(y := v,\ I \wedge yr < yr_0) \Leftrightarrow$
    $1 \leq a \leq v \leq n+1 \wedge v-a < yr_0$
- tästä voitaisiin laskea $wp($**if**$,\ I \wedge yr < yr_0)$, mutta näyttää johtavan isoon lausekkeeseen
- silmukan vartalon alkuosa näyttää helpolta

$\Rightarrow$ simuloimme silmukan vartalon alkuosan ja käytämme **if**-lauseen päättelysääntöä

- yllä olevat wp-kaavat sisältävät v:tä vain vertailuissa a:n, $(y-1)$:n jne. kanssa
  $\Rightarrow$ olennaista lienee vain v:n suuruus suhteessa a:han ja y:hyn
  $\Rightarrow$ otetaan v:n arvosta huomioon vain se
- silmukan vartalon alkuosa
  – $I \wedge B \wedge yr = yr_0 \Leftrightarrow 1 \leq a < y \leq n+1 \wedge y-a = yr_0$
  – $a < y \Rightarrow a < (a+y) / 2 < y$ ja
    $(a+y) / 2 - {}^1/_2 \leq (a+y)$ div $2 \leq (a+y) / 2$
    $\Rightarrow a \leq (a+y)$ div $2 < y$

    $\langle\, 1 \leq a < y \leq n+1 \wedge y-a = yr_0 \,\rangle$
    $v := (a+y)$ div 2;
    $\langle\, 1 \leq a \leq v < y \leq n+1 \wedge y-a = yr_0 \,\rangle$
- nyt voidaan osoittaa **if**-lauseen haarojen wp-ehdot
  – $1 \leq a \leq v < y \leq n+1 \wedge y-a = yr_0 \wedge A[v] <$ *avain*
    $\Rightarrow 1 \leq v+1 \leq y \leq n+1 \wedge y-(v+1) < y-a = yr_0$
    $\Rightarrow wp(a := v+1,\ I \wedge yr < yr_0)$
  – $1 \leq a \leq v < y \leq n+1 \wedge y-a = yr_0 \wedge A[v] \geq$ *avain*
    $\Rightarrow 1 \leq a \leq v \leq n+1 \wedge v-a < y-a = yr_0$
    $\Rightarrow wp(y := v,\ I \wedge yr < yr_0)$

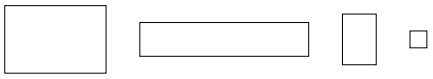
$\Rightarrow \langle I \wedge B \wedge yr = yr_0 \rangle$
$v := (a+y)$ div 2;
$\langle\, 1 \leq a \leq v < y \leq n+1 \wedge y-a = yr_0 \,\rangle$
if $A[v] <$ *avain* **then** $a := v+1$
else $y := v$
endif
$\langle I \wedge yr < yr_0 \rangle$

Havaintoja

- etenimme etuperin (symbolinen suoritus) tai takaperin (*wp*) sen mukaan mikä oli helpointa
  - helpot laskut ensin alta pois
  - $\Rightarrow$ mahdollisimman paljon tukea vaikeille kohdille
- lausetta $v := (a+y) \text{ div } 2$ simuloidessamme emme tallettaneet v:n tarkkaa arvoa
  - olennaista oli vain osoittaa, että $y-a$ pienenee
  - tarkka arvo olisi vaikeuttanut myöhempiä laskuja
  - tilapredikaatin sopiva heikentäminen!
- puolitushaun tavallista esiehtoa
  $\forall\, i\,;\, 1 \le i \le n-1\colon A[i] \le A[i+1]$
  ei tarvittu ollenkaan
  - $\Rightarrow$ algoritmi pysähtyy ja lopussa $1 \le a = y \le n+1$ vaikka taulukko ei olisikaan järjestyksessä
  - vikasietoisuutta!

Osittaisjärjestykset

- joukon A *osittaisjärjestys* on mikä tahansa A:n alkioiden välinen relaatio, jolle pätee $\forall\, a, b, c \in A$:
  - $a \le a$ (*refleksiivisyys*)
  - $a \le b \wedge b \le c \;\Rightarrow\; a \le c$ (*transitiivisuus*)
  - $a \le b \wedge b \le a \;\Rightarrow\; a = b$ (*antisymmetrisyys*)
- esimerkkejä
  - "$\le$" on reaalilukujen välinen osittaisjärjestys
  - "$\subseteq$" on joukkojen välinen osittaisjärjestys
  - "mahtuu sisään" on laatikoiden välinen osittaisjärjestys, jos reunat äärettömän ohuet



- osittaisjärjestys eroaa *täydestä järjestyksestä* (*total order*) vain siten, että osittaisjärjestyksessä ei vaadita, että $\forall\, a, b \in A\colon a \le b \vee b \le a$
  - "$\le$" on reaalilukujen välinen täysi järjestys
  - "$\subseteq$" ei ole joukkojen välinen täysi järjestys, koska $\neg(\{1\} \subseteq \{2\}) \wedge \neg(\{2\} \subseteq \{1\})$
  - "mahtuu sisään" ei ole laatikoiden täysi järjestys
- täyttä järjestystä kutsutaan myös nimellä *lineaarinen järjestys* (*linear order*)
- jos "$\le$" on A:n osittaisjärjestys, niin määrittelemme $\forall\, a, b \in A$:
  - $a < b \;\Leftrightarrow\; a \le b \wedge a \ne b$
  - $a \ge b \;\Leftrightarrow\; b \le a$
  - $a > b \;\Leftrightarrow\; b < a$
  - $a \not\le b \;\Leftrightarrow\; \neg(a \ge b)$ (**huom!** ei sama kuin $a < b$)
  - samoin muille yliviitatuille

Yleistetty ylärajafunktio

- olkoon A jokin joukko ja "$\le$" sen osittaisjärjestys
- pari (A, "$\le$") on *hyvin perustettu* (*well-founded*), jos ja vain jos ei ole olemassa ääretöntä jonoa $a_1, a_2, a_3, \ldots \in A$ siten, että $a_1 > a_2 > a_3 > \ldots$
- esimerkkejä
  - ($\boldsymbol{N}$, "$\le$") on hyvin perustettu: jos $a_1 = n$, niin laskevassa ketjussa $\le n+1$ alkiota
  - ($\boldsymbol{Q}^+$, "$\le$") ei ole hyvin perustettu:
    $1/1 > 1/2 > 1/3 > \ldots$
  - ($\boldsymbol{N} \times \boldsymbol{N}$, "$\le$") on hyvin perustettu, missä
    $(n_1, m_1) > (n_2, m_2) \;\Leftrightarrow$
    $n_1 > n_2 \vee (n_1 = n_2 \wedge m_1 > m_2)$
    kokeile huviksesi!

- ylärajafunktioksi kelpaa mikä tahansa tilan funktio, jonka arvolla on seuraavat ominaisuudet:
  - se kuuluu johonkin hyvin perustettuun joukkoon
  - se pienenee jokaisella kierroksella
- ylärajafunktion alkuarvon tunteminen ei välttämättä takaa, että kierrosten määrää pystytään arvioimaan
  - jos ylärajafunktion arvo $\in \boldsymbol{N} \times \boldsymbol{N}$, niin seuraava pätee millä tahansa n:
    $(1,0) > (0,n) > (0,n-1) > \ldots > (0,1) > (0,0)$
  - $\Rightarrow$ on olemassa mielivaltaisen pitkiä laskevia ketjuja, jotka alkavat $(1,0)$
  - kuitenkaan ääretön laskeva ketju ei ole mahdollinen

Invariantin ja ylärajafunktion valinnasta

- ei ole olemassa aina (tai edes useimmiten) toimivaa automaattista menetelmää invariantin ja ylärajafunktion valitsemiseksi
  $\Rightarrow$ valinnassa täytyy käyttää inhimillistä luovuutta
- invariantti heijastaa silmukan "perusideaa"

$\Rightarrow$ jos invariantin muodostaminen ei meinaa millään onnistua, niin
- ehkä ohjelmoija ei itsekään oikein ymmärrä, miksi hänen silmukkansa tulisi toimia
- $\Rightarrow$ silmukka on todennäköisesti väärin

- ylärajafunktio edustaa jotain suuretta, jolla mitattuna suoritus etenee koko ajan

- ylärajafunktion valinnassa on paljon vapauksia
  - pieneneminen kierroksella voi olla mitä vain, kunhan se on aina ainakin 1
  - alarajan ei tarvitse olla 0 — riittää, että se on jokin kokonaisluku ($-\infty$ ei kelpaa)
  - ylärajafunktion ei tarvitse tuottaa kokonaislukua, vaan mikä tahansa hyvin perustettu joukko kelpaa maalijoukoksi (jolloin alarajakaan ei ole kok.luku)

$\Rightarrow$ jos ylärajafunktion muodostaminen ei meinaa millään onnistua, niin
- ehkä ohjelmoija ei itsekään oikein ymmärrä, miksi hänen silmukkansa tulisi lopettaa
- $\Rightarrow$ silmukka on todennäköisesti väärin

Epädeterministinen silmukkalause

- ohjelmointikielten teoreetikot käyttävät usein epädeterminististä silmukkalausetta

  do $B_1 \to S_1$
  [] $B_2 \to S_2$
  …
  [] $B_n \to S_n$
  od

- toiminta
  - jokaisen kierroksen alkaessa jokaisen B_i täytyy olla laskettavissa
  - jos mikään B_i ei toteudu, tullaan ulos silmukasta
  - muussa tapauksessa valitaan **mielivaltaisesti** jokin haara siten, että sen B_i toteutuu, suoritetaan vastaava S_i, ja palataan silmukan alkuun

- tavalliset silmukkalauseet voidaan toteuttaa epädeterministisellä silmukkalauseella:
  while B **do** S **endwhile**
  on sama kuin
  do $B \to S$ **od**
- epädeterministinen silmukkalause voidaan toteuttaa tavallisella silmukkalauseella ja epädeterministisellä **if**-lauseella:
  do $B_1 \to S_1$ [] … [] $B_n \to S_n$ **od**
  on sama kuin
  while $B_1 \vee \ldots \vee B_n$ **do**
  if $B_1 \to S_1$ [] … [] $B_n \to S_n$ **fi**
  endwhile
- esimerkki:
  do $A[1] > A[2] \to$ vaihda($A[1]$, $A[2]$)
  [] $A[1] > A[3] \to$ vaihda($A[1]$, $A[3]$)
  [] $A[2] > A[3] \to$ vaihda($A[2]$, $A[3]$)
  od
  – jos lopettaa niin silloin $A[1] \le A[2] \le A[3]$
  – ylärajafunktio: sellaisten parien (i, j) määrä, joille $i < j$ mutta $A[i] > A[j]$ (miksi kelpaa?)
  $\Rightarrow$ kyllä lopettaa

$\Rightarrow$ silmukka järjestää taulukon

Harjoitustehtäviä

1. Mikä on wp(**while false do skip endwhile**, Q)? Entä wp(**while** B **do skip endwhile**, Q)?
2. Osoita seuraava ohjelma osittain oikeaksi ja pysähtyväksi. Invariantin tärkein osa on $x \cdot b^i = a^n$.

   { $n \ge 0$ }
   $i := n$; $b := a$; $x := 1$;
   while $i > 0$ **do**
   if i mod 2 = 1 **then** $x := b \cdot x$ **endif**;
   $i := i$ div 2; $b := b \cdot b$
   endwhile
   { $x = a^n$ }
3. (a) Olkoon $A[1\ldots n]$ jokin taulukko, ja
   väärä-järjestys$(i, j) \Leftrightarrow i < j \wedge A[i] > A[j]$
   kun $1 \le i \le n$ ja $1 \le j \le n$. Osoita, että $|V|$ on seuraavan ohjelman ylärajafunktio, jos
   $V = \{ (i, j) \in \{1, \ldots, n\}^2 \mid$ *väärä-järjestys*$(i, j) \}$.

   while $\exists\, i, j\ ;\ 1 \le i \le n \wedge 1 \le j \le n$: *väärä-järjestys*$(i, j)$
   do
   valitse i ja j siten, että *väärä-järjestys*(i, j);
   vaihda($A[i]$, $A[j]$)
   endwhile

   (b) Miten (a)-kohdan tulosta voidaan käyttää hyväksi erilaisten järjestämisalgoritmien oikeaksi osoittamisessa?
4. Käyttäen mallina luennoilla esitettyjä tavallisen silmukkalauseen invariantilta ja ylärajafunktiolta vaadittuja ominaisuuksia, luettele epädeterministisen silmukkalauseen invariantilta ja ylärajafunktiolta vaadittavat ominaisuudet.

5. Muotoile oikeaksi osoittamissäännöt tavalliselle **for**-silmukalle:

   for $i :=$ *ala* **to** *ylä* **do** S **endfor**

   Pätee:
   – S ei saa sijoittaa i:hin
   – *ala* ja *ylä* voidaan ajatella vakioiksi (jos ne ovat lausekkeita, niiden arvot lasketaan vain kerran, nimittäin silmukkaan tultaessa).
6. Merkitään x:n esiintymiskertojen määrää taulukossa $A[1\ldots n]$ symbolilla $\#(A, x)$.

(a) Alkio x on A:n *enemmistöalkio*, jos yli puolet A:n alkioista on arvoltaan x. Kirjoita predikaatti *en*(x, A), joka sanoo, että joko x on A:n enemmistöalkio, tai A:lla ei ole lainkaan enemmistöalkioita.

(b) Osoita seuraava ohjelma täysin oikeaksi. Vihje: jos e on koko A:n enemmistöalkio, niin jokaisen kierroksen alussa joko $x = e \wedge k \ge 2 \cdot \#(A[1\ldots i{-}1], e) {-} i {+} 1$, tai $x \ne e \wedge -k \ge 2 \cdot \#(A[1\ldots i{-}1], e) {-} i {+} 1$.

   ⟨ $n \ge 0$ ⟩
   $k := 0$;
   for $i := 1$ **to** n **do**
   if $k = 0$ **then** $x := A[i]$; $k := 1$
   elsif $A[i] = x$ **then** $k := k{+}1$
   else $k := k{-}1$
   endif
   endfor
   ⟨ *en*(x, A) ⟩

(c) Suunnittele ja osoita oikeaksi ohjelma, joka tuottaa A:n enemmistöalkion tai vastauksen "A:lla ei ole enemmistöalkiota" kahdella A:n selauksella.

