7.5 Katekismus

- △ Afprøvninger kan bruges til at finde fejl i programmer.
- △ Et program kan delvist dokumenteres med et antal prøvekørsler, der ikke afslørede nogen fejl – forudsat at de anvendte prøvedata er repræsentative.
- △ Man kan benytte begreberne ekstern og intern afprøvning til at finde repræsentative prøvedata.

```
do
   b -> S
 | b' -> S'
 | b'' -> S''
od
```

Dette er en generaliseret iteration (~while), men med flere betingelser. Der vælges non-deterministisk mellem de sande betingelser (hvorefter den tilhørende statement eksekveres), indtil alle betingelser er false.

8 Udsagn, korrekthed og invarianter

Programmering ved trinvis forfinelse, som introduceret i kapitel 5, havde som erklæret formål at gøre (større) programmer lettere at læse. Teknikken har imidlertid også den nyttige virkning, at den gør programmer lettere at ***skrive***, fordi man nedbryder en opgave til små overskuelige enheder, hvis løsninger er forholdsvis ligefremme. Dette efterlader imidlertid (mindst) to problemer: det kan være vanskeligt at få selv små programmer til at virke korrekt, og det er ikke altid lige nemt at sammensætte småprogrammer, så det resulterende (større) program kommer til at virke efter hensigten.

I dette kapitel introduceres en teknik, der sigter på at afhjælpe begge disse problemer: programmering med ***udsagn***. Et udsagn er en påstand om (sammenhænge mellem) programmets variabler, som placeres på velvalgte steder i programteksten. Formålet er at finde sådanne egenskaber, som gør det nemt at argumentere for, at programmet fungerer efter hensigten.

Betragt som et simpelt eksempel orddelingsprogrammet fra kapitel 5. Programmet har (på et vist ekspansionsniveau) følgende udseende

```
(+ Var ord: Text
   «read word»
   do |word| ≠ 0 →
      (+ Var vowel, hyp: Vector
         Var result: Text
         «find positions of vowels»
         «find positions of hyphenations»
         { word kan deles præcis før hyp.(0), ..., hyp.(|hyp|-1) }
         «compute result»
         write("Hyphenation:   ", result, eol)
      +)
   od
   «read word»
+)
```

Selve programmet her er ikke vigtig for forståelsen

hvor vi har anbragt et udsagn, omsluttet af { }-parenteser, imellem stum-

perne ≪find positions of hyphenations≫ og ≪compute result≫. Udsagnet

{ word kan deles præcis før hyp.(0), ..., hyp.(|hyp|-1) }

er en påstand om sammenhængen mellem variablerne word og hyp, som siger, at vektoren hyp indeholder de korrekte delepositioner for ord. Formålet med dette udsagn er nu, at hvis ≪find positions of hyphenations≫ er skrevet på en sådan måde, at udsagnet er opfyldt, så er det temmeligt klart, at programmet er korrekt. Udsagn bruges til at beskrive en ønskelig egenskab ved tilstandstabellen, eller blot ***tilstanden***, på det pågældende sted i programmet.

Behovet for at bruge udsagn er også udtalt i forbindelse med **do**-sætninger. Betragt som eksempel veksleprogrammet fra afsnit 2.9. Argumentet for at dette program faktisk veksler korrekt er, "at der ikke bliver noget væk undervejs". Dette kan vi præcisere i form af følgende udsagn

$$\{ \text{amount} = k_{10}*10+k_5*5+k_1+\text{rest}) \wedge (\text{rest} \geq 0) \}$$

Udsagnet er en konjunktion, hvis to faktorer udtrykker

- det oprindelige beløb er lig med summen af det hidtil vekslede og resten.
- resten bliver aldrig til gæld.

Dette udsagn er opfyldt før og efter hvert gennemløb af løkken og kaldes derfor en ***invariant.*** Invarianter for **do**-sætninger skrives i programmer umiddelbart efter nøgleordet **do** på følgende måde


```
(+ Var amount:Int
   read[amount]
   if amount< 0 → abort fi
   (+ Var k1, k5, k10, rest: Int
      k1, k5, k10, rest := 0, 0, 0, amount
      do { (amount = k10*10+k5*5+k1+rest) ∧ (rest ≥ 0) }
         10 ≤ rest → k10, rest := k10+1, rest-10
       | 5 ≤ rest → k5, rest := k5+1, rest-5
       | 1 ≤ rest → k1, rest := k1+1, rest-1
      od
      write(k1, eol, k5, eol, k10, eol)
   +)
+)
```

8.1 Dekorerede programmer

Programmer, der er udstyrede med udsagn, vil blive kaldt ***dekorerede*** programmer. Vi skal i dette afsnit beskrive de grundlæggende begreber i forbindelse hermed.

