Indhold

1 Indledning **1**

2 Tilstande og tilstandsændringer **3**
2.1 Variabler og værdier . . . 3
2.2 Tilstandstabeller . . . 4
2.3 Udtryk . . . 5
2.3.1 Variabeludtryk . . . 5
2.3.2 Værdiudtryk . . . 5
2.3.3 Check for standardværdier . . . 6
2.4 Sætninger . . . 7
2.5 Implicit value . . . 11
2.6 Selektion . . . 12
2.7 Iteration . . . 13
2.8 Indskudte sætninger . . . 15
2.9 Nondeterminisme . . . 19
2.10 Generelle valg . . . 21
2.11 Pseudoreelle tal . . . 21
2.12 Standardværdier . . . 23
2.13 Oversigt . . . 24
2.13.1 Heltal . . . 24
2.13.2 Sandhedsværdier . . . 25
2.13.3 Pseudoreelle tal . . . 26
2.14 Beregningsregler . . . 27
2.15 Katekismus . . . 28

3 Regulære typer **30**
3.1 Navngivne typer . . . 30
3.2 Strukturerede mængder . . . 31
3.2.1 Eksempler . . . 32
3.3 Lister . . . 32
3.3.1 Eksempler . . . 33
3.3.2 Tilstandstabeller . . . 36
3.3.3 Oversigt . . . 38
3.3.4 Typen Text . . . 39
3.4 Produkter . . . 40
3.4.1 Eksempel . . . 41

3.4.2 Tilstandstabeller 42
3.4.3 Oversigt 43
3.5 Summer 44
3.5.1 Eksempel 44
3.5.2 Tilstandstabeller 45
3.5.3 Oversigt 47
3.6 Kassografi 47
3.7 Modellering med typer 48
3.8 Skabeloner for kontrol 50
3.8.1 Listeskabelon 50
3.8.2 Produktskabelon 50
3.8.3 Sumskabelon 51
3.8.4 Akkumulatorvariabler 51
3.8.5 Eksempel: eksamensresultater 52
3.9 Anonyme konstanter 53
3.9.1 Anonyme standardværdier 54
3.9.2 Anonyme lister 54
3.9.3 Anonyme produkter 54
3.9.4 Anonyme summer 55
3.9.5 Anvendelser 55
3.10 Kanoniske tekstformater 55
3.11 Konverteringer 57
3.12 Værdier i filer 58
3.13 Katekismus 58

4 Typeækvivalens **59**
4.1 En kongruensrelation 60
4.2 Normalformer 61
4.3 Katekismus 63

5 Trinvis forfinelse **64**
5.1 Ubestemte stumper 64
5.2 Eksempel: sikker indlæsning 65
5.3 Eksempel: orddeling 68
5.4 Katekismus 71

6 Eksempel: beregninger på relationer **72**
6.1 Relationer som typer 72

6.2 Gamle og nye beregninger . . . 72
6.3 Katekismus . . . 76

7 Systematisk afprøvning **77**
7.1 Afprøvningens psykologi . . . 77
7.2 Ekstern afprøvning . . . 78
7.3 Intern afprøvning . . . 79
7.4 Afprøvningens begrænsninger . . . 81
7.5 Katekismus . . . 82

8 Programmering med udsagn **83**
8.1 Udsagn . . . 83
8.2 Gyldige udsagn og programmer . . . 85
8.3 Bevis af gyldighed . . . 87
8.3.1 Bevisteknikken . . . 87
8.3.2 Sekvenser . . . 89
8.3.3 Selektioner . . . 90
8.3.4 Iterationer . . . 91
8.4 Bevis af terminering . . . 97
8.4.1 Termineringsfunktioner . . . 98
8.4.2 Tre eksempler . . . 98
8.5 Korrekthed . . . 99
8.6 Eksempel: Euklids algoritme . . . 100
8.7 Programmering med udsagn . . . 102
8.8 Brug og begrnsninger . . . 106
8.9 Katekismus . . . 106

9 Procedurer **107**
9.1 Eksempel: Petrinet tegninger . . . 107
9.2 Procedurer og tilstandstabeller . . . 113
9.3 Return sætningen . . . 118
9.4 Værdiprocedurer . . . 119
9.5 Variabelprocedurer . . . 120
9.6 Eksempel: opdatering af database . . . 123
9.7 Aspekter af variabelparametre . . . 124
9.7.1 Effektivitet . . . 124
9.7.2 Sideeffekter . . . 126
9.7.3 Deling . . . 127

