Rökstudd forritun í Java

Snorri Agnarsson
Raunvísindastofnun Háskólans
Háskóla Íslands
IS-107 Reykjavík

snorri@hi.is

Ágúst 1997

Efnisyfirlit

1 Formáli

Þessi grein fjallar um rökstudda forritun. Með rökstuddri forritun er átt við að forritstextinn innihaldi rökstuðning í formi athugasemda, sem gera sæmilega glöggum lesanda kleift að staðfesta að forritið geri það, sem til er ætlast.

Þessi útgáfa greinarinnar er þriðja meginútgáfan og er þessi útgáfa miðuð við forritunarmálið Java. Samsvarandi bæklingar hafa áður verið skrifaðir fyrir forritunarmálin Pascal og C++. Það breytir reyndar litlu um innihaldið hvort miðað er við Pascal, C++ eða Java, aðferðir þær, sem við notum til að rökstyðja okkar forritstexta eru þær sömu.

Bæklingur þessi, í ýmsum útgáfum, hefur verið notaður í kennslu í byrjendanámskeiði í forritun við Háskóla Íslands. Bæklingurinn er ekki ætlaður sem kennslubók um Java forritunarmálið, þótt nota megi hann til hliðsjónar við slíkan lærdóm.

2 Inngangur

Forritun tölva er í eðli sínu rökræn vinna. Forritun felst í að skilgreina rökrænt ferli, atburðarás, sem tölvan fylgir við að leysa okkar vandamál. Ferlið felst í skrefum aðgerða, sem tölvan framkvæmir.

Þegar byrjendur forrita hættir þeim til að einblína á aðgerðir þær, sem tölvan framkvæmir, án þess að skilgreina þær stöður (þau „ástönd“), sem tölvan er í milli þessara aðgerða.

Slík vinnubrögð eru röng og leiða til ótraustra forrita, sem í besta falli „virka“ stundum, eða „gera eitthvað,“ án þess að við getum treyst því að þau geri það, sem til er ætlast. Það er að sjálfsögðu nauðsynlegt að kunna góð skil á því hvernig þær aðgerðir verka, sem við látum tölvuna framkvæma, en við megum ekki missa sjónar af því að það eru ekki aðgerðirnar sjálfar heldur niðurstöðurnar úr þeim, sem eru okkar markmið í forritun.

Í þessari grein er reynt að gera grein fyrir mikilvægi þess að skilgreina stöður þær, sem tölvan er í milli aðgerða, og nota þær skilgreiningar sem skref í röksemdafærslu í okkar forritum.

2.1 Stöðulýsingar

Í rauninni má segja að í þessari grein sé aðeins eitt aðalumræðuefni, þ.e. *stöðulýsingar* í forritun. Stöðulýsingar þessar eru notaðar í ýmsum afbrigðum, en í flestum tilfellum eru þær skrifaðar sem athugasemdir í okkar forritstexta.

Eitt meginverkefni sérhvers, sem vill verða fær forritari er að öðlast færni í að lesa og skrifa slíkar stöðulýsingar. Þegar við skrifum forrit hljótum við um leið að reyna að sannfæra okkur sjálf um að forritið vinni eins og til er ætlast. Ef við getum sannfært okkur sjálf um að forritið virki þá hljótum við einnig að geta sannfært aðra lesendur forritsins. Ef svo er ekki er eitthvað að. Þá erum við annaðhvort að blekkja okkur sjálf (e.t.v. vinsælasta tómstundagaman mannkynsins) eða skortir þjálfun í að koma okkar hugsun á blað (næstvinsælasta umkvörtunarefni háskólakennara). Í báðum tilfellum er aukin vinna og þjálfun eina ráðið.

Nauðsynleg undirstaða færni í forritun er því færni í röksemdafærslu.

3 Athugasemdir í forritum

Setja skal athugasemdir á lykilstaði í öll forrit. Með athugasemdum (e. *comment*) er átt við texta, sem einungis er ætlaður fyrir mennska lesendur, þýðandinn tekur ekki mark á athugasemdum.

Athugasemdirnar skulu yfirleitt vera tengdar ákveðnu andartaki, þ.e. tímapunkti, í keyrslu forritsins. Þær skulu lýsa stöðunni eins og hún er á þessu tiltekna andartaki, og lýsa þar með tilgangi þess sem á undan hefur farið. Forðast skal athugasemdir sem einfaldlega lýsa því sem framkvæmt er, án þess að lýsa tilganginum. Lítum á dæmi um góðar og slæmar athugasemdir í sama forriti.

3.1 Slæmar athugasemdir

Í eftirfarandi forritum eru notaðar breyturnar `System.in` og `System.out`, sem báðar eru klasabreytur (*class variable*). Báðar þessar breytur er í `System` *klasanum*. `System.in` er af tagi `InputStream`, en `System.out` er af tagi `PrintStream`.

Vegna þess að `InputStream` klasinn býður ekki upp á innlestur á heiltölum (`int`) á textasniði, búum við til eintak, `cin`, af klasanum `StreamTokenizer` og notum þá breytu til innlestrar.

```
// Forrit v1.

// Eftirfarandi forrit les heiltölu af skjá og deilir í
// hana með 10, deilir síðan í afganginn með 5 og
// skrifar út útkomurnar og afganginn.

import java.io.*;

class v1 {

  static public void main( String[] argv ) throws IOException {

    int
      Upphaed,          // heiltala sem deilt er í
      Afgangur,         // afgangur úr deilingu
      Tikallar,         // útkoma úr fyrstu deilingu
      Fimmkallar,       // útkoma úr seinni deilingu
      Kronupeningar;    // afgangur úr seinni deilingu

    StreamTokenizer cin = new StreamTokenizer(System.in);
    PrintStream cout = System.out;

    cout.println("Sláðu inn upphæð sem skipta á í mynt.");
    cout.println("Upphæðin skal vera í heilum krónum, ");
    cout.print("milli 0 og 1000: ");
    if( cin.nextToken() != StreamTokenizer.TT_NUMBER )
       throw new IOException("Rangt snið á tölu");
```

```
    Upphaed = (int)cin.nval;   // lesum Upphaed af lyklaborði
    Tikallar = Upphaed / 10;   // deilum í Upphaed með 10
    Afgangur = Upphaed % 10;   // afgangur þegar deilt er með 10
    Fimmkallar = Afgangur / 5; // deilum með 5 og setjum
                               // útkomuna í breytuna Fimmkallar
    Kronupeningar = Afgangur % 5; // afgangur úr deilingu með 5
    // næst skrifum við gildin sem eru í breytunum Tikallar,
    // Fimmkallar og Kronupeningar
    cout.print("Upphæðinni ");
    cout.print(Upphaed);
    cout.println(" skal skipta í eftirfarandi:");
    cout.print(" Tíkallar:      ");
    cout.println(Tikallar);
    cout.print(" Fimmkallar:    ");
    cout.println(Fimmkallar);
    cout.print(" Krónupeningar: ");
    cout.println(Kronupeningar);
  }

}
```

Athugasemdirnar í forritinu hér að ofan eru gagnslausar. Þær endursegja einungis það, sem lesandinn sér hvort eð er með því að lesa forritið, en gefa ekki vísbendingu um tilgang þess sem framkvæmt er.

3.2 Góðar athugasemdir

```
// Forrit v2.

// Eftirfarandi forrit les heiltöluupphæð milli 0 og 1000 af
// lyklaborði tölvunnar og skrifar á skjáinn hvernig skipta
// megi upphæðinni í tíkalla, fimmkalla og krónupeninga,
// þannig að sem fæstar myntir verði notaðar.

import java.io.*;

class v2 {

  static public void main( String[] argv ) throws IOException {

    int
      Upphaed,    // upphæðin sem skipta skal
      Afgangur,   // milliniðurstöður
```

```
      Tikallar,   // fjöldi tíkalla sem nota skal
      Fimmkallar,    // fjöldi fimmkalla sem nota skal
      Kronupeningar; // fjöldi krónupeninga sem nota skal

    StreamTokenizer cin = new StreamTokenizer(System.in);
    PrintStream cout = System.out;

    cout.println("Sláðu inn upphæð sem skipta á í mynt.");
    cout.println("Upphæðin skal vera í heilum krónum, ");
    cout.print("milli 0 og 1000: ");
    if( cin.nextToken() != StreamTokenizer.TT_NUMBER )
       throw new IOException("Rangt snið á tölu");
    Upphaed = (int)cin.nval;
    // 0<=Upphaed<=1000
    Tikallar = Upphaed / 10;
    Afgangur = Upphaed % 10;
    // Tikallar>=0, Upphaed=10*Tikallar+Afgangur, 0<=Afgangur<10
    Fimmkallar = Afgangur / 5;
    Kronupeningar = Afgangur % 5;
    // Tikallar>=0, 0<=Fimmkallar<=1, 0<=Kronupeningar<=4,
    // og Upphaed=10*Tikallar+5*Fimmkallar+Kronupeningar
    cout.print("Upphæðinni ");
    cout.print(Upphaed);
    cout.println(" skal skipta í eftirfarandi:");
    cout.print(" Tíkallar:      ");
    cout.println(Tikallar);
    cout.print(" Fimmkallar:    ");
    cout.println(Fimmkallar);
    cout.print(" Krónupeningar: ");
    cout.println(Kronupeningar);
  }
}
```

Í forritinu að ofan lýsa athugasemdirnar því ástandi, sem náð hefur verið á því andartaki, sem athugasemdin lýsir. Tilgangur þess, sem framkvæmt hefur verið, er sá að ná fram ástandi því, sem athugasemdirnar lýsa.

3.3 Fleiri góðar athugasemdir

```
// Forrit v3.

// Eftirfarandi forrit les heiltöluupphæð milli 0 og 1000 af
// lyklaborði tölvunnar og skrifar á skjáinn hvernig skipta
// megi upphæðinni í fimmtíukalla, tíkalla, fimmkalla og
```

```
// krónupeninga, þannig að sem fæstar myntir verði notaðar.

import java.io.*;

class v3 {

  static public void main( String[] argv ) throws IOException {

    int
      Upphaed,    // upphæðin, sem skipta skal
      Afgangur,   // milliniðurstöður
      Tikallar,   // fjöldi tíkalla, sem nota skal
      Fimmkallar,    // fjöldi fimmkalla, sem nota skal
      Kronupeningar; // fjöldi krónupeninga, sem nota skal

    StreamTokenizer cin = new StreamTokenizer(System.in);
    PrintStream cout = System.out;

    cout.println("Sláðu inn upphæð sem skipta á í mynt.");
    cout.println("Upphæðin skal vera í heilum krónum, ");
    cout.print("milli 0 og 1000: ");
    if( cin.nextToken() != StreamTokenizer.TT_NUMBER )
       throw new IOException("Rangt snið á tölu");
    Upphaed = (int)cin.nval;
    // 0 <= Upphaed <= 1000
    Tikallar = Upphaed / 10; // fjöldi tíkalla í upphæðinni
    Afgangur = Upphaed % 10; // afgangur er það, sem eftir er
                     // þegar tíkallarnir hafa verið dregnir frá
    Fimmkallar = Afgangur / 5;  // fjöldi fimmkalla í afgangnum
    Kronupeningar = Afgangur % 5; // þegar fimmkallarnir hafa
                                  // verið dregnir frá eru
                                  // aðeins krónupeningar eftir
    // skrifum niðurstöðuna:
    cout.print("Upphæðinni ");
    cout.print(Upphaed);
    cout.println(" skal skipta í eftirfarandi:");
    cout.print(" Tíkallar:      ");
    cout.println(Tikallar);
    cout.print(" Fimmkallar:    ");
    cout.println(Fimmkallar);
    cout.print(" Krónupeningar: ");
    cout.println(Kronupeningar);
  }
```

```
}
```

4 Röksemdafærsla í forritum

Til þess að við getum treyst þeim forritum, sem við skrifum verðum við að geta sannað að þau séu rétt. Ekki nægir að prófa forritið með ýmsum gildum, vegna þess að slíkar prófanir geta ekki sýnt fram á að forritið vinni rétt fyrir önnur gildi; með slíkum prófunum er því aðeins hægt að sýna að forritið sé rangt, ekki að það sé rétt. Slíkar prófanir eru þó að sjálfsögðu nauðsynlegar til þess að auka og treysta álit okkar á forritinu; við viljum bæði sanna okkar forrit og prófa þau á raunverulegum gögnum.