Et udsagn er som sagt en påstand om programmets variabler. Denne påstand kan være rigtig eller forkert, afhængigt af variablernes værdier, som jo findes i tilstandstabellen. Et udsagn kan derfor også betragtes som en påstand om tilstandstabellen, en påstand som er rigtig for nogle tilstandstabeller og forkert for andre. Udsagnet

{ x+y = z }

er fx sandt i tilstandstabellen

N	V
x: v_0	v_0: 1
y: v_1	v_1: 2
z: v_3	v_3: 3

men falsk i tabellen

N	V
x: v_0	v_0: 1
y: v_1	v_1: 2
z: v_3	v_3: 0

Vi siger, at en tilstandstabel ***opfylder et udsagn*** U, hvis det udtryk, som fremkommer ved i U at erstatte alle variabler med deres værdier i tilstandstabellen, er sandt. Her og i det følgende antages det, at udsagn kun udtaler sig om variabler, der forefindes i tilstandstabellen på det pågældende sted i programmet.

Når et program udføres, kan man opfatte det, som om tilstandstabellen bevæger sig fra sætning til sætning i overensstemmelse med udfaldet af programmets forskellige betingelser og lader sig udsætte for disse sætningers manipulationer. Da udsagn er knyttet til bestemte lokaliteter i programmet, kan vi på simpel vis tale om, at et udsagn ***nås*** (af den vandrende tilstandstabel) under programudførelsen.

Vi siger nu, at et udsagn er ***gyldigt***, hvis det er opfyldt hver gang det nås under enhver udførelse af programmet. Et program siges at være gyldigt, hvis alle dets udsagn er gyldige.

Udsagn kan placeres alle steder i et program, hvor der må stå en sætning. Herudover kan det være "limet" til **do** i en **do**-sætning; i sidstnævnte tilfælde angiver placeringen et sted, der passeres både ***før*** og ***efter*** hvert gennemløb.

8.2 Gyldige programmer

I dette afsnit viser vi et antal eksempler på (nyttige) dekorationer, der gør programmer gyldige. Det første eksempel er veksleprogrammet ovenfor. Vi påstår, at udsagnet

$$I : \{ \text{ amount} = k_{10}*10+k_5*5+k_1+\text{rest}) \wedge (\text{rest} \geq 0) \ \}$$

er gyldigt, det vil sige, at det opfyldes af programmets tilstandstabel før og efter hvert gennemløb af løkken. Argumentet hvorfor er en slags induktionsargument, hvor man viser at

- udsagnet er opfyldt ***første*** gang det passeres.
- hvis udsagnet er opfyldt ***før*** en iteration, så er det også opfyldt ***efter*** iterationen.

I dette tilfælde er det nemt at se, at I faktisk er gyldig.

- det er klart at initialiseringen

  $$k_1,\ k_5,\ k_{10},\ \text{rest} := 0,\ 0,\ 0,\ \text{amount}$$

  resulterer i en tilstandstabel, der opfylder I.
- antag, at I opfyldes af tilstandstabellen på et givet tidspunkt og betragt en iteration, der starter i denne tilstand. Hvis ingen af de tre kontrollerende betingelser er opfyldt, så sker der ingenting, hvorfor I naturligvis stadig er opfyldt bagefter. Hvis sætningen hørende til en af løkkens alternativer udføres, så sker der det, at det relevante k_i øges med 1 ***samtidig*** med, at rest nedsættes med i. Dette betyder, at den første faktor i I også er opfyldt bagefter. At (rest$\geq$0) også er opfyldt følger af, at den kontrollerende betingelse i det pågældende alternativ var sand før udførelsen af den tilhørende sætning (ellers var sætningen jo ikke blevet udført).