9.8 Katekismus 128

10 Beskyttelse, datatyper og polymorfi **129**
10.1 Beskyttelse 129
10.2 Datatyper 132
10.3 Eksempel: papkasser 136
10.4 Polymorfi 140
10.5 Eksempel: polymorfe sekvenser 141
10.6 Eksempel: polymorfe kataloger 147
10.7 Kanoniske tekstformater 150
10.8 Katekismus 150

11 Definitioner og navne **152**
11.1 Statiske omgivelser 152
11.2 Eksempel 152
11.3 Formel definition 154
11.4 Navneregler 156
11.5 Rekursion 156
11.6 Katekismus 157

12 Rekursive procedurer **158**
12.1 Binære træer 158
12.2 Rekursive procedurer og tilstandstabeller 161
12.3 Eksempel: fraktaler 165
12.4 Eksempel: syntaksanalyse 167
12.5 Eksempel: rekursive planter 171
12.6 Katekismus 173

13 Rekursive typer **175**
13.1 Træer som rekursive typer 175
13.2 To eksempler på træer som rekursive typer 179
13.3 Rekursive værdier og variabler 184
13.4 Rekursiv typeækvivalens 187
13.5 Eksempel: rekursive labyrinter 188
13.6 Eksempel: UNIX filsystem 195
13.7 Katekismus 198

14 Pointere **199**
14.1 Pointer typer 199
14.2 Pointere og tilstandstabeller 200
14.3 Rekursive typer med pointere 203
14.4 Kanoniske tekstformater 207
14.5 Eksempel: cyklisk stak 207
14.6 Eksempel: hurtige sekvenser 210
14.7 Pointere til eksisterende variabler 212
14.7.1 Eksempel: trer med rodhftede blade 213
14.8 Katekismus 213

15 Systemer, processer og kanaler **215**
15.1 Kanaler og kommunikation 215
15.2 Programmering med flere processer 217
15.2.1 Eksempel: avisspalter 218
15.3 Nondeterministisk kommunikation 222
15.3.1 Eksempel: begrænset buffer 222
15.4 Ressourcedeling 226
15.4.1 Eksempel: tegneprocesser 226
15.5 Katekismus 231

16 Tid og plads **232**
16.1 Et fysisk lager 232
16.2 Repræsentation af tilstandstabeller 233
16.3 Tidsforbrug 235
16.4 Katekismus 238

17 Adgang til UNIX **240**
17.1 Systemkald 240
17.2 Menu-boxen 241
17.3 Prompt-boxen 242
17.4 Choice-boxen 243
17.5 Katekismus 245

A Grafik **246**

B Grammatik **250**

C Typografi **256**

D Programeksempler **257**

1 Indledning

Denne bog handler om programmering og programmeringssprog. Den beskriver et programmeringssprog TRINE, der er udviklet specielt til introducerende datalogiundervisning. En væsentlig drivkraft bag designet har derfor været at inkludere så mange af de mest fundamentale programmeringssprogsbegreber i så simpel og "ren" en form som muligt. Vi har også ønsket at konstruere et sprog, der giver et realistisk billede af kompleksiteten af algoritmiske beregninger, og som derfor ikke indeholder "medfødt" ineffektivitet. Disse to hensyn har ført til, at TRINE er blevet et traditionelt sprog i den forstand, at det er et såkaldt *imperativt* programmeringssprog, hvor programmer konstrueres som abstrakte maskiner bestående af et abstrakt lager (programmets variabler) og en slags abstrakte processer (programmets sætninger).

Med hensyn til såkaldte kontrolstrukturer er ingredienserne i TRINE de sædvanlige: tilordning, selektion, iteration, procedurer og processer (de sidste kommunikerer via synkroniserende kanaler). På det såkaldte dataabstraktionsområde er der udviklet et typesystem, som udover de sædvanlige atomare typer indeholder liste-, sum- og produkt-konstruktører, og som tillader fuldt rekursive typedefinitioner. Abstrakte datatyper realiseres ved hjælp af et simpelt polymorft modulbegreb, der understøtter beskyttelse på normal vis. Det er også muligt at benytte sædvanlige pointere.