Sem betur fer er auðveldara en sýnist að sanna forrit. Tækni sú, sem nota skal er vel þekkt. Sú tækni krefst þess að forritið og sönnun þess séu þróuð samhliða. Tækni þessi hjálpar einnig til við hönnun forritsins; með hjálp hennar má vinna forritshluta nokkuð óháð hverjum öðrum.

4.1 Forskilyrði og eftirskilyrði

Þessi tækni felst í því að skrifa fullyrðingar inn í forritið, sem lýsa stöðu útreikninga á lykilandartökum í vinnslunni. Hverja fullyrðingu skal vera unnt að sanna út frá þeim forsendum sem gefnar eru og öðrum fullyrðingum í forritinu. Röksemdafærslan í sönnun forritsins gengur þannig fyrir sig að á gefnu andartaki í keyrslu forritsins gerum við ráð fyrir að tiltekin upphafsfullyrðing sé rétt, framkvæmum því næst kafla í forritinu og sönnum að tiltekin lokafullyrðing sé þar með rétt. Upphafsfullyrðingin er forskilyrði kaflans og lokafullyrðingin er eftirskilyrði hans. Dæmi:

```
// forskilyrði:
//   Upphaed = 10*Tikallar+Afgangur,
//   0<=Afgangur<10
Fimmkallar = Afgangur / 5;
Afgangur = Afgangur % 5;
// eftirskilyrði:
//   Upphaed = 10*Tikallar+5*Fimmkallar+Afgangur,
//   0 <= Fimmkallar <= 1,
//   0 <= Afgangur <= 4
```

Hugmyndin er sú að unnt skuli vera að *sanna* forritskaflann, án þess að þörf sé að líta á aðra hluta forritsins. Við leyfum okkur þó oftast að vera sveigjanleg í þessari kröfu með því að gera ráð fyrir að lesandi forritsins hafi einhverja almenna vitneskju um forritið, sem ekki er tiltekin í hverri einustu fullyrðingu forritsins. Með reynslu ætti það að verða ljóst hversu mikið svigrúm má leyfa sér í þeim efnum.

Þegar við sönnum forrit þurfum við að hafa skýrar fullyrðingar, svipað og í forriti v2 að ofan. Þegar um erfiða forritskafla er að ræða er öruggast að nota nákvæmar og formlegar aðferðir. Oft erum við þó að vinna í tiltölulega einföldum forritsköflum, og þá látum við okkur nægja óformlegar athugasemdir, jafnvel eins og í forriti v3 að ofan.

4.1.1 Ritháttur

Oft viljum við fullyrða um forritskafla P að með tilteknu forskilyrði F valdi framkvæmd hans því að eftirskilyrði E verði fullnægt. Þá fullyrðingu táknum við með rithættinum

$$\{F\}P\{E\}$$

Í Java forritstexta skrifum við þetta yfirleitt í fleiri línum:

```
// F
P
// E
```

eða stundum í einni línu: `/* F */ P /* E */`. Yrðingarnar

$$\{F_1\}P_1\{F_2\}, \{F_2\}P_2\{F_3\} \ldots \{F_n\}P_n\{F_{n+1}\}$$

má skrifa í einu lagi sem

$$\{F_1\}P_1\{F_2\}P_2\{F_3\}P_3 \ldots \{F_n\}P_n\{F_{n+1}\}$$

4.2 Gildisveitingar

Gildisveitingar breyta stöðunni með því að gefa tiltekinni breytu nýtt gildi. Athugasemdirnar skulu lýsa tilgangi þess að gefa breytunni þetta nýja gildi. Athugasemdir, sem einungis endursegja það, sem lesandanum er augljóst eru verri en gagnslausar vegna þess að þær tefja hann við að lesa forritið, en hjálpa honum ekki að skilja það. Dæmi um slíka slæma athugasemd er:

```
Tikallar = Upphaed / 10; // deilum í Upphaed með 10 og
                         // setjum útkomuna í Tikallar
```

Betra væri:

```
Tikallar = Upphaed / 10; // Tikallar inniheldur fjölda
                         // tíkalla í Upphæðinni
```

Í þessu tilfelli væri sennilega betra að sleppa athugasemdinni alveg, þar eð nöfnin á breytunum lýsa tilganginum. Oft er tilgangur gildisveitingarinnar það augljós að óþarfi er að setja athugasemd með henni.

4.3 Draugabreytur

Oft kemur fyrir að við viljum fullyrða eitthvað um samband milli gilda, sem ekki eru lengur í neinni breytu. Þá getum við notað svokallaðar draugabreytur (*ghost variables*). Draugabreyta er einfaldlega breyta, sem ekki er til nema í kollinum á okkur. Við gefum draugabreytum gildi í athugasemdum og þær taka þátt í röksemdafærslunni á sama hátt og aðrar breytur. Í eftirfarandi dæmi er breytan Upphæð draugabreyta:

```
readln(Afgangur);
// Upphæð = Afgangur;
// nú vitum við að Upphæð=Afgangur
Tikallar = Afgangur / 10;
Afgangur = Afgangur % 10;
// nú vitum við að
//   Upphæð = 10*Tikallar + Afgangur,
//   og  0 <= Afgangur < 10
```

4.4 If-setningar

Í *if*-setningum getur framvinda forrits tekið tvær eða fleiri stefnur, þar sem stefnan ákvarðast af útkomunni úr einhverri röksegð (*boolean expression*). Við getum auðveldlega beitt almennri skynsemi til að ákvarða reglur þær, sem gilda um röksemdafærslu í *if*-setningunni. Lítum á *if*-setningu á eftirfarandi sniði:

```
// F
if(R) {
  S
}
else {
  T
}
// E
```

Hér er fullyrðingin F forskilyrði *if*-setningarinnar og fullyrðingin E er eftirskilyrði hennar. Eins og alltaf gerum við ráð fyrir að forskilyrðið F sé rétt áður en setningin er framkvæmd. Eins skulum við reikna með að útreikningurinn á skilyrðinu R hafi ekki áhrif á stöðuna (gildi breyta). M.ö.o. R hefur engar hliðarverkanir.

Forskilyrði kaflans S er þá $\{F\ og\ R\}$ og forskilyrði kaflans T er $\{F\ og\ ekki\ R\}$. Til þess að tryggt sé að eftirskilyrðið E standist eftir að *if*-setningin er framkvæmd verða yrðingarnar

$$\{F\ og\ R\}S\{E\}$$

og

$$\{F\ og\ ekki\ R\}T\{E\}$$

að vera sannanlegar.

Samsvarandi reglur um if setningu án else-hluta er álíka auðvelt að semja. Slík setning er jafngild if setningu eins og að ofan, með tóman kafla T. Almennt lítur slík setning svona út:

```
// F
if(R) {
  S
}
// E
```

Hér verða yrðingarnar $\{F\ og\ R\}S\{E\}$ og $F\ og\ ekki\ R \rightarrow E$ að vera sannanlegar.

Að sjálfsögðu er gagnslaust að reyna að læra ofantaldar reglur eins og páfagaukur. Nauðsynlegt er að skilja þær þannig að þær verði augljósar.

4.5 While-setningar

Í sönnun forrita er einna erfiðast að eiga við lykkjur. Vandamálið felst í því að við vitum yfirleitt ekki fyrirfram fjölda umferða, sem farnar verða um lykkjuna, og jafnvel þótt við vitum fjöldann getur vel verið að hann sé svo mikill að ekki sé fært að smíða röksemdafærslu, sem rekur í gegn um allar umferðir. Setjum upp almenna lykkju:

```
// F
while(R) {
  // I
  S
}
// E
```

Hvernig sönnum við réttmæti þessarar lykkju? Ef við vissum að skilyrðið R væri ósatt í byrjun þá nægði að sanna að forskilyrðið F leiddi af sér eftirskilyrðið E, en við vitum það ekki. Ef við vissum að R yrði rangt eftir fyrstu umferð þá nægði að sanna að setningarunan leiddi til þess að eftirskilyrðið yrði satt, að því tilskyldu að forskilyrðið væri satt í upphafi. En það vitum við ekki. Og svo framvegis. Við virðumst vera í klípu hér. En við höfum undankomuleið. Gerum ráð fyrir að við getum fundið fullyrðingu I, sem er sönn fyrir og eftir sérhverja umferð lykkjunnar, og er þeim kosti búin að þegar lykkjunni er lokið þá er eftirskilyrðið afleiðing hennar. Þá er okkur borgið. Þessa fullyrðingu köllum við fastayrðingu lykkjunnar. Hún gerir okkur kleift að sanna réttmæti lykkjunnar.

Til þess að röksemdafærsla í almennri *while*-setningu eins og að ofan sé í lagi þurfa fullyrðingarnar $F \rightarrow I$ og $\{I\ og\ R\}S\{I\}$ og $I\ og\ ekki\ R \rightarrow E$ að vera sannanlegar.

Lítum á dæmi (hér gerum við í huganum greinarmun á breytum með stórum og litlum stöfum):

```
// p=1, y=Y, x=X, y>=0
while(y!=0) {
  // X^Y = p*x^y,  y>=0
  p=p*x;
  y=y-1;
}
// X^Y = p
```

Þessi forritskafli sýnir okkur hversu kröftug og sannfærandi röksemdafærsla með fastayrðingu getur verið. Takið eftir að réttmæti kaflans má sannreyna með því að kanna röksemdafærsluna í stuttum skrefum, í mesta lagi tvær setningar í einu skrefi.

Fastayrðingu *while*-setningar skrifum við ávallt á undan fyrstu setningu í stofni lykkjunnar. Samt sem áður krefjumst við þess að hún gildi eftir síðustu umferð. Það væri því til dæmis rangt að hafa $y \neq 0$ inni í fastayrðingunni hér að ofan.

Eitt vandamál er óleyst, en það er eftirfarandi: Hvernig finnum við viðeigandi fastayrðingu? Við höfum ekkert einfalt svar við þeirri spurningu. Val á fastayrðingu er eitt erfiðasta vandamálið, sem við eigum við að glíma. Nokkrar hugmyndir má þó telja upp, sem hjálpa til við smíð fastayrðinga:

- Fastayrðinguna skal smíða áður en lykkjan er forrituð.
- Fastayrðing er almennari útgáfa af eftirskilyrði.
  - Oft má fá fastayrðingu með því að breyta fasta í eftirskilyrði í breytu.
  - Oft má fá fastayrðingu með því að breyta eftirskilyrði á sniðinu $\{A \; og \; B\}$ í fastayrðingu $\{A\}$.
  - Stundum má fá fastayrðingu með því að stækka gildissvið breytu í eftirskilyrði.

Þegar við skrifum lykkjur stýrir fastayrðingin að miklu leyti því hvernig við skrifum stofninn í lykkjuna. En þó ekki alveg — lítum á aðra lausn á sama vandamáli og að ofan:

```
// p=1, y=Y, x=X, y>=0
while(y!=0) {
  // X^Y = p*x^y,  y>=0
  if(y%2) {
    p=p*x;
    y=y-1;
  };
  // X^Y = p*x^y,  y>=0, y%2=0
  y=y/2;
  x=x*x;
};
// XY = p
```

Í forritskaflanum að ofan reiknum við X^Y á mjög hraðvirkan hátt. Forritskaflinn er flókinn þótt stuttur sé, og þarfnast vandlegrar skoðunar. Þetta dæmi sannfærir okkur ef til vill enn betur um gagnsemi fastayrðinga vegna þess hér höfum við flókna aðferð, sem er auðveldlega sönnuð með því að nota fastayrðingu. Hafa ber í huga hér að við erum ekki að fullsanna okkar forrit með þessari aðferð. Við sönnum einungis að ef inning forritsins tekur enda þá er niðurstaðan rétt. Seinna munum við íhuga hvernig við fullsönnum okkar forrit með því að sanna að inningin taki enda.

4.6 For-setningar

For-setningar eru að vissu leyti auðveldari viðfangs en *while*-setningar vegna þess að fjöldi umferða um lykkjuna er fyrirfram ákveðinn. Fastayrðing *for*-setningar segir til um ástandið í byrjun hverrar lykkju, eftir að stýribreytan hefur fengið það gildi, sem einkennir umferðina, en áður en nokkur setning innan lykkjunnar er framkvæmd. Dæmi:

```
// r=1
for( i=0 ; i!=n ; i++ ) {
  // r=x^i
  r=r*x;
}
// r=x^n
```

Nokkuð almenn uppsetningin á for lykkju er eftirfarandi:

```
// F
for( i=i1 ; i != iN+1 ; i++ ) {
  // G(i)
  S
}
// E
```

Eftirskilyrðið E verður að vera afleiðing $G(iN+1)$, því að iN+1 er númer þeirrar umferðar, sem kemur á eftir síðustu umferð, og er það gildi stýribreytunnar, sem veldur því að inning for lykkjunnar er stöðvuð.