Som næste eksempel betragtes følgende (lidt overdrevent) dekorerede program til udregning af polynomiet $x^n + x^{n-1} + \cdots + x + 1$.

```
(+ Var x, psum, n, m: Int
   read[x, n]
   if n<0 → abort fi
   { n≥0 }
   psum, m:=1, 0
   do { (psum = x^m + x^(m-1) + ··· + x + 1) ∧ (0 ≤ m ≤ n) }
      m ≠ n →
         psum:=psum*x+1
         m:=m+1
   od
   { psum = x^n + x^(n-1) + ··· + x + 1 }
   write(psum)
+)
```

Argumentet for at dette program er gyldigt består i at vise, at programmets tre udsagn alle er gyldige. Det går som følger

- at udsagnet $\{n \geq 0\}$ er gyldigt følger af, at hvis betingelsen ($n<0$) i **if**-sætningen er sand, så udføres **abort**-sætningen, det vil sige, at programmet ***standser***. Vi når derfor kun til udsagnet $\{n \geq 0\}$, hvis **if**-sætningens betingelse er falsk.
- at invarianten er gyldig indses igen ved at observere, at den opfyldes af initialiseringen, samt at opfyldelsen bevares af en udførelse af løkkens indmad. Vi kan argumentere herfor på samme sproglige måde som ovenfor, men da der i mange sammenhænge er brug for en mere regneorienteret måde til at eftervise invarianter, illustrerer vi nedenfor, hvordan det gøres i dette tilfælde.
- at det sidste udsagn er gyldigt, følger af gyldigheden af invarianten samt det faktum, at løkkens betingelse må være falsk, når vi kommer ud.

Man kan "regne" på invarianter på følgende måde. Vi antager, at invarianten på et givet tidspunkt opfyldes af tilstanden og skal vise, at den også opfyldes af den tilstand, der fremkommer ved at udføre sætningen

```
(*)   psum:=psum*x+1
      m:=m+1
```

Efter at have udført koden.

Lad os betegne ***værdierne*** af programmets variabler psum, x, m og n i de to tilstandstabeller med henholdsvis $psum, x, m, n$ og $psum', x', m', n'$. At invarianten er opfyldt før udførelsen af $(*)$ betyder, at følgende er sandt

$$(psum = x^m + x^{m-1} + \cdots + x + 1) \wedge (0 \leq m \leq n)$$

At den er opfyldt bagefter betyder, at følgende skal være sandt

$$(psum' = x'^{m'} + x'^{m'-1} + \cdots + x' + 1) \wedge (0 \leq m' \leq n')$$

Da $(*)$ kun udføres når **do**-sætningens kontrollerende betingelse er opfyldt, ved vi yderligere at $(n \neq m)$ og endelig gælder der for sammen-

hængen mellem mærkede og umærkede værdier at

$$(**) \quad \begin{array}{rcl} psum' & = & psum * x + 1 \\ x' & = & x \\ m' & = & m + 1 \\ n' & = & n \end{array}$$

jvf. do-indmaden

Alt i alt skal vi altså vise, at

$$(psum' = x'^{m'} + x'^{m'-1} + \cdots + x' + 1) \wedge (0 \leq m' \leq n')$$

under antagelse af

i) $(psum = x^m + x^{m-1} + \cdots + x + 1) \wedge (0 \leq m \leq n)$

ii) $(n \neq m)$

iii) $(**)$

Men det er et simpelt regnestykke, idet

$$\begin{array}{rcll} psum' & = & psum * x + 1 & \text{fra iii)} \\ & = & (x^m + x^{m-1} + \cdots + x + 1) * x + 1 & \text{fra i)} \\ & = & x^{m+1} + x^m + \cdots + x + 1 & \\ & = & x'^{m'} + x'^{m'-1} + \cdots + x' + 1 & \text{fra iii)} \end{array}$$

Ligeledes har vi

$$\begin{array}{rcll} (0 \leq m' \leq n') & = & (0 \leq m + 1 \leq n) & \text{fra iii)} \\ & = & (-1 \leq m < n) & \\ & \Leftarrow & (0 \leq m < n) & \\ & = & \text{true} & \text{fra i) og ii)} \end{array}$$

Den her viste detaljeringsgrad er for høj til at virke i praksis, men ideen med at vise, at den "umærkede" invariant og "umærkede kontrollerende betingelse" medfører den "mærkede invariant" er nyttig og vil også blive brugt fremover. Mere præcist skal vi anvende en skabelon for invariantbeviser, som i ovenstående tilfælde har følgende udseende.

Basis

$(1 = x^0) \wedge (0 \leq 0 \leq n)$ er sand da $n \geq 0$.