3.4 Todistusesimerkkejä

Edellisissä kohdissa esiteltiin ohjelmien oikeaksi osoittamisen perustekniikat

- käytännön todistuksissa niitä voi soveltaa ja yhdistellä monin eri tavoin
- tässä kohdassa käydään läpi joukko esimerkkejä

Esimerkki: alkion poisto taulukosta järjestys säilyttäen

- ohjelman tehtävänä on poistaa alkio i taulukosta $A[1\ldots n]$ siirtämällä muita alkioita askel taaksepäin
- alkuperäinen ohjelma

  for $j := i$ **to** $n - 1$ **do**
  $A[j] := A[j{+}1]$
  endfor
  $n := n - 1$
- lopputilalta vaaditaan
  – i kiinteä, tällä kertaa n ei ole kiinteä
  – alkuosa ennallaan: $\forall\, k;\ 1 \le k < i$: $A[k] = A_0[k]$
  – loppuosa siirtynyt: $\forall\, k;\ i \le k \le n$: $A[k] = A_0[k{+}1]$
  – talukko pienentynyt: $n = n_0 - 1$
  – taulukon alkutilan nimeksi annettu A_0 ja kooksi n_0
- tulee pitkiä, vaikealukuisia kaavoja

$\Rightarrow$ määritellään apupredikaatteja:
– $Enn(a,b) \quad :\Leftrightarrow\ \forall\, k;\ a \le k < b$: $A[k] = A_0[k]$
– $Siirt(a,b) \quad :\Leftrightarrow\ \forall\, k;\ a \le k < b$: $A[k] = A_0[k{+}1]$
– selkeyttävä käytäntö: väli ilmoitetaan alusta lopun perään

- lopputilaa koskeva vaatimus muuttuu muotoon
  $n = n_0 - 1 \wedge Enn(1, i) \wedge Siirt(i, n{+}1)$

- spesifikaatio

$\langle\ A[1\ldots n] = A_0\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  $A[j] := A[j+1]$
endfor
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge Enn(1,i) \wedge Siirt(i,n+1)\ \rangle$

- kohta huomaamme, että tässä on vika (mikä?)

Ratkaisun alku

- muutetaan silmukka **while**-silmukaksi

$\langle\ A = A_0\ \rangle$
$j := i$
while $j < n$ **do**
  $A[j] := A[j+1]$
  $j := j + 1$
endwhile
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge Enn(1,i) \wedge Siirt(i,n+1)\ \rangle$

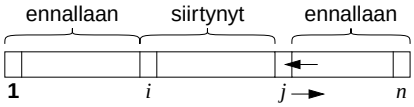
- lasketaan helpot sijoituslauseet

$\Rightarrow$ tehtävä pelkistyy silmukan todistamiseksi

$\langle\ A = A_0\ \rangle$
$j := i$
$\langle\ A = A_0 \wedge j = i\ \rangle$
while $j < n$ **do**
  $A[j] := A[j+1]$
  $j := j + 1$
endwhile
$\langle\ n = n_0 \wedge Enn(1,i) \wedge Siirt(i,n)\ \rangle$
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge Enn(1,i) \wedge Siirt(i,n+1)\ \rangle$

Silmukkainvariantin suunnittelu

- ohjelman toimintaa voi havainnollistaa kuvana



- laaditaan tältä pohjalta silmukkainvariantti
  - n ei muuttunut: $n = n_0$
  - alkuosa ennallaan: $Enn(1,i)$
  - keskiosa siirtynyt: $Siirt(i, j)$
  - loppuosa ennallaan: $Enn(j+1,n+1)$

$\Rightarrow$ $Inv_1 :\Leftrightarrow n = n_0 \wedge Enn(1,i) \wedge Siirt(i, j) \wedge Enn(j+1,n+1)$

Silmukkainvariantin todistus

2. silmukkainvariantin on oltava voimassa silmukan alussa

- sijoitetaan $j = i$, saadaan tavoitteeksi
  $n = n_0 \wedge Enn(1,i) \wedge Siirt(i, i) \wedge Enn(i+1,n+1)$
  $\Leftrightarrow$ $n = n_0 \wedge Enn(1,i) \wedge Enn(i+1,n+1)$
- $A = A_0$ takaa $n = n_0 \wedge Enn(1,n+1)$

$\Rightarrow$ kysymys: seuraako näistä $Enn(1,i)$ ja $Enn(i+1,n+1)$?

- vastaus: vain jos $i \le n + 1$ ja $i + 1 \ge 1$

$\Rightarrow$ ohjelman spesifikaatioon pitää lisätä alkuehto $0 \le i \le n + 1$

$\langle\ A = A_0 \wedge 0 \le i \le n + 1\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  …

- mitä ohjelma tekee jos alussa $i > n + 1$?
- onko se oikein?

4. silmukkainvariantin ja poistumisehdon pitää taata silmukan jälkeinen ehto

- poistumisehto $j \ge n$

$\Rightarrow$ saadaan
$j \ge n \wedge n = n_0 \wedge Enn(1,i) \wedge Siirt(i, j) \wedge Enn(j+1,n+1)$

- tarvittaisiin $n = n_0 \wedge Enn(1,i) \wedge Siirt(i,n)$

$\Rightarrow$ kannattaa verrata kaavoja $Siirt(i, j)$ ja $Siirt(i,n)$

- $j \ge n \wedge Siirt(i, j)$ kyllä takaa $Siirt(i,n)$, mutta puhuu taulukon ulkopuolisista alkioista kun $j > n$

$\Rightarrow$ todennäköisesti lopussa tulee päteä $j = n$

- kysymys: eikö silmukka takaa, että lopussa $j = n$?
  vastaus: vain jos alussa $j \le n$

$\Rightarrow$ ohjelman spesifikaatioon pitää lisätä alkuehto $i \le n$

$\langle\ A = A_0 \wedge 0 \le i \le n\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  …

- mitä ohjelma tekee jos alussa $i = n + 1$?
- onko se oikein?

- nyt nähdään helposti, että jos silmukan loppuun päästään, siellä pätee $j = n$

$\langle\ i \le n\ \rangle$
$j := i$
$\langle\ j \le n\ \rangle$
while $j < n$ **do**
  $\{\ j < n\ \}\ A[j] := A[j+1];\ j := j + 1\ \{\ j \le n\ \}$
endwhile
$\{\ j \le n \wedge j \ge n\ \}\ \{\ j = n\ \}$

- nyt poistumisehdosta ja invariantista saadaan
  $j = n \wedge n = n_0 \wedge Enn(1,i) \wedge Siirt(i, j) \wedge Enn(j+1,n+1)$
  $\Rightarrow$ $n = n_0 \wedge Enn(1,i) \wedge Siirt(i,n)$

- spesifikaation täydennys ei riko osan 2 todistusta
  - todistettiin alkup. alkuehto $\Rightarrow \ldots \Rightarrow Inv$
  - toki silloin alkup. alkuehto $\wedge\ i \le n \Rightarrow \ldots \Rightarrow Inv$

3. invariantin ja silmukkaehdon tulee taata invariantti silmukan lopussa

- lasketaan heikoin esiehto
  $wp(\ A[j] := A[j+1];\ j := j + 1,\ Inv\ )$
- $Inv_1 \Leftrightarrow n = n_0 \wedge Enn(1,i) \wedge Siirt(i, j) \wedge Enn(j+1,n+1)$
  $wp(\ j := j + 1,\ Inv\ ) \Leftrightarrow$
  $n = n_0 \wedge Enn(1,i) \wedge Siirt(i, j+1) \wedge Enn(j+2,n+1)$
  $wp(\ A[j] := A[j+1];\ j := j + 1,\ Inv\ ) \Leftrightarrow$??
- *wp* edellyttää, että indeksoinnit ovat laillisia
  $\Rightarrow$ oltava $1 \le j \le n$ ja $1 \le j + 1 \le n$ eli $1 \le j < n$
  - $j < n$ pätee
  - $1 \le j$ tarvitaan $\Rightarrow \ldots$
  $\Rightarrow$ invariantiin tulee lisätä $1 \le j$ ja alkuehtoon $1 \le i$

$\Rightarrow$ uusi invariantti $Inv_2 :\Leftrightarrow$
$n = n_0 \wedge 1 \le j \le n \wedge Enn(1,i) \wedge Siirt(i, j) \wedge Enn(j+1,n+1)$

- invariantin vahvennus ei riko osan 4 todistusta
- osan 2 todistus meni uusiksi, mutta on helppo

- nyt on siis laskettava
  $wp(\ A[j] := A[j+1];\quad n = n_0 \wedge 1 \le j+1 \le n$
  $\wedge\ Enn(1,i) \wedge Siirt(i, j+1) \wedge Enn(j+2,n+1)\ )$
- *Enn* ja *Siirt* puhuvat *A*:sta
  $\Rightarrow$ tarvitaan tieto mihin niistä "$A[j] := \ldots$" vaikuttaa
  - vaikuttaa osaan $Enn(1,i)$, joss $1 \le j < i$
  - vaikuttaa osaan $Siirt(i, j+1)$, joss $i \le j$
  - ei vaikuta osaan $Enn(j+2,n+1)$

- on helppo nähdä, että koko ajan pätee $i \le j$
  - silmukan alussa $i = j$
  - i ei muutu, j kasvaa
  - (tämän voisi todistaa lisäämällä invarianttiin $i \le j$)

$\Rightarrow$ $wp(\ A[j] := A[j+1];\ j := j + 1,\ Inv_2\ ) \Leftrightarrow$
$n = n_0 \wedge 1 \le j+1 \le n \wedge Enn(1,i) \wedge X \wedge Enn(j+2,n+1)$
missä
$X \Leftrightarrow Siirt(i, j) \wedge A[j+1] = A_0[j+1]$

- $Inv_2 \wedge j < n$ eli $\quad j < n \wedge n = n_0 \wedge 1 \le j \le n \wedge Enn(1,i)$
  $\wedge \quad Siirt(i, j) \wedge Enn(j+1,n+1)$
  takaa
  - $n = n_0$ $\quad (\Leftarrow n = n_0\ )$
  - $1 \le j+1 \le n$ $\quad (\Leftarrow j < n \wedge 1 \le j\ )$
  - $Enn(1,i)$ $\quad (\Leftarrow Enn(1,i)\ )$
  - $Siirt(i, j)$ $\quad (\Leftarrow Siirt(i, j)\ )$
  - $A[j+1] = A_0[j+1]$ $\quad (\Leftarrow Enn(j+1,n+1) \wedge j < n\ )$
  - $Enn(j+2,n+1)$ $\quad (\Leftarrow Enn(j+1,n+1)\ )$

$\Rightarrow$ ·/·

Pysähtymisen todistus

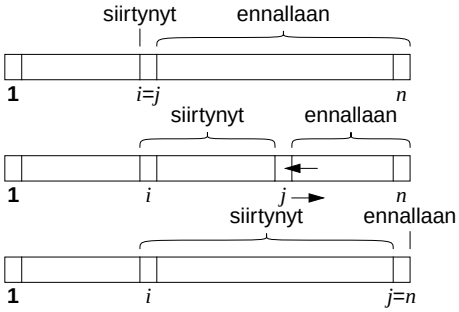
- ohjelman ainoa silmukka on aito **for**-silmukka
  $\Rightarrow$ pysähtyminen on automaattisesti taattu
- ylärajafunktiona voisi käyttää $n - j$

Nyt on todistettu

$\langle\ A[1\ldots n] = A_0 \wedge 1 \le i \le n\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  $A[j] := A[j+1]$
endfor
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge Enn(1,i) \wedge Siirt(i,n+1)\ \rangle$

Havaintoja

- paljastui puuttuva alkuehto $1 \le i \le n$
- tosiasiallinen invariantti oli
  $n = n_0 \wedge 1 \le i \le j \le n \wedge Enn(1,i)$
  $\wedge \quad Siirt(i, j) \wedge Enn(j+1,n+1)$
  - osa $1 \le i \le j$ käsiteltiin erikseen
  - koko ylärivin olisi voinut käsitellä erikseen
- invariantin ydin $Siirt(i, j) \wedge Enn(j+1,n+1)$ oli muotoa
  $I_1 \wedge I_2$
  missä
  - silmukan alussa I_1 puhuu tyhjästä välistä ja silmukan esiehto takaa I_2:n
  - silmukan aikana I_1:n väli kasvaa ja I_2:n kutistuu
  - silmukan lopussa I_2 puhuu tyhjästä välistä ja I_1 takaa silmukan jälkeisen tilapredikaatin



- tämä on varsin tavallista
- osa $Enn(j+1,n+1)$ olisi voinut olla $Enn(j,n+1)$
  - laskut olisivat menneet silti läpi
  - makuasia, kumpi valitaan

Esimerkki: puolitushaku

- olemme jo osoittaneet seuraavan:

$\langle$ **true** $\rangle$
$a := 1;\ y := n+1;$
$\langle\ a = 1 \wedge y = n+1\ \rangle$
while $a < y$ **do**
  $v := (a+y)$ div 2;
  if $A[v] < avain$ **then** $a := v+1$
  else $y := v$
  endif
endwhile
$\langle\ 1 \le a = y \le n+1\ \rangle$

- puolitushaun koko spesifikaatio on:

$\langle\ \forall\ i\ ;\ 1 \le i \le n-1 \colon A[i] \le A[i+1]\ \rangle$
Puolitushaku
$\langle\ (a = n+1 \vee A[a] \ge avain) \wedge$
$(a = 1 \vee A[a-1] < avain)\ \rangle$

- (miksi tässä ei vaadita $1 \le a = y \le n+1$?)
- osoitetaan ensiksi, että lopussa
  $a = 1 \vee A[a-1] < avain$
- tiedämme jo, että lopussa $1 \le a \le n+1$
- on helppo tarkistaa suoraan koodista, että jos lausetta $a := v+1$ ei suoritettu kertaakaan, niin lopussa $a = 1$
- jos lause suoritettiin, niin viimeiselle suoritukselle pätee (miksi?)
  $\langle\ A[v] < avain\ \rangle\ \ a := v+1$
- symbolinen suoritus antaa
  $\langle\ A[v] < avain\ \rangle\ \ a := v+1\ \ \langle\ A[a-1] < avain\ \rangle$

- koska tämän jälkeen a:n arvo ei muutu, ja A ja $avain$ ovat kiinteitä, lopussa pätee
  $A[a-1] < avain$

$\Rightarrow$ on osoitettu, että lopussa $a = 1 \vee A[a-1] < avain$

- vastaavasti tarkastelemalla y:hyn kohdistuvia sijoituslauseita voidaan päätellä, että lopussa
  $y = n+1 \vee A[y] \ge avain$
- koska tiedämme vanhastaan, että lopussa $a = y$, saamme

$\langle$ **true** $\rangle$
Puolitushaku
$\langle\ (a = n+1 \vee A[a] \ge avain) \wedge$
$(a = 1 \vee A[a-1] < avain)\ \rangle$

Puolitushaun todistus ei käyttänyt mitenkään hyväkseen esiehtoa $\forall\ i\ ;\ 1 \le i \le n-1 \colon A[i] \le A[i+1]$!

$\Rightarrow$ **kysymys:** onko todistuksessa virhe?