Eins og í *while*-setningunni skrifum við ávallt fastayrðingu *for*-setningar á undan fyrstu setningu í stofni lykkjunnar, en samt sem áður krefjumst við þess að fastayrðingin sé sönn eftir lykkjuna, jafnvel þótt engin umferð sé farin í lykkjunni.

Til þess að almenna *for*-setningin að ofan sé réttmæt þurfa fullyrðingarnar $F \rightarrow G(i1)$ og $\{G(k)\ og\ i1 \leq k \leq iN\}S\{G(k+1)\}$ og $G(iN+1) \rightarrow E$ að vera sannanlegar.

Reyndar þarf einnig að athuga sérstaklega hvort hugsanlegt sé að i1 sé stærra en iN. Þá gilda augljóslega aðrar reglur því þá er engin umferð farin í lykkjunni og síðasta umferð er umferð númer i1-1, sem er ekki endilega jafnt iN. Í því tilfelli verður yrðingin $F \rightarrow E$ að vera sannanleg. Best er þó að sleppa því að skrifa slíkar *for*-setningar nema hægt sé að beita almennu reglunni að ofan. Til þess að það sé hægt nægir að sjá til þess að $i1 \leq iN+1$.

4.7 Inntakssetningar

Í inntakssetningum lesum við einhver gildi í tilteknar breytur. Á eftir inntakssetningum er því viðeigandi að setja fullyrðingar um innihald breytanna. Ef við höfum sett skilyrði um notkun forritsins mega þau skilyrði koma fram í skilyrðum um gildi þau, sem breyturnar fá. Við sáum dæmi um slík skilyrði í forritinu, sem notað var í dæmum 1 til 3 að ofan.

4.8 Úttakssetningar

Oftast er hreinn óþarfi að setja athugasemdir með úttaksskipunum vegna þess að augljóst er hvað verið er að skrifa, og úttaksskipanir breyta yfirleitt ekki stöðunni á neinn þann hátt, sem erfitt er fyrir lesandann að skilja. Fyrir kemur þó að um flókið úttak að ræða, og ástæða til að lýsa ástandi útskrifaðra gagna. Lítum á dæmi:

```
// Forrit prim.

// Eftirfarandi forrit skrifar alla prímþætti
// tölunnar 12345 eins oft og þeir koma fyrir

class prim {

  static public void main( String[] argv ) {

    int i,j;

    i=12345;
    j=2;
    // í eftirfarandi forritskafla munum við
    // ávallt sjá til þess að i sé jákvæð
    // heiltala, sem gengur upp í 12345.
    while(i!=1) {
      // allir prímþættir 12345/i hafa verið
      // skrifaðir eins oft og þeir koma fyrir
      // og engin prímtala minni en j gengur upp í i
      while(i%j != 0) {
        // sama og að ofan, og auk þess er i!=1
        j=j+1;
      }
      // j er minnsta prímtala, sem gengur upp í i
      System.out.println(j);
      i=i/j;
    }
  }

}
```

4.9 Köll og boð

Nauðsynlegt er að samband sé milli þeirra fullyrðinga, sem lýsa boði og fullyrðinga þeirra, sem lýsa tilteknu kalli á boðið. Eins og aðrar setningar í sönnuðu forriti hafa kallsetningar forskilyrði og eftirskilyrði. Það er því ljóst að boðinu þurfa að fylgja samsvarandi forskilyrði og eftirskilyrði. Forskilyrði þau og eftirskilyrði, sem fylgja boðinu þurfa að vera frumútgáfur allra þeirra forskilyrða og eftirskilyrða sem fylgja kalli á viðkomandi boð. Lítum á dæmi:

```
// Forrit rada.

// Forrit þetta les línu af aðalinntaki og skrifar á
```

```
// aðalúttak línu með sömu stöfum röðuðum í stafrófsröð.

import java.io.*;

class rada {

  // Notkun: Rada(s,i,j);
  // Fyrir:  s er StringBuffer, sem inniheldur
  //         a.m.k. sæti i og j.
  // Eftir:  gildunum í s[i] og s[j] hefur verið
  //         raðað þannig að s[i]<=s[j].
  static void Rada(StringBuffer s, int i, int j) {
    char t;
    if( s.charAt(i) > s.charAt(j) ) {
      // s[i] > s[j]
      t=s.charAt(i);
      s.setCharAt(i,s.charAt(j));
      s.setCharAt(j,t);
      // Nú höfum við víxlað gildunum í s[i] og s[j]
      // þannig að s[i] < s[j]
    }
    // s[i] <= s[j]
  }

  public static void main(String[] argv) throws IOException {

    String line;
    StringBuffer s;
    int i,j,len;

    line = new DataInputStream(System.in).readLine();
    s = new StringBuffer(line);

    len = line.length();  // framvegis verður len==line.length()
    for( i=0 ; i!=len ; i++ ) {
      // s[0..i-1] inniheldur minnstu stafi s í vaxandi röð
      for( j=i+1 ; j!=len ; j++ ) {
        // s[0..i-1] inniheldur minnstu stafi s í vaxandi röð
        // og s[i] inniheldur minnsta staf s[i..j-1]
        Rada(s,i,j);
      }
      // s[0..i] inniheldur minnstu stafi s í vaxandi röð
    }
```

```
    // strengurinn s er í vaxandi stafaröð
    System.out.println(s.toString());
  }
}
```

Boð, hvort sem um er að ræða venjuleg boð eða klasaboð, hjálpa okkur að skrifa smærri og læsilegri forrit á ýmsan hátt. Nota má sama boð oftar en einu sinni, en það nægir að skrifa það og sanna einu sinni. Einnig getum við notað boð á sama hátt og hjálparsetningar eru notaðar í stærðfræði til að safna saman á einn stað í forritinu vinnslu sem hefur heilstæða röksemdafærslu.

Lítum nú á annað dæmi, og notum endurkvæmt klasaboð í þetta skipti. Fibonacci talnarunan er skilgreind á eftirfarandi hátt fyrir jákvæðar heiltölur n:

$$f(n) = \begin{cases} 1 & \text{ef } n = 1 \text{ eða } n = 2 \\ f(n-1) + f(n-2) & \text{annars} \end{cases}$$

Við skulum skrifa einfalt forrit, sem reiknar Fibonacci tölur.

```
// Forrit Fibo1

// Forrit þetta skrifar 10 fyrstu Fibonacci tölurnar.

class fibo1 {

  // Notkun:        x=Fibo(n)
  // Forskilyrði:   x er integer breyta, n er integer gildi, n>0
  // Eftirskilyrði: x inniheldur n-tu Fibonacci töluna
  static int Fibo( int n ) {
    // n>0
    if(n<=2)
      return 1;
    else
      // n er stærra en 2, þar með eru eftirfarandi köll
      // lögleg samkvæmt forskilyrði fallsins, og reikna
      // n-tu Fibonacci tölu
      return Fibo(n-1)+Fibo(n-2);
  }

  public static void main( String[] argv ) {

    int i;

    for( i=0 ; i!=10 ; i++ ) {
      // Fyrstu i Fibonacci tölurnar hafa verið skrifaðar
      System.out.println( "Fibo(" + (i+1) + ")=" + Fibo(i+1) );
```

```
    }
    // Fyrstu tíu Fibonacci tölurnar hafa verið skrifaðar
  }
}
```

Takið eftir að við gerðum engar sérstakar ráðstafanir í okkar fullyrðingu þótt klasaboðið Fibo væri endurkvæmt. Í aðferðinni fyrir Fibo boðið eru tvö endurkvæm köll á Fibo sjálft, og við gerðum einfaldlega ráð fyrir að endurkvæmu köllin ynnu rétt samkvæmt lýsingu boðsins.

Það kemur mönnum ef til vill á óvart að þetta sé rökrétt, en sú er raunin. Þegar við sönnum að eitthvert boð vinni samkvæmt lýsingu þeirri, sem felst í forskilyrði og eftirskilyrði þess megum við gera ráð fyrir að öll þau boð sem kallað er á úr aðferð boðsins vinni rétt, þar með talið boðið sjálft.

Hér virðist vera um að ræða röksemdafærslu í hring, en svo er þó raunverulega ekki þegar betur er að gáð. Hafa ber í huga að við erum einungis að hálfsanna forritin. Við sönnum einungis að ef vinnsla forritsins tekur enda þá verði niðurstaðan rétt. Ef um er að ræða röksemdafærslu í hring þá mun það koma fram í því að einhvers staðar verður til óendanleg lykkja eða óendanleg endurkvæmni í framkvæmd forritsins.

Hálfsönnuð forrit köllum við naumrétt (*partially correct* á ensku). Ef við höfum einnig sannað að vinnsla forritsins taki alltaf enda segjum við forritið vera rammrétt (*totally correct* á ensku). Auðvelt er að skrifa naumrétt forrit, ef þau þurfa ekki að vera rammrétt. Eftirfarandi forrit er naumrétt forrit til að reikna Fibonacci tölur:

```
// forrit Fibo2;

// Forrit þetta er naumrétt en ekki rammrétt.
// Tilgangur þess er að skrifa 10 fyrstu Fibonacci tölurnar,
// en það gerist ekki vegna þess að í forritinu er óendanleg
// endurkvæmni.  Að öðru leyti er röksemdafærslan í þessu
// forriti rétt, því forritinu keyrir ekki til enda án þess
// að hafa skrifað 10 fyrstu Fibonacci tölurnar.

class fibo2 {

  // Notkun:        x=Fibo(n)
  // Forskilyrði:   x er integer breyta, n er integer gildi, n>0
  // Eftirskilyrði: x inniheldur n-tu Fibonacci töluna
  static int Fibo( int n ) {
    // n > 0
    return Fibo(n+2)-Fibo(n+1);
  }

  public static void main( String[] argv ) {

    int i;
```

```
    for( i=0 ; i!=10 ; i++ ) {
      // Fyrstu i Fibonacci tölurnar hafa verið skrifaðar
      System.out.println( "Fibo(" + (i+1) + ")=" + Fibo(i+1) );
    }
    // Fyrstu tíu Fibonacci tölurnar hafa verið skrifaðar
  }
}
```

Forritið að ofan er greinilega ekki rammrétt. Í því er óendanleg endurkvæmni. Reyndar þurfum við lítið að leggja á okkur til að skrifa naumrétt forrit. Eftirfarandi forrit er einnig naumrétt forrit en ekki rammrétt:

```
// forrit Fibonacci 3

// Forrit þetta er naumrétt en ekki rammrétt.
// Tilgangur þess er að skrifa 10 fyrstu Fibonacci tölurnar,
// en það gerist ekki vegna þess að í forritinu er óendanleg
// lykkja.  Að öðru leyti er röksemdafærslan í þessu forriti
// rétt, því forritið keyrir ekki til enda án þess að hafa
// skrifað 10 fyrstu Fibonacci tölurnar.

class fibo3 {
  public static void main( String[] argv ) {

    int i;

    for( i=0 ; i!=10 ; ) {
      // skrifaðar hafa verið i fyrstu Fibonacci tölurnar, i<=10
      ;  // gerum ekkert í hverri umferð!
    }
    // tíu fyrstu Fibonacci tölurnar hafa verið skrifaðar
  }
}
```

5 Rammrétt forrit

Dæmin að ofan ættu að sannfæra okkur um að ekki nægi að forrit okkar séu naumrétt, þau verða að vera rammrétt. Við ættum samt að hafa í huga að við getum leyst þessi tvö vandamál að miklu leyti óháð hvort öðru. Við getum sannað að forrit sé naumrétt án þess að sanna að inning þess taki enda, og við getum sannað að inningin taki enda án þess að sanna að niðurstöðurnar séu réttar. Þetta er stór kostur. Það er ávallt gleðiefni þegar við getum skipt stóru vandamáli upp í tvö smærri.

Til að sanna að tiltekið forrit sé rammrétt þurfum við að sanna að hver einasta lykkja í forritinu taki enda, og að sérhvert kall á boð taki enda. Röksemdafærslurnar, sem notaðar eru, eru oftast eitthver

afbrigði af þrepasönnun. Tökum réttu útgáfuna af Fibonacci forritinu sem dæmi. Til þess að sanna að sérhvert kall á boðið Fibo taki enda notum við eftirfarandi þrepun:
Þrepasönnun. Við notum þrepun á n í kallinu Fibo(n) til þess að sanna að kallið taki enda fyrir öll jákvæð n.