Programmeringsmetodisk er TRINE og TRINE-systemet indrettet til at understøtte programstrukturering ved hjælp af trinvis forfinelse på en meget håndfast måde. Systemet tillader således, at (oversatte) programmer indeholder "huller" som kan udfyldes hen ad vejen, og programmøren kan efter ønske ekspandere programmer automatisk.

Fremstillingen i bogen lægger op til at betragte programmering som en målrettet aktivitet, hvis slutprodukt besidder formaliserbare egenskaber. Der lægges således vægt på ræsonnementer ved hjælp af udsagn, invarianter og lignende. Bogens eksempelmateriale er dog ikke særligt matematisk orienteret, og de større eksempler i de senere kapitler er hovedsagligt orienteret mod grafik og tegn-og-tekstbehandling.

Fremgangsmåden i bogen er at beskrive TRINE "nedefra og op", det vil sige, først introduceres de atomare typer og de fundamentale kontrolstrukturer.

Så følger de regulære typekonstruktører, og på dette grundlag introduceres programmeringsmetodologien i form af trinvis forfinelse, udsagn, invarianter, og så videre. Dernæst følger procedurer og boxe (Trine's moduler), og program- og datatype-biblioteker introduceres. Efter gennemgang af Trine's (traditionelt statiske) virkefeltsregler følger præsentationen af rekursive procedurer og rekursive typer. Disse ledsages af nogle substantielle eksempler. Brugen af pointere beskrives og sammenlignes med rekursive typer. Derefter introduceres programmer med flere processer (med nondeterministisk kommunikation) som et værktøj til løsning af opgaver, hvor en traditionel en-proces løsning er vanskelig eller umulig. Der gås dog ikke dybere ind på emnet multiprogrammering. Til sidst lægges grundlaget for en kvantitativ analyse af programmer, idet programmers tids- og pladsforbrug beskrives og motiveres udfra den fysiske maskine.

2 Tilstande og tilstandsændringer

I dette kapitel skal vi beskrive de grundlæggende byggeklodser i TRINE-processer.

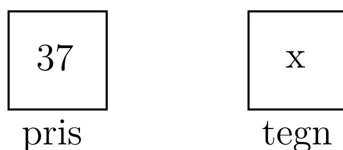
2.1 Variabler og værdier

Udførelsen af en enkelt proces resulterer i en række ændringer af processens *tilstand*. Når processen udføres af en datamat, repræsenterer denne tilstand et abstrakt syn på den del af maskinens lager, som processen har fået stillet til rådighed. Vi kan betragte dette som opdelt i et antal navngivne områder, som vi kalder *variabler*. Desuden kan vi betragte *indholdet* af disse variabler som mere end blot vilkårlige bitfølger; vi kan tildele variablerne *typer*, der giver os mulighed for at fortolke deres indhold som abstrakte *værdier*.

Til at begynde med betragter vi kun TRINE's *atomare* typer: Int, Bool og Char. Til enhver type T knytter vi en mængde af værdier, som betegnes med $\mathrm{Val}(T)$. For enhver type T indeholder $\mathrm{Val}(T)$ en speciel værdi ?, der betegnes *standardværdien* af type T. Indholdet af en variabel af type T er således en værdi i $\mathrm{Val}(T)$. For de nævnte simple typer har vi

- $\mathrm{Val}(\mathrm{Int}) = \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\} \cup \{?\}$
- $\mathrm{Val}(\mathrm{Bool}) = \{\mathrm{true}, \mathrm{false}\} \cup \{?\}$
- $\mathrm{Val}(\mathrm{Char}) = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\} \cup \{?\}$

Hvis en proces indeholder de to variabelnavne "pris" og "tegn" af type henholdsvis Int og Char, så har vi følgende billede



hvor de to "kasser" repræsenterer variablerne. Deres værdier er henholdsvis *tallet* 37 og *tegnet* x. I det følgende skal vi udvikle en notation, der tillader

os at beskrive en simpel proces. Til det formål skal vi beskrive processens tilstand og hvordan denne ændres.