- **vastaus:** ei; ohjelma todellakin takaa, että lopussa $a = n+1 \vee A[a] \ge avain$ jne. vaikka taulukko ei olisi aluksi järjestyksessä
- **kysymys:** mutta eihän puolitushaku edes toimi, ellei taulukko ole järjestyksessä!
- **vastaus:** katsotaanpa …
- puolitushaun varsinainen tehtävä on löytää $avain$ taulukosta
  - jos $avain$ ei ole taulukossa, puolitushaku ei tietenkään voi löytää sitä
  - $\Rightarrow$ puolitushaun toimimattomuus voi ilmetä vain siten, että $avain$ ei löydy, vaikka se on taulukossa
- tarkistamme, päteekö

$\langle\ \exists\ x\ ;\ 1 \le x \le n\colon A[x] = avain\ \rangle$
Puolitushaku
$\langle\ A[a] = avain\ \rangle$

– ei: A = [1, 0] ja $avain$ = 0 tuottaa a = 1

- entä jos lisätään oletus: A on järjestyksessä?
  – siis $\forall\ i\ ;\ 1 \le i \le n-1\colon A[i] \le A[i+1]$
  – seuraava muoto on käytännössä kätevämpi:
    $\forall\ i, j\ ;\ 1 \le i \le j \le n\colon A[i] \le A[j]$
- koska A ja $avain$ ovat kiinteitä, lopussa pätee
  $\forall\ i, j\ ;\ 1 \le i \le j \le n\colon A[i] \le A[j] \wedge$
  $\exists\ x\ ;\ 1 \le x \le n\colon A[x] = avain \wedge$
  $(a = n+1 \vee A[a] \ge avain) \wedge (a = 1 \vee A[a-1] < avain)$
- kannattaako yrittää todistaa, että lopussa $a = x$?
- koska lopussa $a = 1 \vee A[a-1] < avain$ ja A on järjestyksessä, niin $a \le x$
  $\Rightarrow$ $A[a] \ge avain$ (koska $a = n+1$ ei käy koska $x \le n$)
  ja $A[a] \le A[x] = avain$ (koska $a \le x$)
  $\Rightarrow$ $A[a] = avain$
- järjestysoletus tarvitaan siis takaamaan, että jos $avain$ on taulukossa, ainakin yksi $avain$ on alkuperäisen loppuehdon mukaisessa kohdassa
- jos A ei ole järjestyksessä, puolitushaku kylläkin etsii alkuperäisen loppuehdon mukaisen kohdan, mutta yksikään $avain$ ei välttämättä ole siellä

Johtopäätöksiä

- kaikkea ei osoitettu edellisten kohtien tekniikoilla
  – pysähtyminen ja a:n jääminen sallitulle välille osoitettiin niillä
  – a:n osuminen “oikeaan rakoon” pääteltiin tämän jälkeen suoraan koodista
  – avaimen löytyminen pääteltiin erikseen loppuehdon ja järjestysoletuksen perusteella koodia katsomatta
- kaiken olisi voinut osoittaa edellisten kohtien tekniikoilla, mutta se olisi ollut vaikeampaa
  – usein kannattaa todistaa “helpot ensin” ja koota lopullinen tavoite “pikkuhiljaa”
- puolitushaulle jouduttiin muotoilemaan monimutkainen loppuehto, koska sen tulee “toimia” silloinkin, kun etsitty ei ole taulukossa
- puolitushakuohjelma yllättäen takaa loppuehdon silloinkin, kun taulukko ei ole järjestyksessä
- järjestysoletus tarvitaan siksi, että ilman sitä loppuehto ei takaa avaimen löytymistä
  – ei ole yllättävää, koska se oli mielessä loppuehtoa muotoillessa
- hyvin suunniteltu ohjelma takaa joskus (usein?) yllättäviä ominaisuuksia!
- usein pätee
  – hyvin suunnitellulla ohjelmalla on loogisesti kauniita ominaisuuksia
  – huonosti suunniteltu on loogisesti sotkuinen

Vertailun vuoksi todistus kokonaan invarianteilla ja ylärajafunktioilla

- alkuperäisen loppuehdon osalta
  – eipä juuri hankaloidu!

$\langle$ **true** $\rangle$
$a := 1;\ y := n+1;$
$\langle\ a = 1 \wedge y = n+1\ \rangle$
$\langle$ **inv:** $1 \le a \le y \le n+1 \wedge (y = n+1 \vee A[y] \ge avain) \wedge$
$(a = 1 \vee A[a-1] < avain)\ \rangle$
$\langle$ **yr**: $y - a\ \rangle$
while $a < y$ **do**
  $\langle\ 1 \le a < y \le n+1 \wedge (y = n+1 \vee A[y] \ge avain) \wedge$
  $(a = 1 \vee A[a-1] < avain) \wedge y - a = yr_0 > 0\ \rangle$
  $v := (a+y)$ div 2;
  $\langle\ 1 \le a \le v < y \le n+1 \wedge (y = n+1 \vee A[y] \ge avain)$
  $\wedge (a = 1 \vee A[a-1] < avain) \wedge y - a = yr_0\ \rangle$
  if $A[v] < avain$ **then**
    $a := v+1$
    $\langle\ 1 < a \le y \le n+1 \wedge (y = n+1 \vee A[y] \ge avain)$
    $\wedge A[a-1] < avain \wedge y - a < yr_0\ \rangle$
  else
    $y := v$
    $\langle\ 1 \le a \le y \le n \wedge A[y] \ge avain$
    $\wedge (a = 1 \vee A[a-1] < avain) \wedge y - a < yr_0\ \rangle$
  endif
  $\langle\ 1 \le a \le y \le n+1 \wedge (y = n+1 \vee A[y] \ge avain)$
  $\wedge (a = 1 \vee A[a-1] < avain) \wedge y - a < yr_0\ \rangle$
endwhile
$\langle\ 1 \le a = y \le n+1 \wedge$
$(a = n+1 \vee A[a] \ge avain) \wedge$
$(a = 1 \vee A[a-1] < avain)\ \rangle$

- $avain$ löytyy, jos on taulukossa
  – A kiinteä
  $\Rightarrow$ taulukko on koko ajan järjestyksessä
  $\Rightarrow$ emme toista järjestystietoa joka predikaatissa, vaikka käytämme päättelyssä

$\langle\ \exists\ x\ ;\ 1 \le x \le n\colon A[x] = avain \wedge$
$\forall\ i, j\ ;\ 1 \le i \le j \le n\colon A[i] \le A[j]\ \rangle$
$a := 1;\ y := n+1;$
$\langle\ a = 1 \wedge y = n+1 \wedge \exists\ x\ ;\ 1 \le x \le n\colon A[x] = avain\ \rangle$
$\langle$ **inv:** $\exists\ x\colon 1 \le a \le x \le y \le n+1 \wedge x \le n \wedge A[x] = avain\ \rangle$
$\langle$ **yr**: $y - a\ \rangle$
while $a < y$ **do**
  $\langle\ \exists\ x\colon 1 \le a \le x \le y \le n+1 \wedge x \le n \wedge A[x] = avain \wedge$
  $a < y \wedge y - a = yr_0 > 0\ \rangle$
  $v := (a+y)$ div 2; (* nyt lisäksi $a \le v < y$ *)
  if $A[v] < avain$ **then**
    $\langle\ \exists\ x\colon 1 \le v < x \le y \le n+1 \wedge x \le n \wedge$
    $A[x] = avain \wedge y - v \le yr_0\ \rangle$
    $a := v+1$
  else
    $\langle\ (\ A[v] = avain \wedge 1 \le a \le v \le n$
    $\vee A[v] > avain \wedge \exists\ x\colon 1 \le a \le x < v \le n \wedge$
    $A[x] = avain$
    $)\ \wedge v - a < yr_0\ \rangle$
    $\langle\ \exists\ x\colon 1 \le a \le x \le v \le n \wedge A[x] = avain \wedge$
    $v - a < yr_0\ \rangle$
    $y := v$
  endif
  $\langle\ \exists\ x\colon 1 \le a \le x \le y \le n+1 \wedge x \le n \wedge A[x] = avain \wedge$
  $y - a < yr_0\ \rangle$
endwhile
$\langle\ \exists\ x\colon 1 \le a \le x \le y \le n+1 \wedge x \le n \wedge A[x] = avain \wedge a \ge y\ \rangle$
$\langle\ 1 \le a \le n \wedge A[a] = avain\ \rangle$

- todistus oli odottamattoman työläs!
- ensiksi, alustuksen $y := n+1$ vuoksi jouduttiin usein kirjoittamaan erikseen $x \le n$
  - tämä todistus olisi helpottunut, jos olisi alustettu $y := n$
- toiseksi, jos $A[v] = avain$, niin
  - järjestysoletuksen perusteella ei voi päätellä, että $x \le v$
  - silloin voidaan valita v uudeksi x:ksi
  - $\Rightarrow$ **else**-haaran päättely piti jakaa kahtia
- todistettaessa ensin alkuperäinen loppuehto ja sitten sen avulla avaimen löytyminen vältettiin molemmat vaikeudet
  - avaimen sijaintipaikasta ei tarvinnut puhua ohjelmakoodia todistettaessa

$\Rightarrow$ jopa avaimen löytyminen oli helpompi todistaa alkuperäisen loppuehdon avulla kuin suoraan

$\Rightarrow$ alkuperäinen loppuehto tiivistää erinomaisesti puolitushaun “ytimen”!

Toinen esimerkki: COUNTING-SORT: tehtävän asettelu

- järjestää taulukon laskemalla, kuinka monta kappaletta kutakin avainta on
- avainten tulee olla tunnetulla välillä 0, …, M
- syöte: $A[1\ldots n]$ ja M kiinteitä
- tulos: $B[1\ldots n]$
- olennainen lisäpiirre: *vakaa*
  - ts. ei vaihda kahden tietueen keskinäistä järjestystä, jos niillä on sama avain

- vakaudella on merkitystä vain, jos taulukon alkiossa on muutakin kuin avain

$\Rightarrow$ oletamme alkiolle seuraavan rakenteen:
- $A[i].key$ avain
- $A[i]$ koko alkio

- algoritmi

```
COUNTING-SORT( A[1…n], B[1…n], M )
{ ∀ i ; 1 ≤ i ≤ n: 0 ≤ A[i].key ≤ M }
for k := 0 to M do C[k] := 0 endfor
for i := 1 to n do
    C[A[i].key] := C[A[i].key] + 1
endfor
(* nyt C[k] tietää kuinka monen alkion avain = k *)
for k := 1 to M do C[k] := C[k] + C[k–1] endfor
(* nyt C[k] tietää kuinka monen alkion avain ≤ k *)
for i := n downto 1 do
    B[C[A[i].key]] := A[i];
    C[A[i].key] := C[A[i].key] – 1
endfor
```

- muuttujien arvoalueet
  - k: ainakin 0, …, M
  - i: ainakin 1, …, n
  - $C[0\ldots M]$: ainakin 0, …, n
  - $B[1\ldots n]$: ainakin sama kuin $A[1\ldots n]$

COUNTING-SORTin spesifikaatio

- vanha tuttu $järjestyksessä(B) \wedge samat\text{-}alk(A, B)$ ei sisällä vakauden vaatimusta
- vakausvaatimuksen esittämiseksi otamme käyttöön funktion f, joka kertoo, mihin kohti taulukkoa B taulukon A alkiot joutuvat
  - siis lopussa oltava
    $\forall\, i\ ;\ 1 \le i \le n\colon\ 1 \le f(i) \le n \wedge B[f(i)] = A[i]$
- f:n tulee olla bijektio:
  $\forall\, i, j\ ;\ 1 \le i < j \le n\colon\ f(i) \ne f(j)$
- vakausvaatimus
  $\forall\, i, j\ ;\ 1 \le i < j \le n\colon\ (\ A[i].key = A[j].key \rightarrow f(i) < f(j)\ )$

COUNTING-SORTin todistuksen alkuvaiheet

- A kiinteä
  $\Rightarrow$ ehto $\forall\, i\ ;\ 1 \le i \le n\colon\ 0 \le A[i].key \le M$ on koko ajan voimassa
  $\Rightarrow$ emme toista sitä joka predikaatissa
- 1. **for**-silmukan tehtävänä on nollata C

  $\langle\ \forall\, i\ ;\ 1 \le i \le n\colon\ 0 \le A[i].key \le M\ \rangle$
  for $k := 0$ **to** M **do** $C[k] := 0$ **endfor**
  $\langle\ \forall\, h\ ;\ 0 \le h \le M\colon\ C[h] = 0\ \rangle$
  - sijoitusten laillisuus ilmeinen
  - invariantti: $I_1 \Leftrightarrow \forall\, h\ ;\ 0 \le h \le k-1\colon\ C[h] = 0$
  - $k = 0 \Rightarrow I_1$ ilmeinen $\cdot/\cdot$
  - $\langle\ I_1 \wedge k \le M\ \rangle\ C[k] := 0;\ k := k+1\ \langle\ I_1\ \rangle$, koska $\forall\, h\ ;\ 0 \le h \le k\colon\ C([k] \leftarrow 0)[h] = 0 \Leftrightarrow I_1 \wedge 0 = 0$
  - $k = M+1 \wedge I_1 \Rightarrow \forall\, h\ ;\ 0 \le h \le M\colon\ C[h] = 0$ $\cdot/\cdot$

- 2. **for**-silmukan tehtävänä on laskea, kuinka monesti eri alkiot esiintyvät A:ssa
  - invariantti
    $\forall\, h\ ;\ 0 \le h \le M\colon$
    $C[h] = |\{\ j \mid 1 \le j \le i-1 \wedge A[j].key = h\ \}|$
  - todistus harjoitustehtäväksi

    $\langle\ \forall\, h\ ;\ 0 \le h \le M\colon\ C[h] = 0\ \rangle$
    for $i := 1$ **to** n **do**
    $C[A[i].key] := C[A[i].key] + 1$
    endfor
    $\langle\ \forall\, h\ ;\ 0 \le h \le M\colon$
    $C[h] = |\{\ j \mid 1 \le j \le n \wedge A[j].key = h\ \}|\ \rangle$
- 3. **for**-silmukka laskee, kuinka moni A:n alkio $\le h$
  - todistus harjoitustehtäväksi

    $\langle\ \forall\, h\ ;\ 0 \le h \le M\colon$
    $C[h] = |\{\ j \mid 1 \le j \le n \wedge A[j].key = h\ \}|\ \rangle$
    for $k := 1$ **to** M **do** $C[k] := C[k] + C[k-1]$ **endfor**
    $\langle\ \forall\, h\ ;\ 0 \le h \le M\colon$
    $C[h] = |\{\ j \mid 1 \le j \le n \wedge A[j].key \le h\ \}|\ \rangle$

4. **for**-silmukan todistus

- tähän mennessä ei ole tapahtunut muuta kuin että C:hen on laskettu tietoa avainten määristä
- 4. **for**-silmukan tehtävänä on tämän tiedon perusteella kopioida alkiot A:sta B:hen
- spesifikaation kannalta alkioiden vanhojen ja uusien paikkojen välinen riippuvuus f on olennainen

$\Rightarrow$ ensin kannattaa tunnistaa f, ja todistaa, että ohjelma laskee sen oikein

$\langle\ \forall\ h\ ;\ 0 \le h \le M$:
$C[h] = |\{\ j \mid 1 \le j \le n \wedge A[j].key \le h\ \}|\ \rangle$
for $i := n$ **downto** 1 **do**
$B[C[A[i].key]] := A[i]$;
$C[A[i].key] := C[A[i].key] - 1$
endfor
$\langle\ \forall\ j\ ;\ 1 \le j \le n$: $B[f(j)] = A[j]\ \rangle$

missä

$f(x) = |\{\ k \mid 1 \le k \le n \wedge A[k].key < A[x].key\ \}| +$
$|\{\ k \mid 1 \le k \le x \wedge A[k].key = A[x].key\ \}|$