Grunnur: $n = 1$ eða $n = 2$. Fyrir $n = 1$ og $n = 2$ er ljóst að kallið tekur enda.

Þrepun: $n > 2$.

Þrepunarforsenda: Við gerum ráð fyrir að öll lögleg köll á Fibo með viðfangi minna en n taki enda.

Þrepunarskref: Ljóst er að kallið Fibo(n) tekur enda ef köllin Fibo($n-1$) og Fibo($n-2$) taka enda. En við vitum að þau taka enda samkvæmt þrepunarforsendu.

Sönnun lokið.

Stundum tekur þrepunin önnur form en við eigum að venjast, eins og í eftirfarandi forriti:

```
class minnka {
  // Notkun:         Minnka(x,y)
  // Forskilyrði:    x og y eru integer breytur, sem
  //                 innihalda gildi sem eru jákvæð eða núll.
  //                 A.m.k. önnur breytan inniheldur gildi,
  //                 sem er stærra en núll.
  // Eftirskilyrði: Gildin í parinu (x,y) hafa verið minnkuð
  //                 í lesröð.  Með öðrum orðum:
  //                    Annaðhvort er x minna en það var (y
  //                    getur þá verið hvað sem er, >=0) eða
  //                    x er jafnt og það var og y er minna
  //                    en það var.  Bæði gildin eru ennþá
  //                    jákvæð eða núll.
  static void Minnka( int &x, int &y ) {
    // við sleppum stofninum hér, þar eð hann kemur
    // ekki röksemdafærslunni við
    ...
  }

  void main() {
    x=10;
    y=10;
    while(x!=0) {
      // x>=0 og y>=0
      Minnka(x,y);
    }
  }
```

Takið eftir að ekki er unnt að segja fyrir um fjölda umferða, sem farnar verða um lykkjuna. Sanna má þó með tvöfaldri þrepun að inning þessa forrits taki enda (prófið það—það er góð þjálfun).

Einnig má nota eilítið annars konar röksemdafærslu. Gildin, sem breytuparið (x,y) fær í hverri lykkju mynda minnkandi röð samkvæmt röðun, sem er svipuð og venjuleg stafrófsröð tveggja stafa orða. Nú vill svo til að ekki er til nein slík óendanleg runa í minnkandi röð. Þar með hlýtur inning forritsins að taka enda.

Þessi röksemdafærsla byggist á því að við vitum fyrirfram að engin slík óendanleg runa er til, en það getum við sannað í eitt skipti fyrir öll (til dæmis með þrepasönnuninni, sem þið gerðuð vonandi að ofan) og notað niðurstöðuna aftur og aftur í okkar forritum.

Mengi, sem hafa slíka röðun, sem ekki leyfir óendanlegar minnkandi runur köllum við vel grundvölluð mengi. Slík mengi koma oft fyrir í okkar forritum. Til dæmis er mengi náttúrlegra talna vel grundvallað mengi.

6 Einingaforritun

Þegar við skrifum forrit verðum við oft að hafa mörg atriði í huga samtímis. Eftir því sem við verðum að hafa fleiri atriði í huga aukast líkur á villum. Til þess að minnka þær líkur er því nauðsynlegt að takmarka fjölda og flækju þeirra atriða, sem hugsa þarf um í einu.

Til þess að auðvelda okkur lífið skiptum við því okkar forritum upp í sjálfstæðar einingar, sem eru sem mest óháðar hver annari. Hver eining hefur sína lýsingu, sem inniheldur allar þær upplýsingar sem á þarf að halda til að geta notað hana, og jafnframt allar þær upplýsingar, sem á þarf að halda til að geta smíðað hana.

6.1 Upplýsingahuld

David Parnas setti fyrir allnokkru fram kenningu um það hvernig skipta ætti hugbúnaðarkerfum í einingar [Parnas 72a, Parnas 72b]. Meginreglan, sem hann lagði fram, hefur verið kölluð *reglan um upplýsingahuld* (*information hiding*).

Regla þessi segir að skipta skuli verkefnum í einingar á þann hátt að sérhver afdrifarík hönnunarákvörðun, sem hugsanlegt er að breytist, sé einangruð í einhverri einingu. Til dæmis gætum við þarfnast taflna í okkar forriti, þannig að unnt sé að setja gildi í töfluna undir einhverjum lykli (t.d. streng), og sækja seinna gildi geymt undir þeim lykli. Slíka töflu má geyma á ýmsa vegu. Það er þá eðlilegt að einangra töflurnar í einingu ef geymslumátinn er afdrifarík hönnunarákvörðun. Reynslan sýnir að miklu fleiri hönnunarákvarðanir eru afdrifaríkar en virðist við fyrstu sýn. Það er því mikilvægt að fela sem flestar hönnunarákvarðanir til þess að gefa okkur seinna frjálsari hendur í breytingum og viðhaldi.

Við getum beitt þessu lögmáli án þess að taka strax hönnunarákvörðunina. Við lýsum þá okkar einingum fyrirfram á þann hátt að unnt sé að smíða þær á sem flesta vegu. Lýsingar okkar fullnægja þá eftirfarandi skilyrðum:

- Notanda einingar skal gefa nægilega miklar upplýsingar til að nota eininguna, en *ekki meiri upplýsingar*.

- Smið einingar skal gefa nægilega miklar upplýsingar til að smíða eininguna, en *ekki meiri upplýsingar*.

6.2 Huglæg gagnamót

Þegar við hugsum um eiginleika gagna viljum við einbeita okkur að þeim eiginleikum gagnanna, sem máli skipta. Þegar við reiknum með fleytitölur, til dæmis, viljum við oftast ekki þurfa að taka tillit til þess hvernig fleytitölurnar eru geymdar í minni vélarinnar. Okkur nægir að vita hvaða aðgerðir eru tiltækar fyrir fleytitölurnar, og hvaða huglæga eiginleika fleytitölurnar hafa.

Í reynd eru fleytitölurnar í flestum tilfellum hugsaðar sem nálgun rauntalna. Lykilorðið hér er *hugsaðar*, það, sem okkur skiptir máli, eru *huglægir eiginleikar* fleytitalnanna.

Svipaða einföldun getum við vel notað í fleiri tilfellum. Oft getum við lýst á huglægan eða sértækan (*abstract*) hátt þeim eiginleikum okkar gangamóta (*data type*), sem skipta okkur máli.

6.3 Hlutbundin forritun

Í hlutbundinni forritun getur forritarinn skilgreint að tiltekið hluttag (*object type*) T *erfi* frá öðru hluttagi S. Þetta þýðir að hlutur af tagi T er einnig af tagi S. Til þess að unnt sé að sanna slík forrit verður að sjá til þess að öll boð (*message*), sem senda má til hluta af tagi T, fullnægi þeim lýsingum, sem gefnar eru fyrir tagið S.

Lítum á dæmi: Gerum ráð fyrir að hluttagið MYND hafi undirtög KASSI og HRINGUR. Gildi af tagi MYND standi fyrir teikningu á skjánum. Gerum einnig ráð fyrir að boðið FLYTJA sé skilgreint fyrir hluti af tagi MYND, og endurskilgreint fyrir tögin KASSI og HRINGUR. Í lýsingu boðsins í taginu MYND segir að boðið valdi því að myndin færist á skjánum. Sú lýsing verður þá einnig að vera rétt fyrir tögin KASSI og HRINGUR. Ef svo væri ekki þá gætum við ekki skrifað boð, sem tæki viðfang af tagi MYND (sem gæti þá einnig verið af undirtaginu KASSI eða HRINGUR), og færði myndina.

6.4 Quicksort

Röðunaraðferðin *quicksort* var fundin upp af C. A. R. Hoare. Hún er með hraðvirkustu aðferðum, sem þekktar eru til að raða fylkjum. Lítum á quicksort boð.

```
// Forrit quicksort.

// Forrit þetta les textalínur af aðalinntaki og skrifar
// þær í vaxandi stafrófsröð á aðalúttak.

import java.io.*;
import java.util.*;

class quicksort {
```

```
static Random r = new Random();

// Notkun: i = random();
// Eftir:  i er ný jákvæð slembiheiltala.
static int random() {
  int i=r.nextInt();
  if( i<0 ) i=-i;
  return i;
}

// Notkun: vixla(f,x,y)
// Fyrir:  f er fylki strengja, með sæti x og y.
// Eftir:  Strengjunum í f[x] og f[y] hefur verið víxlað
static void vixla( String[] f, int x, int y ) {
  String t;
  t=f[x]; f[x]=f[y]; f[y]=t;
}

// Notkun: skipta(f,i,j,k);
// Fyrir:  f[i..j-1] er svæði í f, i<j,  k er tveggja sæta
//         fylki af int.
// Eftir:  i<=k[0]<=k[1]<=j,
//         Strengjunum í f[i..j-1] hefur verið umraðað þar til
//         eftirfarandi gildir:
//          1) Svæðið f[k[0]..k[1]-1] er ekki tómt og öll sæti
//             innihalda sama streng, p.
//          2) Öll stök í svæðinu f[i..k[0]-1] innihalda strengi,
//             sem eru minni en p (framar í stafrófsröð).
//          3) Öll stök í svæðinu f[k[1]..j-1] innihalda strengi,
//             sem eru stærri en p.
//
//         Þessu má lýsa í mynd á eftirfarandi hátt:
//          | <p | =p | >p |
//            ^    ^    ^    ^
//            i   k[0] k[1]  j
static void skipta( String[] f, int i, int j, int[] k ) {
  vixla(f,i,i+random()%(j-i)); // veljum slembistak sem p
  String p=f[i];
  int q=i,r=i+1,m=j;
  while( r != m ) {
    // | <p | =p | ?? | >p |
    //  ^    ^    ^    ^    ^
    //  i    q    r    m    j
```

```
    int compare = f[r].compareTo(p);
    if( compare < 0 )
      vixla(f,q++,r++);
    else if( compare == 0 )
      r++;
    else
      vixla(f,--m,r);
  }
  k[0]=q; k[1]=r;
}

// Notkun: rada(f,i,j);
// Fyrir:  f er fylki strengja, i og j eru vísar í f, i<=j
// Eftir:  Svæðinu f[i..j-1] hefur verið raðað í vaxandi röð
static void rada( String[] f, int i, int j ) {
  if(j-i < 2) return;
  int[] k=new int[2];
  skipta(f,i,j,k);
  rada(f,i,k[0]); rada(f,k[1],j);
}

public static void main( String[] argv ) throws IOException {
  DataInputStream cin = new DataInputStream(System.in);
  Vector v = new Vector();
  int n = 0;
  String line = cin.readLine();
  while( line != null ) {
    v.addElement(line);
    n++;
    line = cin.readLine();
  }
  String[] f = new String[n];
  for( int i=0 ; i!=n ; i++ ) {
    // f[0..i-1] inniheldur strengina úr v[0..i-1]
    f[i] = (String)v.elementAt(i);
  }
  rada(f,0,n);
  for( int i=0 ; i!=n ; i++ ) {
    // Búið er að skrifa á aðalúttak i fremstu strengina,
    // hinir eru í stafrófsröð í f[i..n-1].
    System.out.println(f[i]);
  }
}
```

```
}
```

7 Biðraðir og merge-sort

Í þessum kafla er rætt um biðraðir og hvernig nota megi þær til að útfæra *merge sort* röðunaraðferðina.

7.1 Biðraðir

Biðraðir eru gagnamót, sem einkennast af tveimur aðgerðum:

- *Put* aðgerðin bætir einu gildi í biðröðina, og fer það aftast í biðröðina.
- *Get* aðgerðin fjarlægir eitt gildi úr biðröðinni, og er það fremsta gildið.

7.1.1 Skil fyrir biðraðir

Eftirfarandi forritstexti skilgreinir skil (e. *interface*) fyrir biðraðir strengja.