Invarians

$$\begin{aligned} psum' &= psum * x + 1 \\ &= (x^{m} + x^{m-1} + \cdots + x + 1) * x + 1 \\ &= x'^{m'} + x'^{m'-1} + \cdots + x' + 1 \end{aligned}$$

$m' = m + 1$, det vil sige, $0 \le m' \le n'$, da $0 \le m < n$.

Som det ses, er fremgangsmåden den samme som i den mere detaljerede udgave, bortset fra, at vi tillader os at bruge selve variabelnavnene (i mærket og umærket udgave) til at angive deres værdier, samt at den kontrollerende betingelse og sammenhængen mellem mærkede og umærkede variabler kun er til stede indirekte.

Det tredje og sidste eksempel i dette afsnit er programmet om cifferblokke fra afsnit 3.3.4. Vi påstår, at følgende dekorerede version er gyldig

```
(+ Var lin: Text
   write("Write a line:  ")
   read[lin]
   lin:=lin++Text(eol)
   (+ Var i, d: Int
      i, d:=0, 0
      do { I }
         i+d < | lin | →
            if '0' ≤ lin.(i+d) ≤ '9' → d:=d+1
            & d=0 → i:=i+1
            | d>0 →
                 write(lin(i..i+d), eol)
                 i, d:=i+d+1, 0
            fi
      od
   +)
+)
```

KURSORISK

hvor I er udsagnet

I = (lin(i..i+d) er kun cifre) ∧ (i+d ≤ |lin|) ∧
((i=0) ∨ (lin.(i-1) er ikke et ciffer))

Denne gang er invarianten mere indviklet, og det er gyldighedsargumentet også. Vi viser det i detaljer.

Basis

Når (i=0) og (d=0) så er lin(i..i+d) = lin(0..0) som er tom. Den tomme tegnfølge opfylder trivielt udsagnet, at den kun indeholder cifre. Invariantens anden og tredje faktor er automatisk opfyldt.

Invarians

Der er tre tilfælde (det er i alle tilfælde klart, at i'+d' ≤ |lin|).

- lin.(i+d) er et ciffer: lin(i'..i'+d') = lin(i..i+d+1) består kun af cifre, fordi lin(i..i+d) gør, og lin.(i+d) er et ciffer.
- lin.(i+d) er ikke et ciffer og (d=0): lin(i'..i'+d') = lin(i'..i') som er tom, og lin.(i'-1) = lin.(i+1-1) = lin.(i+d), som ikke er et ciffer.
- lin.(i+d) er ikke et ciffer og (d>0): lin(i'..i'+d')=lin(i+d+1..i+d+1) som er tom, og lin.(i'-1)=lin.(i+d), som ikke er et ciffer.

Som læseren måske bemærker, er dette argument ikke præget af nogen større snedighed, men derimod af systematik og omhu med detaljer. Dette gælder i almindelighed for denne type ræsonnementer om programmers egenskaber.

8.3 Gyldighed og korrekthed

At et dekoreret program er gyldigt er ikke nødvendigvis særligt interessant. Forudsætningen er, at de valgte udsagn er ***nyttige***, det vil sige, at de siger noget, der hjælper programmøren med at overbevise sig om, at programmet fungerer korrekt. Følgende dekorerede program er således gyldigt, men gyldigheden kan ikke rigtigt bruges til noget

```
(+ Var n, p, i, res: Int
   read[n, p]
   if p<0 → abort fi
   { 5=2+3 }
   i, res:=0, 1
```

```
    do { ∀x : x² ≥ 0 }
        i<p → res, i:=res*n, i+1
    od
    { i≥0 }
    write(res)
+)
```

Udsagnene skal altså vælges på en sådan måde, at de lægger op til simple konklusioner vedrørende et programs egenskaber. Spørgsmålet om hvordan man mere formelt karakteriserer programmers egenskaber, herunder den formelle betydning af begrebet ***programkorrekthed***, ligger uden for rammerne af denne bog, og vi skal derfor nøjes med at appellere til den intuitive forståelse af, hvad det betyder, at et program "gør hvad det skal". I de tre eksempler i forrige afsnit kunne denne intuition være

- programmet veksler et vilkårligt ikke-negativt tal korrekt til enere, femmere og tiere.
- programmet beregner værdien af polynomiet $x^n + x^{n-1} + \cdots + x + 1$ for ethvert ikke negativt n og ethvert x.
- programmet indlæser en tekst og udskriver samtlige dens cifferblokke.