- invariantti $I_4 \Leftrightarrow$
  $\forall\ h\ ;\ 0 \le h \le M$:
  $C[h] = |\{\ k \mid 1 \le k \le n \wedge A[k].key < h\ \}| +$
  $|\{\ k \mid 1 \le k \le i \wedge A[k].key = h\ \}| \quad \wedge$
  $\forall\ j\ ;\ i+1 \le j \le n$: $B[f(j)] = A[j]$
  - $P \Rightarrow I_4$ ilmeinen $\cdot/\cdot$
  - I_4:n säilyminen silmukassa todistetaan kohta
  - $I_4 \wedge i = 0 \Rightarrow Q$ ilmeinen $\cdot/\cdot$
- I_4:n C-osa säilyy, koska
  $\forall\ h\ ;\ 0 \le h \le M$:
  $C(\ [A[i].key] \leftarrow C[A[i].key] - 1\ )[h] =$
  $|\{\ k \mid 1 \le k \le n \wedge A[k].key < h\ \}| +$
  $|\{\ k \mid 1 \le k \le i - 1 \wedge A[k].key = h\ \}|$
  $\Leftrightarrow\ \forall\ h\ ;\ 0 \le h \le M$:
  $(\ h \ne A[i].key \rightarrow C[h] =$
  $|\{\ k \mid 1 \le k \le n \wedge A[k].key < h\ \}| +$
  $|\{\ k \mid 1 \le k \le i \wedge A[k].key = h\ \}|\ )$
  $\wedge\ (\ h = A[i].key \rightarrow C[A[i].key] - 1 = C[h] - 1 =$
  $|\{\ k \mid 1 \le k \le n \wedge A[k].key < A[i].key\ \}| +$
  $|\{\ k \mid 1 \le k \le i \wedge A[k].key = A[i].key\ \}| - 1\ )$
  $\Leftrightarrow$ C-osa

- I_4:n B-osan säilymisen osoittamiseksi on tarpeen osoittaa, että jos $i \ne j$, niin $f(i) \ne f(j)$
  - jos $A[i].key = A[j].key$ ja $i < j$, niin $f(j)$
    $= f(i) + |\{\ k \mid i+1 \le k \le j \wedge A[k].key = A[i].key\ \}|$
    $\ge f(i) + 1 > f(i)$
  - jos $A[i].key < A[j].key$, niin $f(j)$
    $> |\{\ k \mid 1 \le k \le n \wedge A[k].key < A[j].key\ \}|$
    $\ge |\{\ k \mid 1 \le k \le n \wedge A[k].key \le A[i].key\ \}| \ge f(i)$

  $\Rightarrow$ jos $A[i].key < A[j].key$ tai
  $A[i].key = A[j].key \wedge i < j$,
  niin $f(i) < f(j)$

  $\Rightarrow$ jos $i \ne j$, niin $f(i) \ne f(j)$
- nyt $wp(\ S, \forall\ j\ ;\ i+1 \le j \le n$: $B[f(j)] = A[j]\ )$
  $\Leftrightarrow\ \forall\ j\ ;\ i \le j \le n$: $B(\ [C[A[i].key]] \leftarrow A[i]\ )[f(j)] = A[j]$
  $\Leftrightarrow\ \forall\ j\ ;\ i \le j \le n$: $(\ f(j) = C[A[i].key] \rightarrow A[i] = A[j]\ ) \wedge$
  $(\ f(j) \ne C[A[i].key] \rightarrow B[f(j)] = A[j]\ )$
  (koska $C[A[i].key] = f(i)$)
  $\Leftrightarrow\ \forall\ j\ ;\ i \le j \le n$: $(\ f(j) = f(i) \wedge A[i] = A[j]\ ) \vee$
  $(\ f(j) \ne f(i) \wedge B[f(j)] = A[j]\ )$
  $\Leftrightarrow\ A[i] = A[i] \wedge \forall\ j\ ;\ i+1 \le j \le n$: $B[f(j)] = A[j]$,
  joten B-osa säilyy
- sijoitusten laillisuuden todistamiseksi on osoitettava, että $\forall\ i\ ;\ 1 \le i \le n$: $1 \le f(i) \le n$
  - $1 \le |\{\ k \mid 1 \le k \le i \wedge A[k].key = A[i].key\ \}| \le f(i)$
    $\le |\{\ k \mid 1 \le k \le n \wedge A[k].key \le A[i].key\ \}| \le n \quad \cdot/\cdot$

COUNTING-SORTin todistuksen loppuvaiheet

- vakausvaatimus tulikin jo todistettua (milloin?)
  $\forall\ i, j\ ;\ 1 \le i \le n \wedge 1 \le j \le n$:
  $(\ A[i].key = A[j].key \wedge i < j \rightarrow f(i) < f(j)\ )$

- *järjestyksessä*(B):n ja *samat-alk*(A, B):n todistamiseksi on tärkeää huomata, että
  - taulukkoon B on kirjoitettu n kertaa
  - joka kerta kirjoitettiin eri paikkaan
    (ts. $i \ne j \rightarrow f(i) \ne f(j)$)

  $\Rightarrow$ jokaiseen B:n alkioon on kirjoitettu

  $\Rightarrow\ \forall\ i\ ;\ 1 \le i \le n$: $\exists\ k\ ;\ 1 \le k \le n$: $i = f(k)$
- olkoot $1 \le i < j \le n$, ja olkoot k ja l valittu siten, että $i = f(k)$ ja $j = f(l)$
  - jos $A[l].key < A[k].key$, niin $j = f(l) < f(k) = i$

  $\Rightarrow$ $B[i].key = B[\ f(k)].key = A[k].key \le A[l].key = B[\ f(l)].key = B[j].key$

$\Rightarrow\ \forall\ i, j\ ;\ 1 \le i \le j \le n$: $B[i].key \le B[j].key$
- ts. *järjestyksessä*(B)
- jos $1 \le i \le n$, niin *määrä*($A[i]$, B)
  $= |\{\ k \mid 1 \le k \le n \wedge B[k] = A[i]\ \}|$
  $= |\{\ k \mid 1 \le k \le n \wedge B[\ f(k)] = A[i]\ \}|$ (miksi?)
  $= |\{\ k \mid 1 \le k \le n \wedge A[k] = A[i]\ \}|$ = *määrä*($A[i]$, A)
  $\Rightarrow$ *samat-alk*(A, B) $\cdot/\cdot$

Kolmas esimerkki: Fermat'n suuren lauseen testiohjelma: tehtävän asettelu

- seuraava kysymys on kuuluisa
  - oli auki 1600-luvulta, kunnes ratkesi 1995

    Onko olemassa positiivisia kokonaislukuja x, y, z ja n siten, että $n \ge 3$ ja $x^n + y^n = z^n$?

- seuraavan ohjelman tehtävänä on kokeilla kaikki mahdolliset x, y, z ja n kunnes vaaditut löytyvät

```
x := 1; y := 1; z := 1; n := 3;
while x^n + y^n ≠ z^n do
    if y > 1 then x := x+1; y := y–1
    elsif z > 1 then z := z–1; y := x+1; x := 1
    elsif n > 3 then n := n–1; z := x+1; x := 1
    else n := x+3; x := 1
    endif
endwhile
{ x ≥ 1 ∧ y ≥ 1 ∧ z ≥ 1 ∧ n ≥ 3 ∧ x^n + y^n = z^n }
```

$\Rightarrow$ jos ohjelma toimii oikein, **meidän** ei kannata edes yrittää selvittää, pysähtyykö se
- (Andrew Wiles:in tuloksesta seuraa: ei pysähdy)

- sen sijaan on mahdollista ja tarpeellista tarkistaa
  - kokeileeko ohjelma vain laillisia x, y, z ja n? (muutoinhan se saattaisi pysähtyä ennen aikojaan, esim. $x = 3$, $y = 4$, $z = 5$ ja $n = 2$)
  - jos haluttuja x, y, z ja n ei ole, kokeileeko ohjelma kaikki lailliset mahdollisuudet?

$\Rightarrow$ tämän kohdan viimeinen esimerkki

Kokeileeko ohjelma vain laillisia x, y, z ja n?

- merkitään ("L" ~ "laillinen")
  $L \Leftrightarrow x \ge 1 \wedge y \ge 1 \wedge z \ge 1 \wedge n \ge 3$
- osoitettava, että silmukan jokaisen kierroksen alussa L pätee

$\Rightarrow$ tarvitaan invariantti I siten, että
$I \Rightarrow L$

- seuraavat on helppo tarkistaa:
  - alustus saattaa L:n voimaan
  - $\langle\, L \wedge y > 1 \,\rangle\ x := x+1; y := y-1\ \langle\, L \,\rangle$
  - $\langle\, L \wedge z > 1 \wedge \dots \,\rangle\ z := z-1; y := x+1; x := 1\ \langle\, L \,\rangle$
  - $\langle\, L \wedge n > 3 \wedge \dots \,\rangle\ n := n-1; z := x+1; x := 1\ \langle\, L \,\rangle$
  - $\langle\, L \wedge \dots \,\rangle\ n := x+3; x := 1\ \langle\, L \,\rangle$

  joten voimme valita $I \Leftrightarrow L$

$\Rightarrow$ ohjelma kokeilee vain laillisia x, y, z ja n

Jos haluttuja x, y, z ja n ei ole, kokeileeko ohjelma kaikki lailliset mahdollisuudet?

- tämä on aivan eri tyyppinen tehtävä kuin mitä olemme tähän mennessä ratkaisseet
- silti meillä on riittävät keinot!
  - tarkastellaan tiloja (x, y, z, n) **while**-silmukan peräkkäisten kierrosten alussa
  - merkitään tavoitetilaa Q_0:lla
  - muodostetaan jono laillisia tiloja Q_1, Q_2, … siten, että Q_j takaa, että seuraavan kierroksen jälkeen pätee Q_{j-1} (kyllä, "$j-1$"!)
  - osoitetaan ylärajafunktiotekniikalla, että jono päättyy tilaan, jossa $x = y = z = 1 \wedge n = 3$
  - $\Rightarrow$ alkutilasta päästään tavoitetilaan äärellisen monella silmukan kierroksella, jollei sitä ennen jossain tilassa päde $x^n + y^n = z^n$
- merkitään
  - S on **while**-silmukan vartalo
  - $Q \Leftrightarrow x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge n = n_0$
- tavoitetilaksi Q_0 kelpaavat vain lailliset tilat

  $\Rightarrow$ määrittelemme $Q_0 \Leftrightarrow L \wedge Q$

- $\langle\, Q_j \,\rangle\, S\, \langle\, Q_{j-1} \,\rangle$ pätee jos ja vain jos

  $Q_j \Rightarrow wp(S, Q_{j-1})$

  $\Rightarrow$ valitsemme $Q_j \Leftrightarrow L \wedge wp(S, Q_{j-1})$
- merkitään $x_j = x$:n arvo tilassa Q_j, ja vastaavasti y_j, z_j ja n_j
- $Q_j \Leftrightarrow L \wedge wp(S, Q_{j-1}) \Leftrightarrow$

  $x_j \geq 1 \wedge y_j \geq 1 \wedge z_j \geq 1 \wedge n_j \geq 3 \wedge ($

  $\quad y_j > 1 \wedge$

  $\quad x_j + 1 = x_{j-1} \wedge y_j - 1 = y_{j-1} \wedge z_j = z_{j-1} \wedge n_j = n_{j-1}$

  $\vee\ y_j \leq 1 \wedge z_j > 1 \wedge$

  $\quad 1 = x_{j-1} \wedge x_j + 1 = y_{j-1} \wedge z_j - 1 = z_{j-1} \wedge n_j = n_{j-1}$

  $\vee\ y_j \leq 1 \wedge z_j \leq 1 \wedge n_j > 3 \wedge$

  $\quad 1 = x_{j-1} \wedge y_j = y_{j-1} \wedge x_j + 1 = z_{j-1} \wedge n_j - 1 = n_{j-1}$

  $\vee\ y_j \leq 1 \wedge z_j \leq 1 \wedge n_j \leq 3 \wedge$

  $\quad 1 = x_{j-1} \wedge y_j = y_{j-1} \wedge z_j = z_{j-1} \wedge x_j + 3 = n_{j-1}$

  $)$

  $\Leftrightarrow$

  $\quad x_{j-1} > 1 \wedge y_{j-1} \geq 1 \wedge z_{j-1} \geq 1 \wedge n_{j-1} \geq 3 \wedge$

  $\quad x_j = x_{j-1} - 1 \wedge y_j = y_{j-1} + 1 \wedge z_j = z_{j-1} \wedge n_j = n_{j-1}$

  $\vee\ x_{j-1} = 1 \wedge y_{j-1} > 1 \wedge z_{j-1} \geq 1 \wedge n_{j-1} \geq 3 \wedge$

  $\quad x_j = y_{j-1} - 1 \wedge y_j = 1 \wedge z_j = z_{j-1} + 1 \wedge n_j = n_{j-1}$

  $\vee\ x_{j-1} = 1 \wedge y_{j-1} = 1 \wedge z_{j-1} > 1 \wedge n_{j-1} \geq 3 \wedge$

  $\quad x_j = z_{j-1} - 1 \wedge y_j = 1 \wedge z_j = 1 \wedge n_j = n_{j-1} + 1$

  $\vee\ x_{j-1} = 1 \wedge y_{j-1} = 1 \wedge z_{j-1} = 1 \wedge n_{j-1} > 3 \wedge$

  $\quad x_j = n_{j-1} - 3 \wedge y_j = 1 \wedge z_j = 1 \wedge n_j = 3$

- havaintoja
  - jos $(x_{j-1}, y_{j-1}, z_{j-1}, n_{j-1}) = (1, 1, 1, 3)$, niin $Q_j \Leftrightarrow$ **false**
  - muille laillisille tiloille $(x_{j-1}, y_{j-1}, z_{j-1}, n_{j-1})$ kaava määrittelee ainakin yhden mahdollisen tilan (x_j, y_j, z_j, n_j)
  - (x_j, y_j, z_j, n_j) on jopa yksikäsitteinen, mutta sillä ei ole jatkon kannalta merkitystä

$\Rightarrow$ jono Q_j voi päättyä vain kun $x = y = z = 1 \wedge n = 3$

- merkitään
  - $yr_4(j) = x_j + y_j + z_j + n_j - 6$
  - $yr_3(j) = x_j + y_j + z_j - 3$
  - $yr_2(j) = x_j + y_j - 2$
  - $yr_1(j) = x_j - 1$
- laillisissa tiloissa (siis myös kaikissa Q_j)

  $yr_4(j) \geq 0 \wedge yr_3(j) \geq 0 \wedge yr_2(j) \geq 0 \wedge yr_1(j) \geq 0$

  pätee

  $\quad yr_4(j) < yr_4(j-1)$

  $\vee\ yr_4(j) = yr_4(j-1) \wedge yr_3(j) < yr_3(j-1)$

  $\vee\ yr_4(j) = yr_4(j-1) \wedge yr_3(j) = yr_3(j-1) \wedge$
  $\quad yr_2(j) < yr_2(j-1)$

  $\vee\ yr_4(j) = yr_4(j-1) \wedge yr_3(j) = yr_3(j-1) \wedge$
  $\quad yr_2(j) = yr_2(j-1) \wedge yr_1(j) < yr_1(j-1)$

$\Rightarrow$ vektori $(yr_4(j), yr_3(j), yr_2(j), yr_1(j))$ on ylärajafunktio jonolle Q_j

$\Rightarrow$ jono Q_j päättyy johonkin tilaan Q_m

- edellä osoitettiin, että jono Q_j voi päättyä vain tilaan $x = y = z = 1 \wedge n = 3$

$\Rightarrow$ jono Q_j päättyy tilaan $x = y = z = 1 \wedge n = 3$

$\Rightarrow$ jokaiselle lailliselle tilalle Q_0 on olemassa äärellinen jono tiloja Q_0, Q_1, Q_2, …, Q_m siten, että

- $\langle$ **true** $\rangle\ x := 1; y := 1; z := 1; n := 3\ \langle\, Q_m \,\rangle$
- $\langle\, Q_m \,\rangle\, S\, \langle\, Q_{m-1} \,\rangle\, S \dots S\, \langle\, Q_1 \,\rangle\, S\, \langle\, Q_0 \,\rangle$

$\Rightarrow$ jos $L \Rightarrow x^n + y^n \neq z^n$, niin ohjelma tulee lopulta tilaan Q_0

Harjoitustehtäviä

1. (a) Osoita, että COUNTING-SORTin toinen **for**-silmukka täyttää spesifikaationsa.