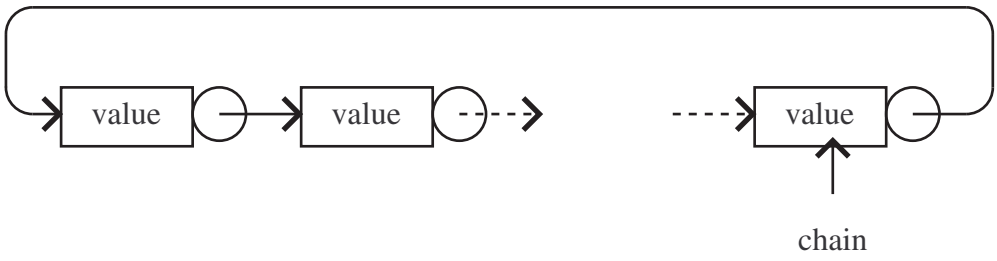
```
// Þessi skil skilgreina biðraðir strengja.
interface StringQueue {

    // Notkun: q.put(x);
    // Fyrir:  q er ekki full.
    // Eftir:  Búið er að bæta x aftan á q.
    public void put( String x );

    // Notkun: s = q.get();
    // Fyrir:  q er ekki tóm
    // Eftir:  s er strengurinn, sem var fremst í q, og hann
    //         hefur verið fjarlægður úr q.
    public String get();

    // Notkun: s = q.peek();
    // Fyrir:  q er ekki tóm
    // Eftir:  s er strengurinn, sem er fremst í q.
    public String peek();

    // Notkun: n = q.count();
    // Eftir:  n er fjöldi staka í q.
    int count();

    // Notkun: b = q.isFull();
    // Eftir:  b er satt þþaa q sé full
```

```
    boolean isFull();
}
```

7.1.2 Biðraðir útfærðar með lista

Forritstextinn hér á eftir inniheldur skilgreiningu á biðröð strengja, sem útfærð er með eintengdum lista.

Í klasa þessum er notaður eintengdur listi, sem tengdur er í hring, eins og þessi mynd sýnir:




```
import StringQueue;

class link {
  String value;
  link next;
}

class QueueList implements StringQueue {

  link chain;
  int n;
  // Fastayrðing gagna:
  //     n er fjöldi staka í biðröðinni.
  //     Ef biðröðin er tóm þá er chain núll, annars
  //     bendir chain á aftasta hlekk í hringkeðju hlekkja,
  //     sem innihalda gildi biðraðarinnar, samkvæmt
  //     eftirfarandi mynd, þar sem v1 er fremsta gildi,
  //     v2 er næstfremsta, o.s.frv., allt að vN, sem er
  //     aftasta gildi.  chain bendir á aftasta hlekk.
  //
  //      ------------------------------
  //     |                              |
  //     ->|v1|.|->|v2|.|->...->|vN|.|-
  //                               ^
  //                               |
```

```
//                                chain

// Notkun: QueueList q = new QueueList();
// Eftir:  q er ný tóm biðröð strengja, með pláss fyrir
//         ótakmarkaðan fjölda, meðan minnisrými leyfir.
public QueueList() {
  chain=null;
  n=0;
}

public String get() {
  String result = chain.next.value;
  if( chain.next == chain )
    chain = null;
  else
    chain.next = chain.next.next;
  n--;
  return result;
}

public String peek() {
  return chain.next.value;
}

public void put( String x ) {
  link temp = new link();
  temp.value = x;
  if( chain != null ) {
    temp.next = chain.next;
    chain.next = temp;
    chain = temp;
  }
  else {
    chain = temp;
    temp.next = temp;
  }
  n++;
}

public int count() {
  return n;
}
```

```
  public boolean isFull() {
    return false;
  }
}
```

7.1.3 Biðraðir útfærðar með fylki

Eftirfarandi forritstexti inniheldur skilgreiningu á biðröð, sem útfærð er með fylki.

```
import StringQueue;

class QueueArray implements StringQueue {

    String[] f;
    int max;
    int out,n;
    // Fastayrðing gagna:
    //     f er fylki af stærð max.
    //     Gildin í biðröðinni eru í f[out], f[(out+1)%max], ...,
    //     f[(out+n-1)%max], frá fremsta til aftasta.  Fjöldi
    //     staka er því n.
    //     0 <= out < max, 0<= n <= max

    // Notkun: QueueArray q = new QueueArray(max);
    // Fyrir:  max >= 0.
    // Eftir:  q er ný tóm biðröð strengja, með pláss
    //         fyrir max strengi.
    public QueueArray(int max) {
      this.max=max;
      f = new String[max];
      out=0; n=0;
    }

    public String get() {
        String result = f[out++];
        out = out%max;
        n--;
        return result;
    }

    public String peek() {
        return f[out];
    }
```

```
    public void put( String x ) {
        f[(out+n++)%max] = x;
    }

    public int count() {
        return n;
    }

    public boolean isFull() {
        return n==max;
    }
}
```

7.2 Merge sort

Eftirfarandi forritstexti inniheldur skilgreiningu á klasanum `MergeSort`, sem inniheldur klasaboðin `sort`, `split` og `merge`. Þau útfæra *merge sort* með hjálp biðraðaklasa. Klasaboð (*static* boð) eru ekki send til tiltekins hlutar. Þau eru fastbundin og í aðferðinni fyrir boðið er enginn „núverandi hlutur“, þ.e. enginn **this** hlutur.

Takið eftir að hér eru boðin öll skilgreind þannig að biðraðirnar eru af almennasta tagi, þ.e. `queuebase`.

```
import StringQueue;
import QueueList;

class MergeSort {

    // Notkun: MergeSort.split(q,q1,q2);
    // Fyrir:  q1 og q2 hafa hvert um sig pláss fyrir a.m.k.
    //         helming strengjanna í q.
    // Eftir:  Allir strengir hafa verið fjarlægðir úr q, og
    //         helmingur þeirra er í q1 og hinn helmingurinn
    //         í q2.
    //         Það munar í mesta lagi einum á fjölda staka í
    //         q1 og q2.
    public static void split( StringQueue q
                            , StringQueue q1
                            , StringQueue q2
                            )
    {
        while( q.count() != 0 ) {
            // Búið er að fjarlægja núll eða fleiri gildi úr
            // q, og nákvæmlega helmingur þeirra gilda er í
            // q1, en hinn helmingurinn í q2.
```

```
        q1.put(q.get());
        if( q.count() == 0 ) return;
        q2.put(q.get());
    }
}

// Notkun: MergeSort.merge(q1,q2,q);
// Fyrir:  q1 og q2 innihalda strengi í vaxandi stafrófsröð,
//         q er tóm og hefur pláss fyrir öll gildi í q1 og q2.
// Eftir:  Allir strengir hafa verið fjarlægðir úr q1 og q2,
//         og eru þeir í q í vaxandi stafrófsröð.
public static void merge( StringQueue q1
                        , StringQueue q2
                        , StringQueue q
                        )
{
    while( q1.count() > 0 && q2.count() > 0 ) {
        // Búið er að fjarlægja núll eða fleiri minnstu strengi
        // úr q1 og q2, og eru þeir í q, í vaxandi röð.
        if( q1.peek().compareTo(q2.peek()) < 0 )
            q.put(q1.get());
        else
            q.put(q2.get());
    }
    while( q1.count() > 0 ) {
        // Sama og að ofan, en auk þess er annaðhvort
        // q1 eða q2 tóm
        q.put(q1.get());
    }
    while( q2.count() > 0 ) {
        // Sama og að ofan, en auk þess er q1 tóm
        q.put(q2.get());
    }
}

// Notkun: MergeSort.sort(q);
// Fyrir:  q er einhver biðröð strengja.
// Eftir:  búið er að raða q í vaxandi stafrófsröð.
public static void sort( StringQueue q ) {
    if( q.count() < 2 ) return;
    QueueList q1 = new QueueList();
    QueueList q2 = new QueueList();
    split(q,q1,q2);
```

```
        sort(q1);
        sort(q2);
        merge(q1,q2,q);
    }

}
```

7.2.1 Notkun röðunarklasa

Eftirfarandi forritstexti inniheldur dæmi um notkun fjölnota klasans `Sorter`.

```
// Notkun:  mstest <INN >UT
// Fyrir:   INN inniheldur fleytitölur á textasniði.
// Eftir:   UT inniheldur sömu tölur í vaxandi röð.

import java.io.*;
import QueueList;
import QueueArray;
import MergeSort;

class mstest {
  public static void main( String[] argv ) throws IOException {

    StringQueue q = new QueueArray(1000);
    // Einnig mætti nota
    //    StringQueue q = new QueueList();
    // sem hefur þann kost að við getum raðað skrám með fleiri
    // en 1000 línum.

    String x;
    DataInputStream cin = new DataInputStream(System.in);
    x = cin.readLine();
    while( x != null ) {
        // Búið er að lesa núll eða fleiri línur úr INN
        // og þeir eru í q.  Síðan er nýbúið að reyna að
        // lesa eina línu enn, sem er í x ef lestur tókst.
        q.put(x);
        x = cin.readLine();
    }
    MergeSort.sort(q);
    while( q.count() != 0 ) {
        // UT inniheldur núll eða fleiri minnstu strengi úr
        // INN í vaxandi röð.  Hinir eru í q í vaxandi röð
        // (þ.a. fremsta gildið í q er minnst).
```

```
            System.out.println(q.get());
        }
    }
}
```

8 Forgangsbiðraðir og heapsort

Í kafla þessum er rætt um forgangsbiðraðir (*priority queue*) og tvær útfærslur þeirra, með hrúgu og með eintengdum lista. Það, sem er einkum mikilvægt hér er:

- Hugmyndin um forgangsbiðröð.
- Aðferðirnar, sem notaðar eru til að viðhalda hrúguskilyrði.
- Aðferðin, sem notuð er í útfærslunni með tengdum lista, til að bæta hlekk í raðaðan lista.

8.1 Skil fyrir forgangsbiðröð

Forgangsbiðröð (*priority queue*) er gagnamót, sem einkennist af eftirfarandi aðgerðum:

- Aðgerð til að setja gildi í forgangsbiðröðina.
- Aðgerð til að sækja og fjarlægja minnsta gildi úr forgangsbiðröðinni.

Sé eitthvert gagnamót búið slíkum aðgerðum, lítum við svo á að gagnamót þetta sé forgangsbiðröð. Að sjálfsögðu skiptir ekki höfuðmáli hér hvort aðgerðin, sem sækir gildi úr gagnamótinu sæki minnsta eða stærsta stak, heldur aðeins að stökin komi í vel skilgreindri röð.

Skil, `PriQueue`, fyrir forgangsbiðraðir Strengja gætu verið eftirfarandi:

```
interface PriQueue {

    // Notkun: x = p.deleteMin();
    // Fyrir:  p er ekki tóm.
    // Eftir:  x er strengur, sem var í p fyrir aðgerðina, og
    //         enginn annar minni strengur var í p.  x hefur
    //         verið fjarlægður úr p (þó geta verið x-gildi
    //         eftir í p, ef fleiri en eitt slíkt gildi var í p).
    public String deleteMin();

    // Notkun: p.put(x);
    // Fyrir:  p er ekki full.
    // Eftir:  x hefur verið bætt ofan í p.
    public void put( String x );

    // Notkun: n = p.count();
```

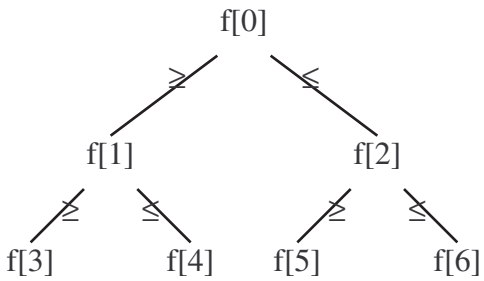
```
    // Eftir:  n inniheldur fjölda gilda í p.
    public int count();

    // Notkun: b = p.isFull();
    // Eftir:  b er satt þá og því aðeins að p sé full.
    public boolean isFull();
}
```

Athugið að ekki er hægt að búa til eintök af þessum klasa því skilgreint er í klasaskilgreiningu að ýmis lausbundin boð hafi enga útfærslu.

8.2 Forgangsbiðröð með hrúgu

Hrúguskipan (*heap*) gefur kost á hraðvirkum aðgerðum til að finna og fjarlægja stærsta (eða minnsta) stak í safni gilda. Þetta getum við notað okkur, og útfært forgangsbiðröð með hrúguskipan. Til þess að það gangi verðum við að sjálfsögðu einnig að hafa hraðvirka aðferð til að bæta staki við hrúgu án þess að hrúguskipanin glatist. Við munum sjá að slík aðferð er til, og er einföld í útfærslu.