Som vi skal se, har vi i alle tre tilfælde har valgt nyttige udsagn. Først skal vi imidlertid påpege en simpel, men særdeles nyttig, kendsgerning om gyldighed af invarianter. Hvis et dekoreret program, hvori der optræder en **do**-sætning

```
do { I }
    b1 → S1
  | b2 → S2
  :
  | bn → Sn
od
```

~ while {I} (b) do S

er gyldigt, så er programmet stadig gyldigt, hvis det udvides med følgende dekoration


```
do { I }
    b1 → S1
  | b2 → S2
  :
  | bn → Sn
od
{ I ∧ ¬b1 ∧ ¬b2 ∧ ... ∧ ¬bn }
```

~ while {I} (b) do S {I ∧ ¬b}

...Forudsat b ikke indeholder side-effekter (e.g. assignments)

Observationen er den simple, at når udførelsen af **do**-sætningen er afsluttet, så skyldes det, at alle de kontrollerende betingelser er falske, hvorfor såvel invarianten (I) og negationen af betingelserne ($\neg b_i$) er opfyldte.

Nytten af udsagnene i de tre programmer kan nu indses som følger

- Princippet ovenfor giver direkte, at følgende program også er gyldigt

```
(+ Var amount: Int
   read[amount]
   if amount<0 → abort fi
   (+ Var k1, k5, k10, rest: Int
      k1, k5, k10, rest:=0, 0, 0, amount
      do { (amount = k10*10+k5*5+k1+rest) ∧ (rest ≥ 0) }
           10 ≤ rest → k10, rest:=k10+1, rest-10
         | 5 ≤ rest → k5, rest:=k5+1, rest-5
         | 1 ≤ rest → k1, rest:=k1+1, rest-1
      od
      { amount = k10*10+k5*5+k1 }
      write(k1, eol, k5, eol, k10, eol)
   +)
+)
```

men her står direkte, at k_{10}, k_5 og k_1 har de rigtige værdier, når de skrives ud.

- I det andet eksempel medfører gyldigheden af det sidste udsagn direkte, at det er det rigtige resultat, der skrives ud.
- Eksemplet med cifferblokkene er mere interessant, derved at det vel ikke er umiddelbart klart, at programmet gør hvad det skal. Den

gyldige invariant giver netop mulighed for at argumentere overbevisende herfor:

Det er naturligvis klart, at de tegnfølger, der udskrives, kun indeholder cifre; hvad der er mindre klart er, at de udskrevne cifferfølger er ***blokke***, det vil sige, at de er maksimale og ikke-tomme. Dette indses på følgende måde. Der udskrives kun noget i en situation hvor d er større end 0, invarianten er opfyldt og lin.(i+d) ikke er et ciffer. Da d er større end 0, følger det umiddelbart at lin(i..i+d) ikke er tom. Ifølge invarianten gælder der enten at (i=0) (det vil sige, at følgen står først på linjen) eller at lin.(i-1) ikke er et ciffer; i begge tilfælde er lin(i..i+d) maksimal og således en cifferblok.

Som læseren måske vil have opdaget, er gyldigheden af de tre programmer ikke tilstrækkelig til at overbevise om, at programmerne faktisk fungerer som de skal. Problemet er, at gyldighed af et udsagn kun er interessant som ***hvis*** udsagnet faktisk nås under programudførelsen. Gyldighed er en såkaldt ***sikkerhedsegenskab*** ved et program, det vil sige, en egenskab som garanterer, at ***hvis*** "der sker noget", så er det det rigtige, der sker – men gyldighed sikrer ikke, at der faktisk sker noget.

Følgende eksempel viser en version af polynomieprogrammet, der er gyldig, men som er uinteressant, fordi det på grund af en uendelig løkke aldrig udskriver noget resultat.

```
(+ Var x, psum, n, m: Int
   read[x, n]
   if n<0 → abort fi
   { n>0 }
   psum, m := 1, 0
   do { (psum = x^m + x^(m-1) + ··· + x + 1) ∧ (0 ≤ m ≤ n) }
      m ≥ 0 →
         psum := psum*x+1
         m := m+1
   od
   { psum = x^n + x^(n-1) + ··· + x + 1 }
   write(psum)
+)
```


De gyldige dekorerede programmer i dette afsnit har altså i realiteten kun hjulpet os med at vise, at

- ***hvis*** veksleprogrammet udskriver tre tal, ***så*** er de rigtige.
- ***hvis*** polynomieprogrammet udskriver et tal, ***så*** er det korrekt.
- ***hvis*** cifferblokprogrammet udskriver et antal tekster, ***så*** er de cifferblokkene.