(b) Keksi sopiva invariantti ja osoita, että COUNTING-SORTin kolmas **for**-silmukka täyttää spesifikaationsa.

(c) Miksi taulukon C alkioiden arvoalueeksi riittää 0, …, n? Käytä perustelussa hyväksi COUNTING-SORTin todistuksessa oikeaksi osoitettuja kaavoja.

2. (a) Fermat-ohjelman todistuksessa itse asiassa osoitettiin, että eräs ohjelma S toteuttaa spesifikaation

$$\langle\, x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3 \,\rangle$$
$$S$$
$$\langle\, x = y = z = 1 \wedge n = 3 \,\rangle$$

jos ja vain jos $\forall\, x, y, z, n \in \boldsymbol{Z}^+$: $(n \geq 3 \rightarrow x^n + y^n \neq z^n)$. Kirjoita S. Huomaa, että S:ksi ei kelpaa esim.

$x := 1;\ y := 1;\ z := 1;\ n := 3$

koska se toteuttaisi spesifikaation, vaikka olisi olemassa x, y, z ja $n \in \boldsymbol{Z}^+$ siten, että $n \geq 3$ ja $x^n + y^n = z^n$.

(b) Ohjelma S käännettynä *takaperin* on ohjelma $-S$, jolle $\langle\, Q \,\rangle\ -S\ \langle\, P \,\rangle$ jos ja vain jos $\langle\, P \,\rangle\ S\ \langle\, Q \,\rangle$. Keksi sääntöjä eri tyyppisten lauseiden ja lausejonojen kääntämiseksi takaperin. Vihje: käytä tarvittaessa epädeterministisiä lauseita.

3. Jos taulukon A alkio vie avaimeen verrattuna paljon muistia, taulukkoa ei kannata järjestää suoraan. On tehokkaampaa ottaa käyttöön aputaulukko C, ja menetellä seuraavasti:

for $i := 1$ **to** n **do** $C[i] := i$ **endfor**
järjestä C käyttäen avaimena $A[C[i]].key$
$\langle\, \forall\, i\ ;\ 1 \leq i \leq n\colon A_l[i] = A[C[i]] \wedge \exists\, x\ ;\ 1 \leq x \leq n\colon C[x] = i \,\rangle$
siirrä A:n alkiot oikeille paikoilleen C:n avulla
$\langle\, \forall\, i\ ;\ 1 \leq i \leq n\colon A_l[i] = A[i] \,\rangle$ (* l = "lopussa" *)

Viimeisen vaiheen voi tehdä seuraavalla ohjelmalla:

```
for i := 1 to n do
    if C[i] ≠ i then
        apu := A[i]; k := i;
        while C[k] ≠ i do
            h := C[k]; A[k] := A[h]; C[k] := k; k := h
        endwhile
        A[k] := apu; C[k] := k
    endif
endfor
```

(a) Osoita, että **then**-haara toteuttaa seuraavan spesifikaation, missä m, c_0, …, c_m, A_0 ja C_0 ovat haamumuuttujia:

$\langle\, A = A_0 \wedge C = C_0 \wedge (\forall\, l\ ;\ 0 \leq l \leq m\colon 1 \leq c_l \leq n) \wedge$
$i = c_0 = C[c_m] \wedge \forall\, l\ ;\ 0 \leq l \leq m-1\colon c_{l+1} = C[c_l] \neq c_0 \,\rangle$
then-haara
$\langle\, \forall\, j\ ;\ 1 \leq j \leq n\colon$
$j \in \{c_0, c_1, \ldots, c_m\} \wedge A[j] = A_0[C_0[j]] \wedge C[j] = j$
$\vee\ j \notin \{c_0, c_1, \ldots, c_m\} \wedge A[j] = A_0[j] \wedge C[j] = C_0[j] \,\rangle$

(b) Osoita, että viimeinen vaihe täyttää spesifikaationsa.

(c) Osoita, että aina kun ohjelma sijoittaa alkion A:han, se sijoittaa sen lopulliselle paikalleen.

4 TODISTAMISEN SOVELLUKSIA

Tässä luvussa pohditaan, mitä muuta hyötyä ohjelman todistamistaidosta on kuin valmiin koodin todistaminen

- valmis koodi ei yleensä edes ole virheetön, joten todistaminen ei voi onnistua!

Sisältö

- todistamisen yrittäminen katselmointikeinona
- todistamisen käyttö ohjelman suunnittelun apuna

4.1 Oikeaksi osoittaminen katselmointikeinona

Jo tehdyn ohjelman osoittaminen oikeaksi ei yleensä onnistu

- silmukkainvariantteja, ylärajafunktioita, sopivia predikaatin vahvennoksia ja heikennöksiä ym. voi olla vaikea keksiä jälkikäteen
- ohjelma tuskin edes on oikein

⇒

  - ohjelman todistus kannattaa laatia yhtäaikaa itse ohjelman kanssa (seuraava alaluku)
  - yritys todistaa valmis ohjelma on hyvä katselmointikeino (tämä alaluku)

Esimerkki: miten käy, jos yritetään todistaa johdannon virheellinen merkkijonojen samuuden vertailija?

```
1  ⟨ true ⟩
2  onsama := ( jono1.pituus = jono2.pituus );
3  if onsama then
4      for i := 1 to jono1.pituus do
5          onsama :=
6              jono1.merkki[i] = jono2.merkki[i];
7  ⟨ onsama ↔ jono1 = jono2 ⟩
```

- otamme käyttöön lyhenteitä
  - $j_1 = jono1,\ j_2 = jono2$
  - $.p = .pituus,\ .m = .merkki$
  - $os = onsama$
  - $P_7 \Leftrightarrow (onsama \leftrightarrow jono1 = jono2)$
    $\Leftrightarrow (os \leftrightarrow j_1 = j_2)$

- **if**-sääntö
  $\Rightarrow$ rivin 3 alussa on oltava voimassa P_3 siten, että
  (a) $P_3 \wedge os \;\Rightarrow\; wp$(rivit 4…6, P_7)
  (b) $P_3 \wedge \neg os \;\Rightarrow\; P_7$
- puretaan "$j_1 = j_2$" auki: $j_1 = j_2 \;\Leftrightarrow$
  $j_1.p = j_2.p \;\wedge\; \forall\, k\,;\, 1 \le k \le j_1.p\colon\, j_1.m[k] = j_2.m[k]$

$\Rightarrow$ $P_7 \;\Leftrightarrow\; (\, os \leftrightarrow$
  $j_1.p = j_2.p \;\wedge\; \forall\, k\,;\, 1 \le k \le j_1.p\colon\, j_1.m[k] = j_2.m[k]\,)$

- symbolisella suorituksella saadaan vahvin P_3 siten, että $\langle$**true**$\rangle$ rivi 2 $\langle P_3 \rangle$:
  $P_3 \;\Leftrightarrow\; (os \leftrightarrow j_1.p = j_2.p)$
- nyt $P_3 \wedge \neg os \;\Rightarrow\; \neg os \wedge j_1.p \ne j_2.p \;\Rightarrow\; \neg os \wedge j_1 \ne j_2$
  $\Rightarrow$ P_7, joten (b) pätee
  – tuli tarkistettua eripituisten jonojen tapaus
- (a):n todistamiseksi tarvitaan **for**-silmukalle invariantti I siten, että
  (c) $P_3 \wedge os \wedge i = 1 \;\Rightarrow\; I$
  (d) $\langle I \wedge 1 \le i \le j_1.p \rangle$ rivit 5…6; $i := i+1$ $\langle I \rangle$
  (e) $I \wedge i = j_1.p + 1 \;\Rightarrow\; P_7$
- varmaankin I:n tulisi olla suunnilleen muotoa
  $I_1 \;\Leftrightarrow\; (\, os \leftrightarrow \forall\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] = j_2.m[k]\,)$
  – (c) pätee
  – (e) ei päde: P_7 testaa myös $j_1.p = j_2.p$
- korjaus
  – $P_3 \wedge os \;\Rightarrow\; j_1.p = j_2.p$
  ja $j_1.p$ ja $j_2.p$ ovat kiinteitä
  $\Rightarrow$ $j_1.p = j_2.p$ voidaan lisätä invarianttiin (c):n ja (d):n rikkoutumatta
  $I_2 \;\Leftrightarrow\; j_1.p = j_2.p \wedge I_1$

- nyt
  – $P_3 \wedge os \wedge i = 1 \;\Rightarrow\; os \wedge j_1.p = j_2.p \wedge i = 1 \;\Rightarrow\; I_2$, siis (c) $\cdot/\cdot$
  – $I_2 \wedge i = j_1.p + 1$
  $\Rightarrow$ $j_1.p = j_2.p \;\wedge$
  $(\, os \leftrightarrow \forall\, k\,;\, 1 \le k \le j_1.p\colon\, j_1.m[k] = j_2.m[k]\,)$
  $\Rightarrow$ $os \leftrightarrow$
  $j_1.p = j_2.p \wedge \forall\, k\,;\, 1 \le k \le j_1.p\colon\, j_1.m[k] = j_2.m[k]$
  (miksi tämä pätee kun $j_1.p \ne j_2.p$?)
  $\Leftrightarrow$ P_7, siis (e) $\cdot/\cdot$
  – vielä (d): wp(rivit 5…6; $i := i+1$, I_2)
  $\Leftrightarrow$ $I_2[\, i \leftarrow i+1\,][\, os \leftarrow (j_1.m[i] = j_2.m[i])\,]$
  $\Leftrightarrow$ $(\, j_1.p = j_2.p \;\wedge$
  $(\, os \leftrightarrow \forall\, k\,;\, 1 \le k \le i\colon\, j_1.m[k] = j_2.m[k]\,)\,)$
  $[\, os \leftarrow (j_1.m[i] = j_2.m[i])\,]$
  $\Leftrightarrow$ $j_1.p = j_2.p \wedge (\, j_1.m[i] = j_2.m[i] \leftrightarrow$
  $\forall\, k\,;\, 1 \le k \le i\colon\, j_1.m[k] = j_2.m[k]\,)$
  $\Leftrightarrow$ $j_1.p = j_2.p \wedge (\, j_1.m[i] \ne j_2.m[i] \vee$
  $\forall\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] = j_2.m[k]\,)$
  $\Leftrightarrow\colon X$
- nyt on osoitettava, että $I_2 \wedge 1 \le i \le j_1.p \;\Rightarrow\; X$
  – jos os pätee, niin $j_1.p = j_2.p \wedge I_1$
  $\Rightarrow$ $j_1.p = j_2.p \wedge \forall\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] = j_2.m[k]$
  $\Rightarrow$ X
  – jos os ei päde, niin $j_1.p = j_2.p \wedge I_1 \;\Leftrightarrow$
  $j_1.p = j_2.p \wedge \neg \forall\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] = j_2.m[k]$,
  joten ainoa tapa varmistaa X on osoittaa $j_1.m[i] \ne j_2.m[i]$
  – aputulos: $\neg \forall\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] = j_2.m[k]$
  $\Leftrightarrow\; \exists\, k\,;\, 1 \le k \le i-1\colon\, j_1.m[k] \ne j_2.m[k]$

- ongelma: on osoitettu, että $j_1.m[k] \ne j_2.m[k]$ jollekin $1 \le k \le i-1$, mutta tarvitsisi osoittaa, että se pätee kun $k = i$

$\Rightarrow$ onko invariantti liian heikko vai onko ohjelmassa virhe?

- koe: mitä tapahtuu, jos $j_1.m[k] \ne j_2.m[k]$ jollekin $1 \le k < i$, mutta $j_1.m[i] = j_2.m[i]$?