Munum að hrúguskipanin í þessu tilfelli felst í því að foreldrið á ávallt að vera minna en börnin eða jafnt þeim, þ.e. `f[i]` $\leq$ `f[2*i+1]`, `f[2*i+2]`. Myndrænt lítur fremri hluti fylkisins þá svona út:



Eftirfarandi forritstexti skilgreinir klasa fyrir forgangsbiðröð með hrúguskipan:

```
import PriQueue;

class PriQueueHeap implements PriQueue {
  String[] f;
  int max,n;
  // f er fylki strengja með max sæti, max >= 0.
  // Strengirnir í forgangsbiðröðinni eru í f[0..n-1] í
  // hrúgu með minnsta efst.  n er því földi staka.

  // Notkun: PriQueueHeap p = new PriQueueHeap(max);
  // Fyrir:  max >= 0.
```

```
// Eftir:  p er ný tóm forgangsbiðröð með pláss fyrir
//         max strengi.
public PriQueueHeap( int max ) {
  this.max=max;
  f=new String[max];
  n=0;
}

public String deleteMin() {
  String result=f[0];
  f[0]=f[--n];
  int i=0;
  while( true ) {
    // result inniheldur gildið, sem skila skal.
    // Hin gildin eru í f[0..n-1] í hrúgu með minnsta
    // efst, nema hvað f[i] er e.t.v. of ofarlega í
    // trénu, ef 0<=i<n.
    // 0 <= i <= n, og verið getur að n sé 0.
    int b=i+i+1;       // vinstra barn i
    if( b>=n )
      return result;  // i er barnlaust
    if( b+1<n && f[b+1].compareTo(f[b])<0 )
      b++;             // hægra barn er minna
    // b vísar nú á minnsta barn i
    if( f[b].compareTo(f[i])>=0 )
      return result;  // f[i] er á réttum stað
    // víxlum nú á f[i] og f[b]:
    String temp=f[i];
    f[i]=f[b];
    f[b]=temp;
    i=b;
  }
}

public void put( String x ) {
  int i=n++;
  f[i]=x;
  while( i>0 && f[i].compareTo(f[(i-1)/2]) < 0 ) {
    // Gildin í forgangsbiðröðinni eru í f[0..n-1]
    // í hrúgu með minnsta efst, nema hvað f[i] er
    // e.t.v. of neðarlega.
    String temp=f[i];
    f[i]=f[(i-1)/2];
```

```
      f[(i-1)/2]=temp;
      i=(i-1)/2;
    }
  }

  public int count() {
    return n;
  }

  public boolean isFull() {
    return (n==1000);
  }

}
```

Í aðferðinni `put` sjáum við að það er einfalt að bæta einu gildi í hrúgu. Aðferðin fyrir deleteMin, sem eyðir minnsta staki, er heldur flóknari.

8.3 Forgangsbiðröð með tengdum lista

Forgangsbiðröð má einnig útfæra með tengdum lista. Eftirfarandi forritstexti sýnir slíka forgangsbiðröð.

```
import PriQueue;

class link {
  String val;
  link next;
}

class PriQueueChain implements PriQueue {

  link chain;
  // Fastayrðing gagna:
  //   Stök forgangsbiðraðarinnar eru í keðjunni
  //   chain, í vaxandi röð frá fremsta hlekk til
  //   aftasta.

  // Notkun: PriQueueChain p = new PriQueueChain();
  // Eftir:  p er ný tóm forgangsbiðröð með pláss fyrir
  //         ótakmarkaðan fjölda strengja, meðan
  //         minnisrými leyfir.
  public PriQueueChain() {
    chain=null;
```

```
}

public String deleteMin() {
  String result=chain.val;
  chain=chain.next;
  return result;
}

public void put( String x ) {
  link newlink=new link();
  newlink.val=x;
  if( chain == null || x.compareTo(chain.val)<=0 ) {
    newlink.next=chain;
    chain=newlink;
    return;
  }
  link fremri=chain;
  while(  fremri.next != null
       && x.compareTo(fremri.next.val) > 0
       )
  {
      // [g1|.]->...->[gN|.]->...->[gM|/]
      //  ^            ^
      //  chain        fremri
      //
      // g1,...,gN eru öll minni en x
      fremri=fremri.next;
  }
  newlink.next=fremri.next;
  fremri.next=newlink;
}

public int count() {
  int n=0;
  link leif=chain;
  while( leif != null ) {
      // leif er keðja allra ótalinna hlekkja.
      // n er fjöldi talinna hlekkja.
      n++;
      leif=leif.next;
  }
  return n;
}
```

```
  public boolean isFull() {
    return false;
  }
}
```

9 Yfirlit yfir ýmsar fastayrðingar

Þessi kafli inniheldur yfirlit yfir nokkrar fastayrðingar, sem koma oft fyrir í ýmsum forritum.

9.1 Hraðvirkar reikniaðgerðir

9.1.1 Veldishafning

Gefið: Tala x og heiltala $y \geq 0$.

Verkefni: Reikna $p = x^y$.

Upphafsstaða: $p = 1$, $q = x$, $r = y$, $y \geq 0$

Fastayrðing: $x^y = pq^r$, $r \geq 0$

Lokastaða: $r = 0$, og þar með er $p = x^y$

9.1.2 Margföldun

Gefið: Tala x og heiltala $y \geq 0$.

Verkefni: Reikna $p = xy$.

Upphafsstaða: $p = 0$, $q = x$, $r = y$, $y \geq 0$

Fastayrðing: $xy = p + qr$, $r \geq 0$

Lokastaða: $r = 0$, og þar með er $p = xy$

9.2 Helmingunarleit

9.2.1 Helmingunarleit að núllstöð

Gefið: $\epsilon > 0$ og fall (t.d. klasaboð) f, sem er samfellt á lokaða bilinu $[a, b]$ (þar sem $a \leq b$) og hefur þann eiginleika að $f(a)f(b) \leq 0$.

Verkefni: Finna x í $[a, b]$ þ.a. til sé z þ.a. $f(z) = 0$ og $|x - z| < \epsilon$.

Upphafsstaða: $p = a$, $q = b$.

Fastayrðing: $a \leq p \leq q \leq b$, $f(p)f(q) \leq 0$.

Lokastaða: $q - p < \epsilon$, og þar með eru bæði p og q innan ϵ fjárlægðar frá rót (núllstöð) f.

9.2.2 Helmingunarleit að staðbundum lággildispunkti

Gefið: $\epsilon > 0$ og fall f, sem er samfellt á lokaða bilinu $[a, c]$, punktur b þ.a. $a < b < c$, sem hefur þann eiginleika að $f(b) \leq f(a)$ og $f(b) \leq f(c)$.

Verkefni: Finna x í $[a, c]$ þ.a. til sé staðbundinn lággildispunktur z þ.a. $|x - z| < \epsilon$.

Upphafsstaða: $p = a$, $q = b$, $r = c$.

Fastayrðing: $a \leq p < q < r \leq c$, $f(q) \leq \min(f(p), f(r))$.

Lokastaða: $r - p \leq \epsilon$, og þar með er q innan ϵ fjárlægðar frá staðbundnum lággildispunkti f.

9.2.3 Helmingunarleit að gildi í fylki

Gefið: Gildi x og svæði f[a..b-1], sem er raðað í vaxandi röð.

Verkefni: Finna x í f[a..b-1] eða staðinn þar sem x ætti að vera, ef x væri í fylkinu.

Upphafsstaða: i = a, j = b.

Fastayrðing: f[a..i-1] < x ≤ f[j..b-1], a ≤ i ≤ j ≤ b.

Lokastaða: i = j og þar með gildir að ef x finnst í f[a..b-1] þá er f[i] = x. Auk þess er þá ljóst að ef bæta ætti x í fylkið þannig að fylkið haldist í vaxandi röð, þá ætti að setja það í sæti i, því þar er fyrsta gildi ≥ x (ef það sæti er til).

9.3 Röðun fylkja

9.3.1 Selection sort

Ytri lykkja

Gefið: Óraðað svæði f[a..b-1] í fylki.

Verkefni: Raða svæðinu í vaxandi röð.

Upphafsstaða: i = a.

Fastayrðing: f[a..i-1] inniheldur minnstu stök í vaxandi röð, a ≤ i ≤ b.

Lokastaða: i = b, og þar með er f[a..b-1] raðað í vaxandi röð.

Innri lykkja, 1. afbrigði

Upphafsstaða: f[a..i-1] inniheldur minnstu stök í vaxandi röð, $a \leq i < b$, $k = i + 1$.

Fastayrðing: f[a..i-1] inniheldur minnstu stök í vaxandi röð, $a \leq i < k \leq b$, f[i] er minnst af f[i..k-1].

Lokastaða: $k = b$, og þar með inniheldur f[a..i] minnstu stök í vaxandi röð.

Innri lykkja, 2. afbrigði

Upphafsstaða: f[a..i-1] inniheldur minnstu stök í vaxandi röð, $a \leq i < b$, $k = b$.

Fastayrðing: f[a..i-1] inniheldur minnstu stök í vaxandi röð, $a \leq i < k \leq b$, $f[i] \leq f[k..b-1]$.

Lokastaða: $k = i + 1$, og þar með inniheldur f[a..i] minnstu stök í vaxandi röð.

9.3.2 Insertion sort

Ytri lykkja

Gefið: Óraðað svæði f[a..b-1] í fylki.

Verkefni: Raða svæðinu í vaxandi röð.

Upphafsstaða: $i = a$.

Fastayrðing: f[a..i-1] er í vaxandi röð, $a \leq i \leq b$.

Lokastaða: $i = b$, og þar með er f[a..b-1] raðað í vaxandi röð.

Innri lykkja

Upphafsstaða: f[a..i-1] er í vaxandi röð, $k = i$.

Fastayrðing: f[a..i] er í vaxandi röð, nema hvað e.t.v. er f[k] of aftarlega.

Lokastaða: f[k] er ekki of aftarlega, og þar með er f[a..i] í vaxandi röð.

9.3.3 Quicksort skiptingaraðferð 1

Gefið: Óraðað svæði f[a..b-1] í fylki.

Verkefni: Umraða svæðinu þannig að því sé skipt í þrjú svæði, sem raða megi óháð hvert öðru, og þannig að það sé minna vandamál að raða þeim svæðum en upphaflega svæðinu.

Upphafsstaða: $f[a] = p$, $i = k = a+1$, þar sem p er eitthvert gildi, helst nálægt því að vera miðgildið.

Fastayrðing: $f[a] = p$, $f[a+1..k-1] < p$, $f[k..i-1] \geq p$, $a < k \leq i \leq b$.

Lokastaða: $i = b$ og þar með er $f[a] = p$, $f[a+1..k-1] < p$, $f[k..b-1] \geq p$.

Staða eftir leiðréttingu: Eftir að lykkju lýkur, minnkum við k og víxlum á f[a] og f[k]. Eftir það höfum við: $f[a..k-1] < p$, $f[k] = p$, $f[k+1..b-1] \leq p$.

9.3.4 Quicksort skiptingaraðferð 2

Gefið: Óraðað svæði f[a..b-1] í fylki.

Verkefni: Umraða svæðinu þannig að því sé skipt í þrjú svæði, sem raða megi óháð hvert öðru, og þannig að það sé minna vandamál að raða þeim svæðum en upphaflega svæðinu.

Upphafsstaða: f[a] $=$ p, k $=$ a, i $=$ a $+ 1$, n $=$ b, þar sem p er eitthvert gildi, helst nálægt því að vera miðgildið.

Fastayrðing: f[a..k-1] $<$ p, f[k..i-1] $=$ p, f[n..b-1] $>$ p, a $\leq$ k $\leq$ i $\leq$ n $\leq$ b.

Lokastaða: i $=$ n og þar með er f[a..k-1] $<$ p, f[k..i-1] $=$ p, f[i..b-1] $>$ p.

9.3.5 Heapsort

Hrúguskilyrði Svæði f[i..j] uppfyllir hrúguskilyrði ef fyrir öll sæti í svæðinu gildir að foreldri er $\geq$ afkvæmi (þar sem sæti k hefur afkvæmi 2k+1 og 2k+2).

Fyrri lykkja

Gefið: Óraðað svæði f[0..n] í fylki.

Verkefni: Umraða svæðinu þ.a. það verði hrúga.

Upphafsstaða: f[0..n] er óraðað, 2k $+ 1 >$ n, $0 \leq$ k $\leq$ n.

Fastayrðing: f[k..n] uppfyllir hrúguskilyrði, $0 \leq$ k $\leq$ n.

Lokastaða: k $= 0$ og þar með er f[0..n] hrúga (með stærsta gildi í f[0]).

Seinni lykkja

Gefið: Svæði f[0..n] í fylki, sem er hrúga.

Verkefni: Umraða svæðinu þ.a. það verði í vaxandi röð.

Upphafsstaða: f[0..n] er hrúga.

Fastayrðing: f[0..k] er hrúga, f[k+1..n] inniheldur stærstu gildin í vaxandi röð.