De egenskaber ved et program, der gør det muligt at fjerne ***hvis***'erne og sikre, at der faktisk sker noget, er de såkaldte ***fremdriftsegenskaber***, som diskuteres i næste afsnit.

8.4 Terminering og korrekthed

Vi betragter endnu en gang vore tre programmer. Ved en simpel inspektion af programteksterne ser vi, at det eneste der kunne forhindre de to første i at nå til den afsluttende udskrift er, at de går i en uendelig løkke, det vil sige, at **do**-sætningerne bliver udført uendeligt længe. Med hensyn til cifferblokprogrammet ville manglende terminering ikke nødvendigvis betyde, at der slet ikke blev skrevet noget ud, men det ville stadig være en uhensigtsmæssig opførsel.

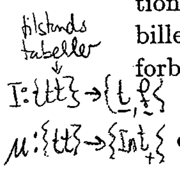
Vi argumenterer for, at samtlige **do**-sætninger terminerer

- I veksleprogrammet mindskes værdien af variablen rest i hvert gennemløb, og da den ifølge invarianten (som jo er gyldig!) ikke kan blive negativ, så må antallet af iterationer være endeligt.
- I polynomieprogrammet mindskes forskellen mellem n og m i hvert gennemløb, og da m ifølge invarianten altid er mindre end eller lig med n, så må antallet af iterationer være endeligt.
- I cifferblokprogrammet aftager størrelsen |lin|-(i+d) i hvert gennemløb, og da invarianten forhindrer den i at blive negativ, så må antallet af iterationer være endeligt.

KURSORISK

Ensartetheden af disse tre argumenter er påfaldende. Vi finder i alle tilfælde en størrelse (rest, n-m, |lin|-(i+d)) som aftager i hvert gennemløb, og som er nedadtil begrænset af invarianten.

En sådan størrelse kaldes en ***termineringsfunktion***, og den er i en vis forstand analog til et udsagn. Et udsagn er en (rigtig eller forkert) påstand om et programs tilstandstabel. En anden måde at sige dette på er, at et udsagn er en ***funktion*** fra tilstandstabeller ind i mængden af sandhedsværdier {true,false}. En termineringsfunktion er også en funktion fra tilstandstabeller, men i stedet for mængden {true,false} er dens billedmængde de hele tal. En termineringsfunktion er derudover altid forbundet med en gyldig invariant I, og har følgende to egenskaber



- værdien af termineringsfunktionen på en tilstandstabel, der opfylder I, er ikke-negativ.
- termineringsfunktionens værdi aftager med mindst 1 efter udførelsen af en iteration.

Der findes andre aspekter af fremdrift end sådanne termineringsfunktioner, men vi skal ikke studere dem nærmere i denne sammenhæng. I stedet slutter vi med en opsummering af, hvordan de i dette kapitel introducerede begreber tilsammen udgør en brugbar teknik til at argumentere for at programmer "gør som de skal". Som eksempel vælger vi følgende program, der realiserer den udvidede Euklids Algoritme, der beregner største fælles divisor og mindste fælles multiplum af to tal.

```
(+ Var n, m: Int
   write("Write n and m:  ")
   read[n, m]
   if (n<1) ∨ (m<1) → abort("Only positive numbers!") fi
   (+ Var p, q, s, t: Int
      p, q, s, t := n, m, m, n
      do { I }
         p>q → p, t := p-q, s+t
       | q>p → q, s := q-p, t+s
      od
      { U }
```



```
      write("Greatest common divisor:  ", p, eol)
      write("Least common multiple:  ", (s+t)/2, eol)
   +)
+)
```

Det er vel ikke umiddelbart indlysende, at dette program er korrekt, så her skal der findes en nyttig invariant og termineringsfunktion. Følgende invariant er brugbar (sfd betegner største fælles divisor, og mfm betegner mindste fælles multiplum)

$$I\text{: } (p*s+q*t = 2*sfd(n, m)*mfm(n, m)) \wedge (sfd(p, q) = sfd(n, m)) \wedge (p, q \geq 1)$$

og det er den tilhørende termineringsfunktion også

$$T\text{: } p+q$$

Vi viser først, at invarianten er gyldig.