$\Rightarrow$ ajetaan ohjelma syötteellä 'aa' 'ba'
  – vastaus: "`true`" väärin!!!
  – ohjelma laskee itse asiassa
  os := $(\, j_1.p = j_2.p\,) \wedge (\, j_1.m[j_1.p] = j_2.m[j_1.p]\,)$

Havainto

- todistusyritys ajautui odottamattomiin vaikeuksiin tietyssä tilanteessa
  $\Rightarrow$ onko vika ohjelmassa vai todistuksessa?
- asiaa kokeiltiin k.o. tilanteeseen johtavalla syötteellä
  $\Rightarrow$ ohjelmasta löytyi vika

$\Rightarrow$ todistusyritys auttoi löytämään syötteen, joka paljasti virheen

Muistilista ohjelmanpätkän katselmoimiseksi todistamalla

1. Palauttaako ohjelmanpätkä oikean tuloksen, jos pääsee loppuun asti?
   – ohjelma koristellaan predikaateilla
   – predikaatit yritetään osoittaa voimassa oleviksi aina kun (ja jos) ohjelma on ko. kohdassa
   – tarvittaessa predikaatteja täsmennetään, kunnes voimassa olon osoittaminen onnistuu, tai paikallistuu virhe ohjelmassa
   – yksinkertaisuuden vuoksi toistaiseksi unohdetaan se mahdollisuus, että ohjelma tekee kiellettyjä operaatioita
2. Tekeekö ohjelmanpätkä mitään kiellettyä?
   – ovatko kaikki lausekkeet laskettavissa?
   – pysyvätkö taulukkojen indeksoinnit sallitulla alueella?
   – …
   – tarkistetaan 1. kohdassa johdettujen predikaattien avulla
   – tarvittaessa täsmennetään predikaatteja ja palataan 1. kohtaan
3. Pääseekö ohjelmanpätkä varmasti loppuun?
   – eihän sijoiteta **for**-silmukan silmukkamuuttujaan?
   – löytyykö jokaiselle **while**-, **repeat**- tms. silmukalle ylärajafunktio? Tarkistettava, että se on ylärajafunktio!
   – onko jokaisella rekursiolla pohja? Todistus esim. antamalla ylärajafunktio rekursiotasoille

- käytännössä suurin osa asioista on tarkistettavissa helposti
  - esim. Pascalissa kieli estää sijoittamisen **for**-silmukan silmukkamuuttujaan
    ⇒ **for**-silmukka lopettaa varmasti
  - esim. alla taulukon $A[1\ldots n]$ indeksointi on aina laillinen **for**-silmukan alun vuoksi

    for $i := 1$ **to** n **do**
    $A[i] := 0$
    endfor
- kohtia 1 – 3 ei kannata aina tarkistaa tässä järjestyksessä, vaan helpot ensin
  - muuten helpot saattavat jäädä kokonaan tarkistamatta, ja siellähän ne ylenkatsotut virheet usein ovat!
  - predikaattien valinnassa on hyvä, että tiedetään, millaisia kohdat 2 ja 3 tarvitsevat
  - vaikeiden asioiden tarkistamista helpottaa, jos käytettävissä joukko tarkistettua yleistä tietoa

Esimerkki: tarkistettava taulukon $A[1\ldots n]$ indeksointi

$i := 1$;
while $i < n \wedge A[i] \neq x$ **do**
$i := i+1$
endwhile

- osoitettava: aina kun testataan, mennäänkö seuraavalle kierrokselle, on oltava $1 \le i \le n$

⇒ otetaan $1 \le i \le n$ silmukkainvariantin osaksi

- yritetään tarkistaa, että $1 \le i \le n$ on invariantti:
  - $\langle$ **true** $\rangle$ $i := 1$ $\langle$ $i = 1$ $\rangle$
  - $i = 1 \Rightarrow 1 \le i \le n$ **vain jos** $n \ge 1$!!

⇒ tarvitaan esiehto $n \ge 1$

- todistaminen on tehokas keino juuri tällaisten piilevien esiehtojen havaitsemiseen!
  (jos laskija on huolellinen …)
- uusi yritys …
  - $\langle\ n \ge 1\ \rangle$ $i := 1$ $\langle\ i = 1 \wedge n \ge 1\ \rangle$
  - $i = 1 \wedge n \ge 1 \Rightarrow 1 \le i \le n$
  - $wp(i := i+1,\ 1 \le i \le n) \Leftrightarrow 1 \le i+1 \le n \Leftrightarrow 0 \le i \le n-1 \Leftarrow 1 \le i \le n \wedge i < n$, joten
    $\langle\ 1 \le i \le n \wedge i < n \wedge A[i] \neq x\ \rangle$ $i := i+1$ $\langle\ 1 \le i \le n\ \rangle$

Esimerkki: toisenlainen taulukon indeksointi:

$i := 1$;
while $i \le n$ **andthen** $A[i] \neq x$ **do** $i := i+1$
endwhile

- **andthen** laskettaa jälkimmäisen osan vain, jos edellinen tuotti **true**

⇒ **andthen** takaa, että indeksoitaessa aina $i \le n$
⇒ riittää ottaa invarianttiin $i \ge 1$
⇒ esiehtoa $n \ge 1$ ei tarvita

Seuraaviin asioihin kannattaa kiinnittää erityistä huomiota:

- sivuvaikutukset: voiko funktio tai aliohjelma muuttaa sellaista muuttujaa, jota ei ole tarkoitettu muutettavaksi?
- “aliasing”: voiko kaksi eri nimeä tarkoittaa samaa muuttujaa (jolloin toisen muuttaminen muuttaa molempia)?
  - esim. $A[i]$, $A[j]$ kun $i = j$
  - esim. aliohjelmien viiteparametrit tai viiteparametri vs. ei-paikallinen muuttuja
  - esim. osoittimet

- makrot: säilyykö rakenne haluttuna aukaisussa?
  - vrt. `#define sqr(x) = x*x`
    `sqr(2+3)` ➠ `2+3*2+3` eikä `(2+3)*(2+3)`
- muuttujien alustukset: onko ohjelmalla toteutumattomia esiehtoja muotoa “$i = 0$”?
- patologiset syötteet: olettaako esim. taulukkoa $A[1\ldots n]$ käyttävä ohjelma, että $n \ge 1$?
  - tällaisessa tapauksessa on yleensä mielekästä sallia $n = 0$, mutta ei $n < 0$
- muistin varaus ja vapautus
  - roskaantuuko muistia?
  - syntyykö viitteitä vapautettuun muistiin?
- onko käskyn kaikkiin mahdollisiin lopputuloksiin varauduttu?
  - esim. käsitelläänkö kaikki erilaiset palvelun paluukoodit asianmukaisesti?
- voiko muisti loppua kesken?
  - pinon käyttö ➠ rekursio ➠ paikallisten muuttujien määrät ja koot
  - taulukoiden koot
  - dynaamisesti varattavan muistin määrä

  (näitä on usein vaikea tarkistaa todistustekniikoilla, mutta joskus löytyy sopivia invariantteja, joilla esim. tietueiden määrää voi arvioida)
- voivatko lukualueet ylittyä?
  - varsinkin 1 ja 2 tavun kokonaisluvut

  (tätäkin on yleensä vaikea tarkistaa)
- rinnakkaisuuden ongelmat: voiko joku toinen ohjelmanpätkä tunkeutua kriittiseen väliin?
- …

⇒ 8101000 Ohjelmointikielten periaatteet

Esimerkki: tekstialkioiden tunnistamisohjelma

- annettu:
  - rivi merkkejä taulukossa *rivi*[1…*rivipit*]
  - taulukollinen tekstialkioita: *alkio*[1…*alkioita*]
    alkion i pituus: *alkio*[*i*].*pit*
    alkion i merkit: *alkio*[*i*].*m*[1…*alkio*[*i*].*pit*]
  - seuraava ohjelmanpätkä:

```
 1. k := 1;  (* k = kohdistin *)
 2. while k ≤ rivipit do
 3.     if rivi[k] = ' ' then k := k+1
 4.     else
 5.         täsmää := false; i := 0;
 6.         while i < alkioita ∧ ¬täsmää do
 7.             i := i+1; täsmää := true;
 8.             for j := 1 to alkio[i].pit do
 9.                 täsmää := täsmää ∧
10.                     alkio[i].m[j] = rivi[k+j–1]
11.             endfor
12.         endwhile
13.         if täsmää then
14.             write(i); k := k+alkio[i].pit
15.         else
16.             write('virhe kohdassa ', k)
17.         endif
18.     endif
19. endwhile
```

- aloitetaan kohdasta 2: tekeekö mitään kiellettyä?

- ohjelman käyttötarkoituksen huomioon ottaen on järkevää tehdä seuraavia **oletuksia:**
  - $rivipit \geq 0$, $alkioita \geq 0$ (vai ≥ 1?), $alkio[i].pit \geq 1$
  - alkioissa ei välilyöntejä
- jos ohjelmalle ei ole annettu tarkkaa spesifikaatiota, niin
  - tämän tapaisia oletuksia on pakko tehdä
  - **ne on muistettava kirjata näkyville!**
- havaintoja (helppo tarkistaa ohjelmasta):
  (a) $alkio[i].pit \geq 1 \Rightarrow$ k:n arvo ei koskaan pienene
  $\Rightarrow$ rivin 1 jälkeen aina $k \geq 1$
  (b) riveillä 6, …, 17 pätee $0 \leq i \leq alkioita$
- mahdollisesti kiellettyjä operaatioita ovat
  - ylivuotovaara jokaisen sijoituksen yhteydessä (rivit 1, 3, 5ab, 7ab, 8, 9 ja 14)
  - taulukoiden indeksoinnit (riv. 3, 8, 10abc ja 14)

Ylivuotovaarojen tarkistus

- **oletuksia:**
  - ≥ 16 bitin kokonaisluvut
  - $rivipit$, $alkioita$ ja $alkio[i].pit \leq 10\,000$
- y.o. oletukset tehdään, koska
  - realistisia: luultavasti voimassa aina ohjelmaa käytettäessä (silti muistettava kirjata näkyviin!)
  - helpottavat tarkistuksia (vertailun vuoksi tilannetta analysoidaan myös ilman näitä oletuksia)
- rivit 1, 5, 7b: sijoitetaan sallitun kokoinen vakioarvo $\cdot/\cdot$
- rivi 3:
  - $k \leq rivipit \leq 10\,000 \Rightarrow k+1 \leq 10\,001 \Rightarrow$ ei ylivuotoa
  - k kasvaa $\Rightarrow$ ei alivuotoa $\cdot/\cdot$
  - (ilman: oltava $rivipit <$ suurin kokonaisluku)

- rivi 7a:
  - $i < alkioita \leq 10\,000 \Rightarrow i+1 \leq 10\,000$
  $\Rightarrow$ ei ylivuotoa
  - i kasvaa $\Rightarrow$ ei alivuotoa $\cdot/\cdot$
  - (ilman: saa olla $alkioita$ = suurin kok.luku
  $\Rightarrow$ toteutuu automaattisesti, jos i samaa tai suurempaa tyyppiä kuin $alkioita$)
- rivi 8: $1 \leq j \leq alkio[i].pit \leq 10\,000$ $\cdot/\cdot$
  - (ilman: saa olla $alkio[i].pit$ = suurin kok.luku)
- rivi 9: Boolean-tyyppisille arvoille ei voi tulla ylivuotoja $\cdot/\cdot$
- rivi 14:
  - riveillä 4…13 ei sijoiteta k:hon
  $\Rightarrow$ tänne tultaessa $k \leq rivipit \leq 10\,000$
  $\Rightarrow$ $k+alkio[i].pit \leq 20\,000$ $\cdot/\cdot$
  - (ilman: saattaa olla tarpeen varmistaa, että $rivipit + alkio[i].pit \leq$ suurin kokonaisluku)

$\Rightarrow$ ei ylivuotovaaraa **yllä mainituilla oletuksilla**!

Indeksoinnit:

- rivi 3: hav. (a) $\Rightarrow$ $1 \leq k \leq rivipit$ $\cdot/\cdot$
- rivi 8: hav. (b) $\wedge$ rivit 6, 7 $\Rightarrow$ $1 \leq i \leq alkioita$ $\cdot/\cdot$
- rivi 10a: $\cdot/\cdot$, koska rivi 8 $\cdot/\cdot$
- rivi 10b: rivin 8 vuoksi $1 \leq j \leq alkio[i].pit$ $\cdot/\cdot$
- rivi 10c: $1 \leq j \leq alkio[i].pit \Rightarrow$
  $k \leq k+j-1 \leq alkio[i].pit+k-1$
  - k:sta tiedetään vain $1 \leq k \leq rivipit$
  $\Rightarrow$ mikään ei takaa, että $alkio[i].pit+k-1 \leq rivipit$!

$\Rightarrow$ löytyi virhe: jos ollaan niin lähellä rivin loppua, että testattava alkio ei enää mahdu kokonaan riville, ohjelma yrittää indeksoida rivin lopun ohi

Tämäkin virhe olisi ollut hankala havaita testaamalla, mutta olisi saattanut esiintyä käytännössä

- väärä tulos vain, jos rivin lopun takana olevat merkit sopivat kokeiltavaan alkioon
  - epätodennäköistä
- ylivuoto vasta, kun etsintä ohittaa taulukolle *rivi* varatun muistitilan
  $\Rightarrow$ syöterivin oltava lähes maksimipituinen

Harjoitustehtäviä

1. Mikä puolitushaun todistuksessa menee pieleen (jos mikään), kun ohjelmaa muutellaan seuraavasti? Jos todistus särkyy mutta ohjelma ei, niin korjaa todistus.

   ⟨ **true** ⟩
   1 a := 1; y := n+1;
   2 **while** $a < y$ **do**
   3 v := (a+y) div 2;
   4 **if** $A[v] <$ $avain$ **then** a := v+1
   5 **else** y := v
   6 **endif**
   7 **endwhile**
   ⟨ ($a = n+1 \vee A[a] \geq avain$) $\wedge$
   ($a = 1 \vee A[a-1] < avain$) ⟩

   (a) Rivi 1 muutetaan muotoon a := 1; y := n;

   (b) Rivi 4 muutetaan muot. **if** $A[v] \leq avain$ **then** a := v+1

   (c) Rivi 4 muutetaan muotoon **if** $A[v] < avain$ **then** a := v

   (d) Rivi 5 muutetaan muotoon
   elsif $A[v] > avain$ **then** y := $v-1$ **else** a := v; y := v

2. Eräs sankari ei millään voinut ymmärtää, kun väitin, että hänen suunnilleen alla olevan kaltainen Basic-ohjelmansa toimii väärin. Ohjelman tuli löytää muutaman luvun joukosta maksimi. Mikä virhe? Laske heikoin esiehto, jonka vallitessa ohjelma toimii oikein, kun syöteaineisto on n (> 0), a_1, a_2, …, a_n.

```
10 read n
20 for i = 1 to n
30 read x
40 if x <= max goto 60
50 max = x
60 next i
70 print max
```

3. Luennoilla löydetyn virheen korjaamiseksi tekstialkioiden tunnistamisohjelma muutettiin alla olevaksi. Vaan ei se ole vieläkään kunnossa. Miksei?

```
k := 1;  (* k = kohdistin *)
while k ≤ rivipit do
    if rivi[k] = ' ' then k := k+1
    else
        täsmää := false; i := 0;
        while i < alkioita ∧ ¬ täsmää do
            i := i+1;
            if k+alkio[i].pit−1 ≤ rivipit then
                täsmää := true;
                for j := 1 to alkio[i].pit do
                    täsmää := täsmää ∧
                        alkio[i].m[j] = rivi[k+j−1]
                endfor
            endif
        endwhile
        if täsmää then
            write(i); k := k+alkio[i].pit
        else
            write('virhe kohdassa ', k)
        endif
    endif
endwhile
```


4.2 Ohjelman ja todistuksen suunnittelu yhdessä

Hankala ohjelmanpätkä kannattaa usein suunnitella yhtäaikaa todistuksensa kanssa

- ohjelman etenemisen pääideat kannattaa dokumentoida kirjoittamalla välivaiheita kuvaavia predikaatteja
- silmukka, sen invariantti ja ylärajafunktio kannattaa suunnitella yhtäaikaa