Lokastaða: k $= 0$ (eða k $= -1$) og þar með er f[0..n] raðað í vaxandi röð.

9.4 Lestur og skrift

9.4.1 Innlestur úr skrá í fylki

Gefið: Fylki f.

Verkefni: Lesa öll gildi úr skrá í fremsta hluta fylkisins.

Upphafsstaða: n=0.

Fastayrðing: Búið er að lesa n gildi úr skránni. Gildin eru geymd í f[0..n-1] í lesröð. Allar lestraraðgerðir skiluðu gagnagildi, nema e.t.v. sú síðasta.

Lokastaða: Síðasta lestraraðgerð gaf til kynna að skránni væri lokið, og þar með inniheldur f[0..n-1] allt innihald skrárinnar í lesröð.

9.4.2 Lestur frá notanda

Gefið: Spurning, sem notandinn þarf að svara á tilgreindan hátt áður en lengra er haldið.

Verkefni: Lesa svarið frá notandanum.

Upphafsstaða: Notandi hefur enn ekki verið spurður um svar við spurningu þeirri, sem forritið þarfnast svars við.

Fastayrðing: Reynt hefur verið núll sinnum eða oftar að spyrja notandann. Öll hans svör, nema e.t.v. það síðasta, ef eitthvert er, hafa verið óviðunandi.

Lokastaða: Nýjasta svar notandans var viðunandi. Þar með er ljóst að við erum komin með viðunandi svar, og að notandinn var ekki spurður oftar eftir að hann gaf viðunandi svar.

9.4.3 Útskrift úr fylki

Gefið: Svæði f[0..n-1] í fylki.

Verkefni: Skrifa gildin.

Upphafsstaða: Ekkert hefur verið skrifað úr fylkinu, i=0.

Fastayrðing: Búið er að skrifa gildin f[0..i-1], í þeirri röð.

Lokastaða: i=n og þar með er búið að skrifa n fyrstu gildin úr f, þ.e. f[0..n-1], í þeirri röð.

9.5 Fylkjareikningar

Í eftirfarandi aðferðum við fylkjareikninga fer hvert skref í lykkjunni þannig fram að gildi þau, sem verið er að vinna með eru margfölduð með einhverju andhverfanlegu fylki, hægra eða vinstra megin eftir atvikum.

Takið eftir að frumaðgerðir á fylki, svo sem að víxla súlum eða röðum, margfalda súlu eða röð með tölu ($\neq 0$) og leggja röð við röð, eða súlu við súlu, eru allt aðgerðir, sem eru jafngildar því að margfalda fylkið hægra eða vinstra megin með andhverfanlegu fylki.

9.5.1 Lausn jöfnuhneppis $\mathbf{A}\mathbf{x} = \mathbf{b}$

Gefið: Fylki $\mathbf{A}$ og vigur $\mathbf{b}$.

Verkefni: Finna $\mathbf{x}$ þannig að $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Upphafsstaða: $\mathbf{A}' = \mathbf{A}$, $\mathbf{b}' = \mathbf{b}$.

Fastayrðing: Til er andhverfanlegt fylki $\mathbf{E}$ þ.a. $\mathbf{A}' = \mathbf{E}\mathbf{A}$ og $\mathbf{b}' = \mathbf{E}\mathbf{b}$.

Lokastaða: $\mathbf{A}' = \mathbf{I}$ og þar með er $\mathbf{x} = \mathbf{b}'$.

9.5.2 Vinstri andhverfa fylkis

Gefið: Fylki $\mathbf{A}$.

Verkefni: Finna $\mathbf{B}$ þannig að $\mathbf{B}\mathbf{A} = \mathbf{I}$.

Upphafsstaða: $\mathbf{A}' = \mathbf{A}$, $\mathbf{B} = \mathbf{I}$.

Fastayrðing: $\mathbf{B}$ er andhverfanlegt fylki og $\mathbf{A}' = \mathbf{B}\mathbf{A}$.

Lokastaða: $\mathbf{A}' = \mathbf{I}$ og þar með er $\mathbf{B}$ vinstri andhverfa $\mathbf{A}$.

9.5.3 Hægri andhverfa fylkis

Gefið: Fylki $\mathbf{A}$.

Verkefni: Finna $\mathbf{B}$ þannig að $\mathbf{A}\mathbf{B} = \mathbf{I}$.

Upphafsstaða: $\mathbf{A}' = \mathbf{A}$, $\mathbf{B} = \mathbf{I}$.

Fastayrðing: $\mathbf{B}$ er andhverfanlegt fylki og $\mathbf{A}' = \mathbf{A}\mathbf{B}$.

Lokastaða: $\mathbf{A}' = \mathbf{I}$ og þar með er $\mathbf{B}$ hægri andhverfa $\mathbf{A}$.

9.6 Sérkennilegar stöðulýsingar

9.6.1 Ósönn fastayrðing

Fastayrðing lykkju verður að vera sönn fyrir og eftir sérhverja umferð lykkjunnar, þar með talið fyrir fyrstu og eftir síðustu, og jafnvel þótt engin umferð sé farin.

Þar með getur fastayrðing lykkju aðeins verið ósönn ef ekki er komið að lykkjunni í keyrslu.

9.6.2 Ósatt eftirskilyrði

Ef eftirskilyrði er ávallt ósatt þá lýkur lykkju aldrei á eðlilegan hátt. Þetta getur samt sem áður komið fyrir í skynsamlegu forriti, til dæmis ef return-setning kemur fyrir í lykkjunni.

9.7 Hvar setjum við fastayrðingar?

Fastayrðingar skal setja á alla þá staði í forritstexta, sem möguleiki er á framkvæmdin *stökkvi í* á einvern hátt, í keyrslu.

Athugið að samkvæmt þessari skilgreiningu þarf fastayrðingu í byrjun hverrar lykkju, sama hvort um er að ræða while-lykkju, þar sem skilyrði lykkjunnar er reiknað fyrir hverja umferð, eða do-lykkju, þar sem skilyrðið er reiknað eftir hverja umferð.

Athugið einnig að samkvæmt þessari reglu þarf fastayrðingu í byrjun hvers boðs, en slíka fastayrðingu köllum við þá forskilyrði boðsins. Þá þarf einnig fastayrðingu eftir sérhvert kall á boð, en það er þá eftirskilyrði boðsins.

10 Röksemdafærsla og arfgengi

Í þessum kafla er rætt um samband það, sem gilda verður milli lýsinga í yfirklösum og undirklösun, og, sem einnig þarf að gilda milli lýsinga í skilum (e. *interface*) og klösum, sem uppfylla skilin.

10.1 Lýsingar boða

Boð í klösum í Java hafa lýsingar, þ.e. forskilyrði og eftirskilyrði, og það verða að gilda vissar reglur um sambandið milli lýsinga á mismunandi útfærslum á sama boði í mismunandi klösum.

Reglur þessar helgast af okkar kröfu um að röksemdafærsla okkar sé ávallt traust, þ.e. að þegar við færum okkur áfram í tíma í framkvæmd forrits frá stöðulýsingu S_1 til stöðulýsingar S_2, án þess að stöðunni sé breytt þar á milli, gildi ávallt að stöðulýsing S_2 sé bein afleiðing af stöðulýsingu S_1.

Íhugum almennt dæmi um tvo klasa `A` og `B` þar sem `B` erfir frá `A`. Skilgreiningin á klasa `A` gæti verið eitthvað á þessa leið:

```
class A {
    ...
    // Notkun: x.f();
    // Fyrir:  F_A
    // Eftir:  E_A
```

```
    public void f();
    ...
}
```

þar sem F_A er þá sú stöðulýsing, sem er forskilyrði fyrir því að senda megi hlut `x` af klasa `A` boðið `f()`, og E_A er samsvarandi eftirskilyrði.

Klasinn `B` gæti haft eftirfarandi skilgreiningu:

```
class B extends A {
    ...
    // Notkun: x.f();
    // Fyrir:  F_B
    // Eftir:  E_B
    public void f();
    ...
};
```

Einfaldast er að forskilyrði og eftirskilyrði boðsins `f()` í klasanum `B` sé það sama og í klasa `A`, þ.e. að $F_A = F_B$ og $E_A = E_B$. Það mun sjá til þess að okkar röksemdafærsla sé traust, og þá viljum við reyndar helst sleppa því að endurtaka lýsingu boðsins í klasa `B`.

Íhugum nú boð `h()`, sem tekur viðfang af tilvísunartagi `A`. Slíkt boð gæti í stórum dráttum litið svona út:

```
void h( A z ) {
    ...
    // Staða α, stöðulýsing: F
    z.f();
    // Staða β, stöðulýsing: E
    ...
}
```

Á þetta boð má kalla með viðfangi af tagi `A` eða af tagi `B`, eða viðfangið má vera af öðru undirtagi `A`.

Nú er ljóst eð við verðum að gera þær kröfur til útfærslu á klasanum `B` að það sé traust forritun að senda tilvik af klasa `B` sem viðfang í boðið `h()`. Við skulum því gera eilitla greiningu á því hvaða skilyrði þarf þá að uppfylla.

Í stöðu α í boði `h()` vitum við að hluturinn `z` er af tagi `A` eða af undirtagi og við vitum að stöðulýsingin F er sönn. Þetta þarf að duga til þess að löglegt sé að senda `z` boðið `h()` og að það leiði til þess að stöðulýsingin E verði sönn eftir það.

Nú er ljóst að þegar við forritum aðferð boðsins `h()` höfum við almennt einungis lýsingu klasans `A` í fórum okkar, og jafnvel þótt við hefðum lýsingu einhverra undirklasa `A` þá gætum við ekki komið í veg fyrir að aðrir undirklasar `A` yrðu búnir til seinna meir. Við verðum því að setja einhverjar almennar

reglur, sem fylgja verður í öllum undirklösum `A`, og sem tryggja að aðferð boðs eins og `h()` vinni rétt.

Aðferð boðsins `h()` verðum við að geta forritað með því að taka einungis tillit til lýsingarinnar á klasa `A`. Það verður því að duga fyrir réttmæti boðsins `h()` að röksemdafærsla með tilliti til klasa `A` sé rétt. Röksemdafærsla með tilliti til klasa `A` gengur fyrir sig á eftirfarandi hátt:

```
void h( A z ) {
    ...
    // Staða α, stöðulýsing: F
    // Staða α′, stöðulýsing: F_A
    z.f();
    // Staða β′, stöðulýsing: E_A
    // Staða β, stöðulýsing: E
    ...
}
```

Hér þarf því að gilda að $F \to F_A$ og $E_A \to E$, því annars er okkur ekki leyfilegt að ganga frá stöðu α til stöðu α' og frá stöðu β' til stöðu β. Þetta er því sú röksemdafærsla, sem við göngum í gegnum þegar við forritum boðið `h()`.

En hvað gerist þá þegar breytan `z` er í raun af einhverjum undirklasa `B`? Þá ætti röksemdafærsla af eftirfarandi tagi að vera gild:

```
void h( A z ) {
    ...
    // Staða α, stöðulýsing: F
    // Staða α′, stöðulýsing: F_B
    z.f();
    // Staða β′, stöðulýsing: E_B
    // Staða β, stöðulýsing: E
    ...
}
```

Við verðum því að tryggja að fyrir alla undirklasa `B` gildi að $F \to F_B$ og $E_B \to E$. Til þess að það sé tryggt er ljóst að eftirfarandi verður að gilda:

$$F_A \to F_B \tag{1}$$

og

$$E_B \to E_A \tag{2}$$

eða, með öðrum orðum: Forskilyrðið F_B í undirklasanum má vera *víðara* en forskilyrðið F_A í grunnklasanum, og eftirskilyrðið E_B í undirklasanum má vera *þrengra* en eftirskilyrðið E_A í grunnklasanum. Þetta er uppfyllt þá og því aðeins að til séu fullyrðingar F' og E' þannig að

$$F_B = F_A \text{ eða } F' \tag{3}$$

og

$$E_B = E_A \text{ og } E' \tag{4}$$

Takið eftir að ef skilyrði 1 og 2 að ofan eru uppfyllt þá mun $F' = F_B$ og $E' = E_B$ uppfylla skilyrði 3 og 4.