Basis
Det følger af definitionen på største fælles divisor og mindste fælles multiplum, at der for vilkårlig positive heltal n og m gælder

$$n*m = sfd(n, m)*mfm(n, m)$$

Derfor opfylder initialiseringen klart de to første faktorer i I; den sidste faktor er opfyldt fordi programmet aborterer, hvis ikke $n, m \geq 1$.

Invarians
At $p', q' \geq 1$ følger af, at vi altid subtraherer det mindste af p og q fra det største.

At $p'*s'+q'*t' = p*s+q*t$ (m og n ændres jo ikke) følger i tilfælde q>p (det andet er analogt) af, at $p'*s'+q'*t' = p*(s+t)+(q-p)*t$, der reducerer til $p*s+q*t$.

At vise, at sfd(p',q')=sfd(p,q) er det samme som at vise (vi betragter igen tilfælde nummer to, det første er analogt) sfd(p,q-p)=sfd(p,q). Antag, at d går op i både p og q. Så er det klart, at d også går op i q-p, således at

$$\text{sfd}(p,q) \le \text{sfd}(p,q-p)$$

Antag nu, at d går op i både p og q−p; så går d også op i q, således at

$$\text{sfd}(p,q) \ge \text{sfd}(p,q-p)$$

Heraf følger gyldigheden af invarianten.

Da I er gyldig, er programmet også gyldigt med U som følgende udsagn

$$U : (p=q=\text{sfd}(n,m)) \wedge (\text{mfm}(n, m) = (s+t)/2)$$

fordi U følger direkte af $I \wedge (p \ge q) \wedge (q \ge p)$.

Så mangler vi kun at vise, at T er en termineringsfunktion. Med det følger af

- i enhver tilstand, der opfylder I, er $p+q \ge 2$.
- efter et gennemløb af løkken er $p'+q' = \max(p,q)$, som er mindre end p+q, fordi $p,q \ge 1$.

Vi konkluderer, at programmet er korrekt.

8.5 Programmering ved hjælp af udsagn

Hvis læseren på et tidspunkt har spurgt sig selv, hvordan man bærer sig ad med at trække en invariant som Euklids ud af ærmet, så er svaret, at ***det gør man ikke***. Sandheden er, at invarianten var der først, og at programmet blev skrevet på en sådan måde, at invarianten blev gyldig og termineringsfunktionen fik de ønskede egenskaber. Denne tilsyneladende "venden tingene på hovedet" er faktisk det vigtigste aspekt af begreberne udsagn, gyldighed, og så videre, fordi de dermed kommer til at optræde som elementer i en nyttig programmeringsteknik. Vi slutter kapitlet med at vise nogle eksempler på denne teknik.

Som det første eksempel betragter vi potensopløftningsprogrammet fra afsnit 2.7. Vi kan formulere opgaven som bestående i at skrive stumpen

≪Beregn res≫

på en sådan måde, at følgende program bliver gyldigt og terminerer

```
(+ Var n, p, res: Int
   read[n, p]
   if p<0 → abort fi
   ≪Beregn res≫
   { res = n^p }
   write(res)
+)
```

Et naturligt næste skridt er følgende

```
(+ Var n, p, res: Int
   read[n, p]
   if p < 0 → abort fi
   (+ Var i: Int
      ≪Initialiser res og i≫
      do {(res*n^i = n^p) ∧ (0 ≤ i ≤ p) }
         i ≠ 0 → ≪Opdater res og i≫
      od
   +)
   { res = n^p }
   write(res)
+)
```

Men nu er det nemt at se, at vi kan klare sagen med

where ≪Initialiser res og i≫ **is** res, i := 1, p

where ≪Opdater res og i≫ **is** res, i := res*n, i−1

og hvor i er termineringsfunktion.

Bemærk, at vi her startede med at anføre det ønskede slutudsagn. På basis heraf konstruerede vi invariant og kontrollerende betingelse i en **do**-sætning, så invarianten og negationen af den kontrollerende betingelse

tilsammen implicerede slutudsagnet. Herefter realiserede vi de to programstumper og kontrollerede, at der var en brugbar termineringsfunktion. Dette er en generel teknik, som brugt med afslappet omhu kan være særdeles nyttig.