Esimerkki: polynomin arvon laskeminen Hornerin säännöllä

- n:nnen asteen polynomi $P(x)$ on lauseke muotoa
  $$P(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$
  – sopimus: $\forall\, x\colon x^0 = 1$
- suoraviivainen laskeminen edellyttää
  – $2n$ kertolaskua (tai typeröiden jopa $(^1/_2)(n^2+n)$)
  – n yhteenlaskua
- $P(x)$ voidaan laskea vähemmillä kertolaskuilla Hornerin säännön avulla:
  $$P(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0$$
  – n kertolaskua
  – n yhteenlaskua
- kuinka tehdään tätä hyödyntävä ohjelma?
- syötteet ja tulosteet:
  – $n \geq -1$
  – $a_0 \ldots a_n$ on annettu taulukossa $A[0\ldots n]$
  – $A[0\ldots n]$ kiinteä
  – lopputulos kootaan muuttujaan Px
- tavoite: lopussa $Px = \Sigma_{j=0}^{n} A[j]x^j$

- Hornerin säännön muoto houkuttelee etenemään sisältä ulos, eli käyttämään **for**-silmukkaa muotoa

```
for i := n downto 0 do
    Px := f(i, Px)
endfor
```

  eli **while**-silmukka muotoa

```
i := n;
while i ≥ 0 do
    Px := f(i, Px); i := i−1
endwhile
```

- invariantin muodostamiseksi ja f:n keksimiseksi jaetaan polynomin arvo kahteen osaan:
  – jo lasketun esittää Px
  – vielä laskematon esitetään kaavana

$$\begin{aligned} P(x) &= (\ldots((\ldots(a_n x + a_{n-1})x + \ldots)x + a_i)x + \ldots)x + a_0 \\ &= (\ldots((\quad\; Px \quad\;)x + a_i)x + \ldots)x + a_0 \end{aligned}$$

$\Rightarrow$ invariantti
$$P(x) = (\ldots((Px)x + a_i)x + \ldots)x + a_0$$

- alussa $i = n$, joten täytyy olla $Px \cdot x = 0$
  – saadaan voimaan sijoittamalla $Px := 0$
- lopussa $i = -1$, joten $P(x) = Px$
- jokaisella silmukan kierroksella Px laajenee kattamaan yhden uuden sulkulausekkeen
  $\Rightarrow$ valitaan $f(i, Px) = Px \cdot x + A[i]$
- siis

```
Px := 0; i := n;
while i ≥ 0 do
    Px := Px·x + A[i]; i := i−1
endwhile
```


eli **for**-silmukkana

```
Px := 0;
for i := n downto 0 do
    Px := Px·x + A[i]
endfor
```

- varmuuden vuoksi osoitetaan tämä oikeaksi vielä tavallisella polynomin lausekkeella ja summamuotoon kirjoitetulla invariantilla
  – invariantiin joudutaan lisäämään $i \geq -1$

```
⟨ n ≥ −1 ⟩
Px := 0; i := n;
⟨ Px = 0 ∧ i = n ≥ −1 ⟩
⟨ inv: Px·x^(i+1) + Σ_{j=0}^{i} A[j]x^j = Σ_{j=0}^{n} A[j]x^j ∧ i ≥ −1 ⟩
while i ≥ 0 do
    Px := Px·x + A[i]; i := i−1
endwhile
⟨ Px = Σ_{j=0}^{n} A[j]x^j ⟩
```

- silmukan tarkistus
  – $Px = 0 \wedge i = n \geq -1 \;\Rightarrow$
    $Px \cdot x^{i+1} + \Sigma_{j=0}^{i} A[j]x^j = 0 + \Sigma_{j=0}^{n} A[j]x^j \;\wedge\; i \geq -1$
    $\Rightarrow\; I\;\; \cdot/\cdot$
  – $wp(\, Px := Px \cdot x + A[i];\; i := i-1,\; I\,)$
    $\Leftrightarrow\; (\, Px \cdot x + A[i]\,) \cdot x^i + \Sigma_{j=0}^{i-1} A[j]x^j = \Sigma_{j=0}^{n} A[j]x^j$
    $\wedge\;\; i \geq 0$
    $\Leftrightarrow\; Px \cdot x^{i+1} + \Sigma_{j=0}^{i} A[j]x^j = \Sigma_{j=0}^{n} A[j]x^j \;\wedge\; i \geq 0$,
    joten
    $\langle\, I \wedge i \geq 0 \,\rangle\;\; Px := Px \cdot x + A[i];\; i := i-1 \;\; \langle\, I \,\rangle\;\; \cdot/\cdot$
  – $I \wedge i < 0 \;\Rightarrow\; I \wedge i = -1$
    $\Rightarrow\; Px = Px \cdot x^0 + 0 = \Sigma_{j=0}^{n} A[j]x^j\;\; \cdot/\cdot$
  – päättyminen: $i+1$ on ylärajafunktio $\;\cdot/\cdot$

Esimerkki: pisin nouseva sisäjono: tehtävän asettelu

- jono β on jonon α *sisällä* tai *sisäjono*, jos β voidaan tehdä α:sta poistamalla 0 tai useampia alkioita
  - esim. jonon $\langle 1,9,2,1 \rangle$ sisäjonot ovat $\langle\rangle$, $\langle 1 \rangle$, $\langle 9 \rangle$, $\langle 2 \rangle$, $\langle 1,9 \rangle$, $\langle 1,2 \rangle$, $\langle 1,1 \rangle$, $\langle 9,2 \rangle$, $\langle 9,1 \rangle$, $\langle 2,1 \rangle$, $\langle 1,9,2 \rangle$, $\langle 1,9,1 \rangle$, $\langle 1,2,1 \rangle$, $\langle 9,2,1 \rangle$ ja $\langle 1,9,2,1 \rangle$
- jono on
  - *nouseva*, jos seuraava alkio on aina edellistä suurempi
  - *ei-laskeva*, jos seuraava alkio on aina vähintään edellisen suuruinen
- tehtävänä on laatia ohjelma, jolle annetaan jono taulukossa $A[1\ldots n]$, ja joka tuottaa luvun k ja taulukon $B[1\ldots k]$ siten, että B on mahdollisimman pitkä A:n nouseva sisäjono

Pisin nouseva sisäjono: spesifikaatio

- syötteet: kiinteä taulukko $A[1\ldots n]$
- tuloksen määrittelemiseksi tarvitsemme predikaatin ilmaisemaan, että $B[1\ldots k]$ on A:n nouseva sisäjono; määritellään se vaiheittain:
  - $\mathit{sisäjono}(B, A) \;:\Leftrightarrow\; \exists\, j_1, j_2, \ldots, j_k$:
    $1 \le j_1 < j_2 < \ldots < j_k \le n \;\wedge$
    $\forall\, i\,;\, 1 \le i \le k$: $B[i] = A[j_i]$
  - $\mathit{nouseva}(B) \;:\Leftrightarrow\; \forall\, i\,;\, 1 \le i < k$: $B[i] < B[i+1]$
  - $\mathit{ns}(B,A) \;:\Leftrightarrow\; \mathit{nouseva}(B) \wedge \mathit{sisäjono}(B, A)$
- lopussa B:n pitää olla *maksimaalinen* eli mahdollisimman pitkä nouseva sisäjono
  - voidaan ilmaista esim. $\mathit{ns}(C[1\ldots m], A) \rightarrow m \le k$
  - helposti huomataan, että
    $\mathit{ns}(C[1\ldots m], A) \wedge 0 \le j < m \;\Rightarrow\; \mathit{ns}(C[1\ldots j], A)$
  - $\Rightarrow$ helpompi tapa: $\forall\, C[1\ldots k+1]$: $\neg\, \mathit{ns}(C, A)$

$\Rightarrow$ määritellään maksimaalinen nouseva sisäjono:

$\mathit{mns}(B[1\ldots k], A[1\ldots n]) \;\Leftrightarrow$
$\mathit{ns}(B, A) \wedge \forall\, C[1\ldots k+1]$: $\neg\, \mathit{ns}(C, A)$

- ohjelman spesifikaatioksi saadaan

$\langle$ **true** $\rangle$
S
$\langle\ \mathit{mns}(B[1\ldots k], A[1\ldots n])\ \rangle$

Pisin nouseva sisäjono: ohjelman suunnittelun alku

- koetetaan “pala kerrallaan” -strategiaa: yritetään kerätä vastaus selaamalla A läpi silmukassa

$\Rightarrow$ arvaus silmukaksi ja sen invariantiksi I:

$\langle$ **inv:** $\mathit{mns}(B[1\ldots k], A[1\ldots i-1])$ $\rangle$
for i := 1 **to** n **do**
  ??
endfor

- I triviaalisti totta kun $i = 1$, jos alustetaan $k := 0$
- $I \wedge i = n+1 \;\Rightarrow\; \mathit{mns}(B, A)$

$\Rightarrow$ ohjelma näyttää seuraavalta:

$\langle$ **true** $\rangle$
$k := 0$;
$\langle$ **inv:** $\mathit{mns}(B[1\ldots k], A[1\ldots i-1])$ $\rangle$
for i := 1 **to** n **do**
  $\langle\ \mathit{mns}(B[1\ldots k], A[1\ldots i-1]) \wedge 1 \le i \le n\ \rangle$
  ??
  $\langle\ \mathit{mns}(B[1\ldots k], A[1\ldots i])\ \rangle$
endfor
$\langle\ \mathit{mns}(B[1\ldots k], A[1\ldots n])\ \rangle$

- mitä pitää tehdä, jotta invariantti säilyisi voimassa, kun i kasvaa yhdellä?

- jos $A[i] > B[k]$, riittää lisätä $A[i]$ B:n jatkoksi

if $A[i] > B[k]$ **then**
  $k := k+1$; $B[k] := A[i]$
else
  ??
endif

- jos $A[i] \le B[k]$, niin $B[1\ldots k]$ on $A[1\ldots i]$:n nouseva sisäjono, mutta ei välttämättä maksimaalinen!
  - esimerkki: $A = \langle 8,1,2,4,0 \rangle$, $k = 1$, $B = \langle 8 \rangle$, $i = 3$

$\Rightarrow$ näyttää tarpeelliselta pitää kirjaa ainakin joistakin vaihtoehtoisista nousevista sisäjonoista

Mistä sisäjonoista tarvitsee pitää kirjaa?

- hyvinkin lyhyt sisäjono voi kasvaa pisimmäksi
  - esim. $\langle 6,7,\mathbf{1},8,9,\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5} \rangle$
- saman pituisista se, joka päättyy pienempään, ei voi olla huonompi kuin isompaan päättyvä
  - esim. $\langle 6,\mathbf{1},7,\mathbf{2},9,\mathbf{8},\ldots \rangle$

$\Rightarrow$ kannattaa pitää kirjaa jokaiselle sisäjonon pituudelle yhdestä mahdollisimman pieneen alkioon päättyvästä sisäjonosta

- esim. $\langle 4,6,1,9,2,\ldots \rangle$

	sisäjonot i:n funktiona				
pituus	1	2	3	4	5
1	$\langle 4 \rangle$	$\langle 4 \rangle$	$\langle 1 \rangle$	$\langle 1 \rangle$	$\langle 1 \rangle$
2	—	$\langle 4,6 \rangle$	$\langle 4,6 \rangle$	$\langle 4,6 \rangle$	$\langle 1,2 \rangle$
3	—	—	—	$\langle 4,6,9 \rangle$	$\langle 4,6,9 \rangle$
4	—	—	—	—	—

  - merkitään näitä jonoja B_h, missä h on pituus
  - merkitään k = suurin käytössä oleva h:n arvo
  - rajatapausten helpottamiseksi olkoon $B_0 = \langle\rangle$

- tarvitaan predikaatti “pienimpään loppuva nouseva sisäjono”, joka väittää, että $B[1\ldots h]$:lla on A:n alkuosan h:n pituisista nousevista sisäjonoista pienin viimeinen alkio

$\mathit{plns}(\ B[1\ldots h], A[1\ldots i]\ ) \;:\Leftrightarrow\; \mathit{ns}(B, A[1\ldots i])$
$\wedge\quad \forall\, C[1\ldots h]$: $\neg \mathit{ns}(C, A[1\ldots i]) \vee C[h] \ge B[h]$

$\Rightarrow$ ohjelma näyttää nyt seuraavalta:

$\langle$ **true** $\rangle$
$k := 0$;
$\langle$ **inv:** $\forall\, h\,;\, 1 \le h \le k$: $\mathit{plns}(\ B_h[1\ldots h], A[1\ldots i-1]\ )$
  $\wedge\ \mathit{mns}(B_k[1\ldots k], A[1\ldots i-1])$ $\rangle$
for i := 1 **to** n **do**
  $\langle\ \forall\, h\,;\, 1 \le h \le k$: $\mathit{plns}(\ B_h[1\ldots h], A[1\ldots i-1]\ )$
    $\wedge\ \mathit{mns}(B_k[1\ldots k], A[1\ldots i-1]) \wedge 1 \le i \le n\ \rangle$
  ??
  $\langle\ \forall\, h\,;\, 1 \le h \le k$: $\mathit{plns}(B_h[1\ldots h], A[1\ldots i])$
    $\wedge\ \mathit{mns}(B_k[1\ldots k], A[1\ldots i])\ \rangle$
endfor
$\langle\ \mathit{mns}(B_k[1\ldots k], A[1\ldots n])\ \rangle$
$B := B_k$
$\langle\ \mathit{mns}(B[1\ldots k], A[1\ldots n])\ \rangle$

Sisäjonojen ylläpito

- miten sisäjonoja B_h on muutettava, kun $A[i]$ käsitellään?
- $A[i]$:n voi sijoittaa vain niiden sisäjonojen jatkoksi, joiden viimeistä alkiota suurempi se on
  - ts. $A[i] > B_h[h]$
  - tältä osin B_0:n jatkoksi saa sijoittaa mitä vain
  - $\Rightarrow$ vaaditaan $h = 0 \vee A[i] > B_h[h]$
- sisäjonon B_h pidennys ei itse asiassa muuta B_h:ta, vaan uusii B_{h+1}:n

- toisaalta B_h:n pidennys $A[i]$:lla kannattaa vain, jos siten saadaan kaikkia aikaisempia pitempi jono, tai entistä parempi $h+1$:n mittainen jono
  - ts. $h = k \vee A[i] < B_{h+1}[h+1]$
  - (kun $A[i] = B_{h+1}[h+1]$ niin pidennyksestä ei haittaa eikä hyötyä)

$\Rightarrow$ B_h pidennetään (ts. B_{h+1} uusitaan) jos ja vain jos $(\ h = 0 \vee A[i] > B_h[h]) \wedge (h = k \vee A[i] < B_{h+1}[h+1]\ )$

$\Rightarrow$ ohjelma

$\langle$ **true** $\rangle$
$k := 0$;
$\langle$ **inv:** $\forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i-1]\ )$
$\quad \wedge\ mns(B_k[1\ldots k],\ A[1\ldots i-1])\ \rangle$
for $i := 1$ **to** n **do**
$\quad \langle\ \forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i-1]\ )$
$\quad\ \wedge\ mns(B_k[1\ldots k],\ A[1\ldots i-1]) \wedge 1 \le i \le n\ \rangle$
$\quad \langle$ **inv:** ?? $\rangle$
$\quad$ **for** $j := 0$ **to** k **do**
$\quad\quad$ **if** $(\ j = 0$ **orelse** $A[i] > B_j[j]\ ) \wedge$
$\quad\quad\quad (\ j = k$ **orelse** $A[i] < B_{j+1}[j+1]\ )$ **then**
$\quad\quad\quad\ B_{j+1}[1\ldots j] := B_j$; $B_{j+1}[j+1] := A[i]$
$\quad\quad\quad\ $ **if** $j = k$ **then** $k := k+1$ **endif**
$\quad\quad$ **endif**
$\quad$ **endfor**
$\quad \langle\ \forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i]\ )$
$\quad\ \wedge\ mns(B_k[1\ldots k],\ A[1\ldots i])\ \rangle$
endfor
$\langle\ mns(B_k[1\ldots k],\ A[1\ldots n])\ \rangle$
$B := B_k$
$\langle\ mns(B[1\ldots k],\ A[1\ldots n])\ \rangle$

- kolme **for**-silmukkaa sisäkkäin (missä kolmas?)
  $\Rightarrow$ näyttää aika tehottomalta!