10.2 Dæmi

Eftirfarandi klasar sýna hvernig ofangreindum reglum má beita. Grunnklasinn `IntStack` er eftirfarandi:

```
// Eintök af þessum klasa eru hlaðar heiltalna.
class IntStack {

   int f[10];
   int n;
   // Fastayrðing gagna:
   //    Stök hlaðans eru í f[0..n-1] frá neðsta til efsta

   // Notkun: IntStack s = new IntStack();
   // Eftir:  s er nýr tómur hlaði heiltalna
   //         með pláss fyrir 10 tölur.
   public IntStack() {
     n=0;
   }

   // Notkun: s.push(i);
   // Fyrir:  s er ekki fullur.
   // Eftir:  i hefur verið sett ofan á s.
   public void Push( int i ) {
     f[n++]=i;
   }

   // Notkun: i = s.pop();
   // Fyrir:  s er ekki tómur.
   // Eftir:  Efsta stakið hefur verið fjarlægt úr
   //         s og er í i.
   public int pop() {
     return f[--n];
   }

   // Notkun: c = s.count();
   // Eftir:  c er fjöldi staka í s.
   public int count() {
```

```
    return n;
  }
}
```

Klasinn `SafeIntStack` er eftirfarandi:

```
import IntStack;

class SafeIntStack extends IntStack {

  // Notkun: SafeIntStack s;
  // Eftir:  s er nýr tómur hlaði heiltalna,
  //         með pláss fyrir 10 tölur.
  SafeIntStack() {
    super();
  }

  // Notkun: s.push(i);
  // Fyrir:  Ekkert (þá má sleppa þessu forskilyrði)
  // Eftir:  Ef s var fullur hefur ekkert gerst, annars
  //         hefur i verið sett ofan á s.
  public void Push( int i ) {
    if(Count()<10) {
      super.push(i);
    }
  }

  // Notkun: i = s.pop();
  // Fyrir:  Ekkert (einnig má sleppa þessu)
  // Eftir:  Ef hlaðinn var tómur er i=0, annars
  //         hefur efsta stakið verið fjarlægt af s og
  //         er í i.
  public int pop() {
    return (Count()==0) ? 0 : stack::Pop();
  }
}
```

Hér er ljóst að vegna þess að við fylgdum þeim reglum, sem lýst er að ofan, munu allir forritskaflar, sem sannanlega vinna rétt fyrir breytur af tagi `IntStack` einnig vinna rétt fyrir breytur af tagi `SafeIntStack`.

11 Æfingar

Setjið viðeigandi fastayrðingar í eftirfarandi fullyrðingar svo ljóst sé að forritskaflarnir séu naumréttir.

Dæmi 1.

```
// m>=0
x=0;
for( n=1 ; n<=m ; n++ ) {
  // ?
  x=x+n;
}
// x = m*(m+1) / 2
```

Dæmi 2.

```
// s=X*Y-x*y
while(x%2==0) {
  // ?
  y=2*y;
  x=x/2
}
// s=X*Y-x*y og x%2=1
```

Dæmi 3.

```
// s == X*Y-x*y
while(x!=0) {
  // ?
  while( x%2 == 0 ) {
    // ?
    y=2*y;
    x=x/2;
  }
  s=s+y;
  x=x-1
}
// s=X*Y
```

Dæmi 4.

```
// x=X og y=Y
s=0;
while(x!=0) {
  // ?
  while( x%2 == 0 ) {
    // ?
    y=2*y;
    x=x/2;
  }
```

```
  s=s+y;
  x=x-1;
}
// s=X*Y
```

Sannið eftirfarandi fullyrðingar.

Dæmi 5.

```
// n>=0
prod=0;
for( x=1 ; x<=n ; x++ ) prod=prod+m;
// prod=m*n
```

Dæmi 6.

```
// x>0, y>0
s=0;
for( i=1 ; i<=x ; i++ )
  for( j=1 ; j<=y ; j++ )
    s = s+1;
// s=x*y
```

Dæmi 7.

```
// s=(X-x)*y, r inniheldur drasl
r=0;
while(r!=y) {
  s=s+1;
  r=r+1;
}
x=x-1;
// s=(X-x)*y
```

Dæmi 8.

```
// x=X og y=Y, r inniheldur drasl
s=0;
while( x!=0 ) {
  r=0;
  while( r!=y ) {
    s=s+1;
    r=r+1;
  }
  x=x-1;
}
// s=X*Y
```

Dæmi 9.

```
// x=X, n=N
p=1;
while( n!=0 ) {
  // p*x^n=X^N
  if( n%2 == 1 ) p=p*x;
  n=n/2;
  x=x*x
}
// p=X^N
```

Dæmi 10. Sannið að forritskaflinn

```
z=0;
while( x!=0 ) {
  if( x%2 ) z=z+y;
  y=y+y;
  x=x/2
}
```

reikni margfeldi upphaflegu gildanna í x og y og setji niðurstöðuna í z.
Dæmi 11. Hannið reglur, sem nota má til að sanna réttmæti do-setningar á sniðinu

```
// F
do {
  S
} until R;
// E
```

Hvar viljið þið setja fastayrðinguna? Hvert er sambandið milli fullyrðinganna í setningunni?
Dæmi 12. Sannið réttmæti eftirfarandi falls (klasaboðs):

```
// Notkun: x=fact(n)
// Fyrir:  n er ekki-neikvæð heiltala.
// Eftir:  x inniheldur n!
double fact( int n ) {
  if(n==0)
    return 1.0;
  else
    return n*fact(n-1);
}
```

Dæmi 13. Sannið réttmæti eftirfarandi falls (klasaboðs):

```
// Notkun: x=fact(n)
// Fyrir:  n er ekki-neikvæð heiltala
// Eftir:  x inniheldur n!
double fact( int n ) {
  double res = 1.0;
  while( n!=0 ) {
    res=res*n;
    n=n-1;
  }
  return res;
}
```

Dæmi 14. Sannið réttmæti eftirfarandi falla (klasaboða):

```
// Notkun: x=hjalp(n,y)
// Fyrir:  n er ekki-neikvæð heiltala
//         y er rauntala
// Eftir:  x=n!*y
double hjalp( int n, double y ) {
  if( n==0 )
    return y;
  else
    return hjalp(n-1,n*y)
}

// Notkun: x=fact(n)
// Fyrir:  n er ekki-neikvæð heiltala
// Eftir:  x inniheldur n!
double fact( int n ) {
  return hjalp(n,1.0);
}
```

12 Lausnir

Lausn á dæmi 1. $x = (n-1)n/2$ og $n \leq m+1$.
Lausn á dæmi 2. $s = XY - xy$.
Lausn á dæmi 3. Báðar fastayrðingarnar eru $s = XY - xy$.
Lausn á dæmi 4. Báðar fastayrðingarnar eru $s = XY - xy$.
Lausn á dæmi 5. Fullyrðingin er augljós þegar fastayrðingin $prod = m(x-1)$ er notuð.
Lausn á dæmi 6. Fullyrðingin er augljós þegar fastayrðingarnar $s = (i-1)y$ og $s = (i-1)y + j - 1$ eru notaðar.
Lausn á dæmi 7. Bætum fullyrðingum inn á eftirfarandi hátt:

```
// s=(X-x)*y, r inniheldur drasl
```

```
r=0;
while( r!=y ) {
  // s=(X-x)*y+r
  s=s+1;
  r=r+1
}
// s=(X-x)*y+y
x=x-1
// s=(X-(x+1))*y+y
// s=(X-x)*y-y+y
// s=(X-x)*y
```

Lausn á dæmi 8. Fullyrðingin er augljós þegar fastayrðingarnar

$$s = (X - x)Y,\ y = Y$$

og

$$s = (X - x)Y + r,\ y = Y$$

eru notaðar.

Lausn á dæmi 9. Hér eru tvær leiðir í gegn um almenna umferð lykkjunnar, og ljóst er að það nægir að sýna að báðar leiðir viðhalda fastayrðingunni. Annað tilfellið er þegar n er oddatala. Þá nægir að sanna eftirfarandi:

```
  // p*x^n=X^N, n er oddatala
  p=p*x;
  n=n/2;
  x=x*x
  // p*x^n=X^N
```

En þetta er augljóst ef við bætum skilyrðum milli setninganna:

```
  // p*x^n=X^N, n er oddatala
  p=p*x;
  // p*x^n=x*X^N, n er oddatala
  n=n/2;
  // p*x^(2n+1)=x*X^N
  // p*x^(2n)=X^N
  // p*(x^2)^n=X^N
  x=x*x
  // p*x^n=X^N
```

Hitt tilfellið er þegar n er jöfn tala. Þá nægir að sanna eftirfarandi:

```
  // p*x^n=X^N, n er j{\"o}fn tala
  n=n/2;
  x=x*x
  // p*x^n=X^N
```

En þetta er augljóst ef við bætum skilyrðum milli setninganna:

```
  // p*x^n=X^N, n er j{\"o}fn tala
  n=n/2;
  // p*x^(2n)=X^N
  // p*(x^2)^n=X^N
  x=x*x
  // p*x^n=X^N
```

Lausn á dæmi 10.

```
// X=x, Y=y
z=0;
while( x!=0 ) {
  // z+x*y=X*Y
  if( x%2==1 ) z=z+y;
  y=y+y;
  x=x/2
}
// X,Y innihalda upphafleg gildi
// x,y og z=X*Y
```

Einfalt er að sannfæra sig um að fastayrðingunni er viðhaldið í tilfellunum tveimur, sem eru möguleg, þ.e. x oddatala og x jöfn tala.

Lausn á dæmi 11. Setjum fastayrðingar I og Y í do-setninguna á eftirfarandi hátt:

```
// F
do {
  // I
  S
  // Y
}
until R;
// E
```

og heimtum að I sé ávallt sönn í byrjun hverrar umferðar og Y sé ávallt sönn í lok hverrar umferðar. Sambandið milli fullyrðinganna verður þá eftirfarandi: $F \rightarrow I$, $\{I\}S\{Y\}$, $Y\ og\ ekki\ R \rightarrow I$, $Y\ og\ R \rightarrow E$.

Lausn á dæmi 12. Augljóslega naumrétt því þegar n er núll er eftirskilyrði augljóslega fullnægt með því að skila einum. Ef n er ekki núll, þá hlýtur n að vera stærra en núll (samkvæmt forskilyrði), og endurkvæma kallið er því löglegt og skilar $(n-1)!$, og þarmeð skilar upphaflega kallið $n!$ og fullnægir eftirskilyrði.

Fallið er einnig rammrétt vegna þess að í endurkvæma kallinu er viðfangið einu minna en það var áður, en verður aldrei minna en 0.

Lausn á dæmi 13. Augljóst er að fallið er naumrétt eftir að við bætum við draugabreytum og fullyrðingum á eftirfarandi hátt í stofninn:

```
res=1.0;
// Látum N=n
// N er upphaflegt n, res=1, n>=0
while( n!=0 ) {
  // res=N!/n!, n>=0
  res=res*n;
  n=n-1;
}
// res=N!, N er upphaflegt n
return res;
```

Fallið er einnig rammrétt vegna þess að í hverri umferð lykkjunnar minnkar n, en verður aldrei minna en 0.

Lausn á dæmi 14. Augljóst. Fallið fact er augljóslega rammrétt miðað við lýsinguna á hjalp. Fallið hjalp er augljóslega rétt í tilfellinu $n = 0$. Í tilfellinu $n \neq 0$ er ljóst að n er stærra en núll, og $n - 1$ er því ekki-neikvæð heiltala. Þar með er endurkvæma kallið leyfilegt og skilar $(n-1)!ny$, sem er jafnt $n!y$, sem er einmitt það, sem skila á úr upphaflega kallinu.

Kalli á fallið hjalp lýkur avallt vegna þess að gildin á n í endurkvæmum vakningum þess eru síminnkandi, ekki-neikvæðar heiltölur.

13 Ritaskrá

Heimildir

[Dijkstra 76] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[Floyd 67] R. W. Floyd, Assigning Meaning to Programs. Proc. Symp. Appl. Math. Í **Mathematical Aspects of Computer Science**, ritstj. J. T. Schwartz. Providence, RI: American Mathematical Society, 1967.

[Gries 81] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[Hoare 69] C. A. R. Hoare, An Axiomatic Basis for Computer Programming. **Comm. ACM 12** 10 (október 1969): 576–580.

[Hoare 72] C. A. R. Hoare, Proof of Correctness of Data Representation. **Acta Informatica 1** (1972): 271–281.

[Parnas 72a] D. L. Parnas, A Technique for Module Specification with Examples. **Comm. ACM 15** 5 (maí 1972): 330–336.

[Parnas 72b] D. L. Parnas, On the Criteria to Be Used in Decomposing Systems into Modules. **Comm. ACM 15** 12 (desember 1972): 1053–1058.