Lad os (blandt andet for at illustrerer invarianter, der udtaler sig om lister) konstruere et program, der finder indeks for det største element i en vektor. Hvis V er en vektor og k er et heltal, skal vi tillade os at skrive k≥V, hvormed menes, at k er større end eller lig med alle værdierne i V.

Opgaven går ud på at gøre følgende program gyldigt og "terminerende"

```
(+ Var V: Vector
   Var x: Int
   read[V]
   if |V|=0 → abort fi
   ≪Find max indeks≫
   { V.(x) ≥ V }
   write(x)
+)
```

Det "største indeks" kan findes som "grænseværdien" for det "største indeks" i passende valgte delvektorer på følgende måde

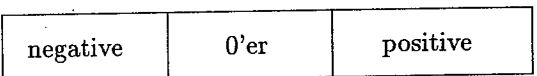
```
where ≪Find max indeks≫ is
      (+ Var i: Int
         ≪Initialiser i og x≫
         do { (V.(x) ≥ V(0..i)) ∧ (0 ≤ i ≤ |V|) }
            i ≠ |V| → ≪Opdater i og x≫
         od
      +)
```

Men så er det let at se, at følgende virker

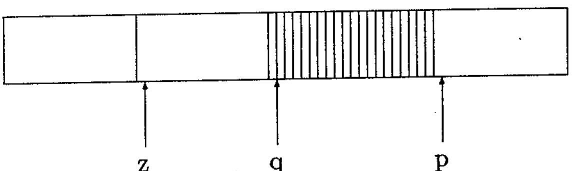
```
where ≪Initialiser i og x≫ is x, i:=0, 1

where ≪Opdater i og x≫ is
      if V.(x)<V.(i) → x:=i fi
      i:=i+1
```


hvor vi som termineringsfunktion kan bruge |V|-i.

Vi slutter med at udvikle følgende måske lidt mere interessante program, der indlæser en vektor og arrangerer dens elementer så alle de negative står til venstre, nul'erne står i midten, og de positive står til højre



Denne slutsituation er igen "grænseværdien" for en følge af situationer, nemlig



hvor elementerne til venstre for z er negative, elementerne mellem z og q er 0, og elementerne til højre for p er positive. Elementerne mellem q og p kender vi ikke noget til. Følgende program

```
(+ Var V: Vector
   Var z, q, p: Int
   read[V]
   ≪Initialiser z,q og p≫
   do { I }
      q≠p → ≪Omroker≫
   od
   write(V)
+)
```

har netop dette billede som invariant

$$I: (V(0..z)<0) \wedge (V(z..q)=0) \wedge (V(p..|V|)>0) \wedge (0 \leq z \leq q \leq p \leq |V|)$$

Det er klart, at hvis dette program er gyldigt og terminerer, så er det også korrekt. Stumperne kan skrives som

```
where ≪Initialiser z,q og p≫ is z, q, p:=0, 0, |V|

where ≪Omroker≫ is
        if V.(q)<0 →
              V.(z):=:V.(q)
              z,q:=z+1,q+1
         | V.(q)=0 → q:=q+1
         | V.(q)>0 →
              V.(q):=:V.(p-1)
              p:=p-1
        fi
```

med termineringsfunktion p-q.

Det er ikke noget tilfælde, at vi i disse eksempler har konstrueret invarianterne som "approximationer" til slutudsagnene. Det er faktisk en standardmetode at lade invarianten udtrykke, at man står med løsningen til en del af problemet, og så, ved hjælp af den kontrollerende betingelse, sørge for, at iterationen fortsætter, indtil denne del er lig med det hele. Det enkelte skridt i iterationen kommer så til at dreje sig om at udvide den løste del af problemet med et enkelt element.

8.6 Katekismus

- △ Et udsagn er enten sand eller falsk i en given tilstand.
- △ Et program kan dekoreres med udsagn.
- △ Et udsagn er gyldigt i et program, hvis det er sandt i den aktuelle tilstand hver gang det nås under programmets udførelse.
- △ Et program er gyldigt, hvis alle dets udsagn er gyldige.
- △ Ved hjælp af gyldige udsagn kan man bevise egenskaber ved et programs opførsel.

- △ I forbindelse med iterationer benyttes invarianter, hvis gyldighed vises ved induktion.
- △ Et programs terminering kan vises ved hjælp af en termineringsfunktion.