Sisäjonokirjanpidon tehostaminen 1

- pahimmillaan $k = n$
  $\Rightarrow$ taulukoissa $B_0, \ldots, B_k$ yhteensä $\Theta(n^2)$ alkiota
  - kuitenkin jonot muodostetaan vain n alkiosta
  $\Rightarrow$ voitaisiinko välttää jokaisen jonon pito omassa B-taulukossaan?

$\Rightarrow$ kysymyksiä
- millä eri tavoin sama alkio voi olla useassa jonossa?
- minkä jonojen jatkeeksi $A[i]$ voi tulla?

- reunatapauksia $h = 0$ ja $h = k$ lukuunottamatta $A[i]$ lisätään jonon B_h jatkeeksi vain jos
  $B_h[h] < A[i] < B_{h+1}[h+1]$
  mutta tällöin
  $B_h[h] < B_{h+1}[h+1]$

$\Rightarrow$ kysymys: milloin $B_h[h] < B_{h+1}[h+1]$?

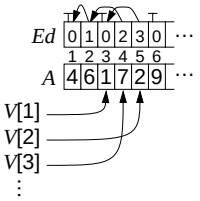
- vastaus: jos $B_h[h] \ge B_{h+1}[h+1]$, niin
  $B_h[h] \ge B_{h+1}[h+1] > B_{h+1}[h]$
  $\Rightarrow$ B_h ei olekaan pienimpään loppuva h:n pituinen nouseva sisäjono $\lightning$
  - siis $B_h[h] < B_{h+1}[h+1]$ pätee aina kun $1 \le h < k$
- tapaus $h = 0$: $A[i]$ lisätään jonon B_0 jatkeeksi jos ja vain jos se synnyttää jonon B_1 tai parantaa sitä
  - ts. $k = 0 \vee A[i] < B_1[1]$
- tapaus $h = k$: $A[i]$ lisätään jonon B_k jatkeeksi jos ja vain jos
  $k = 0 \vee A[i] > B_k[k]$
- $A[i]$ lisätään jonon B_h jatkeeksi jos ja vain jos
  $(\ h = 0 \vee B_h[h] < A[i]\ ) \wedge (\ h = k \vee A[i] < B_{h+1}[h+1]\ )$

$\Rightarrow$ $B_1[1] < \ldots < B_k[k]$ määrittelevät $k+1$ väliä, ja $A[i]$ lisätään
- yhteen niistä, jos $A[i] \ne B_h[h]$ kaikille $1 \le h \le k$
- ei mihinkään, jos $A[i] = B_h[h]$ jollekin $1 \le h \le k$

$\Rightarrow$ $A[i]$ lisätään korkeintaan yhden jonon jatkoksi

$\Rightarrow$ jonot voidaan esittää muistamalla jokaisen jonon viimeisen alkion paikka, ja liittämällä kohtaan i osoitin, joka kertoo jonon edellisen alkion paikan

- otetaan käyttöön taulukot $V[0\ldots n]$ ja $Ed[1\ldots n]$
  - $V[h]$ h-pituisen jonon viimeisen alkion paikka
  - $Ed[i]$ paikassa i olevan alkion edeltäjän paikka



- ohjelman helpottamiseksi sovitaan
  - $Ed[i] = 0$, kun edeltävää alkiota ei ole
  - $V[0]$ on olemassa ($\Rightarrow V[0] = 0$)
  (nämä voitaisiin välttää sopivilla **if**-lauseilla)
- jonojen B_h "lukemiseksi" V:stä ja Ed:stä määritellään
  - $Ed^0[x] = x$, jos $1 \le x \le n$
  - $Ed^{i+1}[x] = Ed[Ed^i[x]]$, jos $1 \le Ed^i[x] \le n$
  - $B_h =$
    $\langle\ A[Ed^{h-1}[V[h]]],\ \ldots,\ A[Ed^1[V[h]]],\ A[Ed^0[V[h]]]\ \rangle$

- edellä olevan perusteella
  $\langle\ \forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i-1]\ )$
  $\quad\ \wedge\ mns(B_k[1\ldots k],\ A[1\ldots i-1]) \wedge 1 \le i \le n\ \rangle$
  jos jollekin h pätee
  $\quad (\ h = 0 \vee A[V[h]] < A[i]\ ) \wedge$
  $\quad (\ h = k \vee A[i] < A[V[h+1]]\ )$
  niin suorita
  $\quad Ed[i] := V[h]$; $V[h+1] := i$;
  $\quad$ **if** $h = k$ **then** $k := k+1$ **endif**
  $\langle\ \forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i]\ )$
  $\quad\ \wedge\ mns(\ B_k[1\ldots k],\ A[1\ldots i]\ )\ \rangle$
- näillä keinoin ohjelma muuntuu muotoon

$\langle$ **true** $\rangle$
$k := 0$; $V[0] := 0$;
$\langle$ **inv:** $\forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i-1]\ )$
$\quad \wedge\ mns(\ B_k[1\ldots k],\ A[1\ldots i-1]\ )\ \rangle$
for $i := 1$ **to** n **do**
$\quad \langle$ **inv:** ?? $\rangle$
$\quad k' := k$
$\quad$ **for** $j := 0$ **to** k' **do**
$\quad\quad$ **if** $(\ j = 0$ **orelse** $A[i] > A[V[j]]\ ) \wedge$
$\quad\quad\quad (\ j = k$ **orelse** $A[i] < A[V[j+1]]\ )$ **then**
$\quad\quad\quad\ Ed[i] := V[j]$; $V[j+1] := i$;
$\quad\quad\quad\ $ **if** $j = k$ **then** $k := k+1$ **endif**
$\quad\quad$ **endif**
$\quad$ **endfor**
$\quad \langle\ \forall\ h\ ;\ 1 \le h \le k$: $plns(\ B_h[1\ldots h],\ A[1\ldots i]\ )$
$\quad\ \wedge\ mns(\ B_k[1\ldots k],\ A[1\ldots i]\ )\ \rangle$
endfor

$\langle\ mns(B_k[1\ldots k], A[1\ldots n])\ \rangle$
$v := V[k];$
$\langle$ **inv:** $mns(B_k[1\ldots k], A[1\ldots n]) \wedge v = Ed^{k-i}[V[k]] \wedge$
$\forall\ h\ ;\ i+1 \le h \le k\colon B[h] = B_k[h]\ \rangle$
for $i := k$ **downto** 1 **do**
$B[i] := A[v];\ v := Ed[v]$
endfor
$\langle\ mns(\ B[1\ldots k], A[1\ldots n]\ )\ \rangle$

Sisäjonokirjanpidon tehostaminen 2

- tiedämme, että $A[i]$ lisätään enintään yhden jonon jatkoksi

⇒ sisin **for**-silmukka tarvitaan vain oikean paikan etsimiseen — voisiko sen löytää tehokkaammin?

- voisi, puolitushaulla!

$\langle$ **true** $\rangle$
$k := 0;\ V[0] := 0;$
$\langle$ $\mathbf{inv_1}$**:** $\forall\ h\ ;\ 1 \le h \le k\colon plns(\ B_h[1\ldots h], A[1\ldots i-1]\ )$
$\wedge\ mns(\ B_k[1\ldots k], A[1\ldots i-1]\ )\ \rangle$
for $i := 1$ **to** n **do**
$a := 1;\ y := k+1;$
while $a < y$ **do**
$v := (a+y)$ div 2;
if $A[V[v]] < A[i]$ **then** $a := v+1$
else $y := v$
endif
endwhile
$\langle\ \mathbf{inv_1} \wedge 1 \le a \le k+1 \wedge$
$(\ a = 1 \vee A[V[a-1]] < A[i]\ ) \wedge$
$(\ a = k+1 \vee A[V[a]] \ge A[i]\ )\ \rangle$
if $a = k+1$ **orelse** $A[V[a]] > A[i]$ **then**
$Ed[i] := V[a-1];\ V[a] := i$

endif
if $a = k+1$ **then** $k := a$ **endif**
$\langle\ \forall\ h\ ;\ 1 \le h \le k\colon plns(B_h[1\ldots h], A[1\ldots i])$
$\wedge\ mns(\ B_k[1\ldots k], A[1\ldots i]\ )\ \rangle$
endfor
$\langle\ mns(\ B_k[1\ldots k], A[1\ldots n]\ )\ \rangle$
$v := V[k];$
$\langle$ **inv:** $mns(B_k[1\ldots k], A[1\ldots n]) \wedge v = Ed^{k-i}[V[k]] \wedge$
$\forall\ h\ ;\ i+1 \le h \le k\colon B[h] = B_k[h]\ \rangle$
for $i := k$ **downto** 1 **do**
$B[i] := A[v];\ v := Ed[v]$
endfor
$\langle\ mns(\ B[1\ldots k], A[1\ldots n]\ )\ \rangle$

Suoritusaika

- **while**-silmukka $O(\lg k) \le O(\lg n)$
- 1. **for**-silmukka $O(n)$ kierrosta
- 2. **for**-silmukka $O(k) \le O(n)$ kierrosta
- muut operaatiot vakioaikaisia

⇒ kaikkiaan $O(n \lg n)$

- odottamattoman nopea!

Formaaliuden asteen valinnasta

- sisäjono-ohjelman spesifikaatiossa ja todistuksessa tarvittiin aika pitkiä tilapredikaatteja
  - *mns* ja *plns*
  - vastaavat käsitteet eivät kovin monimutkaisia

⇒ tarvittavat päättelyt olivat työläitä formalisoida, vaikka eivät kovin monimutkaisia
⇒ emme formalisoineet kovin perusteellisesti

- formalismi on työkalu, ei itsetarkoitus!
  - ohjelman johto ja todistus eivät olisi ratkaisevasti kärsineet, vaikka *mns* ja *plns* olisi jätetty osittain formalisoimatta
  - formalisointi varmisti, että niiden merkitys on yksikäsitteinen

Sisäjono-ohjelman suunnittelun analyysia

- ohjelman kehitys oli yhdistelmä tilanteiden ja sisäjonojen ominaisuuksien huolellista analyysia ja tunnettujen algoritmisten keinojen käyttöä
  - ohjelmien todistustekniikat tukivat analyysia hyvin
- algoritmisten keinojen valinnassa aikaisempi kokemus oli kovasti avuksi
  - jonojen B_h esitys “taaksepäin”-osoittimilla
  - puolitushaun käyttö
- matemaattisessa käsittelyssä aikaisempi kokemus oli kovasti avuksi
  - tilapredikaattien ja invarianttien laadinta
  - funktion Ed^i käyttöön otto
  - sisäjonojen ominaisuuksien keksiminen
- joissakin kohdin edettiin arvaamalla
  - esim. “pala kerrallaan” -strategian valinta
  - arvaukset olisivat voineet epäonnistuakin
  - silloin olisi ollut tarpeen koettaa muitakin strategioita, esim. hajoita-ja-hallitse
  - strategia “dynaaminen ohjelmointi” olisi johtanut olennaisesti samaan ratkaisuun

- välillä käytettiin epärealistista “teoreettista” tietorakennetta
  - jonot B_h
  - riski: etukäteen oli aika varmaa, että ne voi toteuttaa jotenkin, mutta ei ollut varmaa, että ne voi toteuttaa riittävän tehokkaasti
- vaikka jonot B_h puuttuvat lopullisesta ohjelmasta, ne ovat sen toiminnan ymmärtämisen kannalta olennaisia
  ⇒ jätettiin lopullisiin tilapredikaatteihin
  - tilapredikaatit ovat hyvä keino dokumentoida ohjelmassa “piilevänä” oleva rakenteita

⇒ johtopäätöksiä
  - kokemusta ei korvaa mikään
  - matemaattinen järkeily on hyödyllinen työkalu
  - järkeily vaatii ohjelman toiminta-alueen lainalaisuuksiin perehtymistä
- perusteellisen järkeilyn jälkeen muutosten vaikutus ohjelmaan on helppo arvioida
  - harjoitustehtävät

Harjoitustehtäviä

1. (a) Emme muistaneet todistaa, että sisäjono-ohjelman 1. **for**-silmukan jokaisen kierroksen alussa
   $\forall\ h\ ;\ 0 \le h \le k\colon\ 0 \le V[h] < i$
   Korjaa unohdus.

   (b) Päteekö aina 1. **for**-silmukan kierroksen alussa
   $\forall\ j\ ;\ 1 \le j < i\colon\ 0 \le Ed[j] < i$

2. (a) Anna sisäjono-ohjelmalle mahdollisimman tarkka suoritusajan alaraja Θ-merkinnällä.

   (b) Voiko alustuksen $V[0] := 0$ jättää pois tekemättä ohjelmaan muita muutoksia? Muuttuuko tilanne, jos oletetaan, että kielessä on ajonaikaiset tarkistukset, jotka varmistavat, että kaikki sijoitettavat arvot ovat välillä 0, …, $n+1$? (Esim. Pascal-ohjelmaan saa tällaiset tarkistukset.)

   (c) Minkälaisten muutosten jälkeen alkion $V[0]$ voi jättää kokonaan pois?

3. Kirjoita ohjelma, joka tuottaa luvun k ja taulukon $B[1\ldots k]$ siten, että B on mahdollisimman pitkä kiinteän syötetaulukon $A[1\ldots n]$ ei-laskeva sisäjono. Perustele, että ohjelmasi toimii oikein.

4. (a) Taulukot $A[1\ldots n_A]$, $B[1\ldots n_B]$ ja $C[1\ldots n_C]$ ovat suuruusjärjestyksessä pienin ensin, ja on olemassa ainakin yksi alkio, joka on kaikissa kolmessa. Suunnittele todistuksineen ohjelma, joka etsii mahdollisimman pienen kaikissa kolmessa taulukossa olevan alkion.

   (b) Kuten (a), mutta nyt pitää löytää alkio, joka on taulukon $A[1\ldots m, 1\ldots n]$ jokaisella rivillä $A[i, 1\ldots n]$, kun jokainen rivi on suuruusjärjestyksessä pienin ensin.

5. Suunnittele todistuksineen ohjelma, joka etsii kiinteästä taulukosta $A[1\ldots m, 1\ldots n]$ mahdollisimman suuren 0-neliön (spesifikaatio löytyy kohdasta 4.2).