2.2 Tilstandstabeller

Processens tilstand udgøres som nævnt af en samling navngivne variabler, hvis *indhold* (blandt andet) er værdier. Vi har brug for en simpel notation, der kan beskrive en procestilstand. Da det i almindelighed er for vanskeligt at tegne "kasser" som ovenfor, benytter vi *tilstandstabeller* som denne

N	V
$n_0 : v_0$	$v_0 : e_0$
$n_1 : v_1$	$v_1 : e_1$
$n_2 : v_2$	$v_2 : e_2$
$\cdots$	$\cdots$
$n_k : v_k$	$v_k : e_k$

Tabellen viser en tilstand hvor processen har variablerne $v_0, v_1, \ldots, v_k$ hvor n_i er et navn på variablen v_i og indholdet af v_i er værdien e_i. I eksemplet ovenfor har tilstandstabellen således følgende udseende

N	V
pris : v_0	v_0 : 37
tegn : v_1	v_1 : x

Betegnelserne v_0 og v_1 for de to variabler er helt vilkårlige; enhver form for navngivning af "kasserne" er acceptabel. Vi kan således repræsentere den samme tilstand med fx tabellen

N	V
pris : α	α : 37
tegn : β	β : x

Når en variabel netop er blevet skabt indeholder den altid standardværdien. Det ser i tilstandstabellen ud som fx

N	V
$n : v$	v: ?

2.3 Udtryk

Når vi skal beskrive en proces, bliver det nødvendigt at kunne *inspicere* tilstanden, og det gøres ved hjælp af *udtryk*. Et udtryk er et stykke tekst, der angiver en variabel eller en værdi relativt til den aktuelle tilstand. Der findes således to slags udtryk i TRINE: *variabeludtryk* og *værdiudtryk*.

2.3.1 Variabeludtryk

Et variabeludtryk angiver en variabel. På nuværende tidspunkt har vi kun meget simple variabeludtryk, nemlig *variabelnavne*, og de angiver den variabel, som navnet er knyttet til i N-søjlen af den aktuelle tilstand. I eksemplet ovenfor er "pris" et variabeludtryk, der angiver variablen v_0. Senere skal vi indføre mere komplicerede og sammensatte variabeludtryk.

2.3.2 Værdiudtryk

Ethvert variabeludtryk x angiver en variabel, der indeholder en værdi. Denne værdi kan angives med værdiudtrykket

value[x]

I eksemplet fra før har vi således, at value[pris] angiver værdien 37.

Der findes en række værdiudtryk, som ikke stammer fra variabeludtryk. Blandt disse er *konstanter*, som er udtryk, der angiver værdier *uafhængigt* af tilstanden. Alle værdierne i de atomare typer kan udtrykkes som konstanter:

- Man skriver ?-Int, ?-Bool og ?-Char for de forskellige standardværdier.
- Int-værdier skriver man i den sædvanlige decimale notation.
- De to Bool-værdier skrives som true og false.
- De forskellige tegn af type Char skrives omgivet af apostroffer, som fx `'a'`, `'b'`, `'c'` og så videre.

Det er forførende let at forveksle værdier og konstanter, men som man kan se ved Char, så er der forskel på de to: *konstanten* `'a'` er et udtryk, det vil sige et stykke tekst, der angiver *værdien* a – det første bogstav i alfabetet. Denne forskel ville komme endnu tydeligere frem, hvis vi af en eller anden grund havde valgt at angive TRINEs Int-værdier som romertal. Konstanterne ville da have set ud som IV, XIII, MCMXCVII og så videre, men de ville stadig angive værdierne 4, 13, 1997, etc. Værdimængden Val(Int) ville således være uforandret. Konstanter er udtryk, der har en konkret *syntaks*, men deres *semantik* er abstrakte værdier som tal, tegn, og så videre.

Det er ikke kun alle de atomare værdier, der kan udtrykkes ved hjælp af konstanter. Dette princip vil også blive overholdt for alle andre TRINE-værdier.

Blandt værdiudtrykkene finder vi også de *sammensatte* udtryk såsom

    2+2

der ikke overraskende angiver værdien 4. Sammensatte udtryk kan også involvere variabelnavne, som fx

    2*value[x]+value[y]

Når dette udtryk udregnes i tilstanden

N	V
x: v_1	v_1: 40
y: v_2	v_2: 7

bliver resultatet 87. Sammensatte udtryk kan opbygges af de sædvanlige aritmetiske og logiske operatorer, foruden mange andre beskrevet i afsnit 2.13.

2.3.3 Check for standardværdier

Hvis u er et værdiudtryk, så er

is(u)

et værdiudtryk af type Bool, der angiver false, hvis u angiver standardværdien, og true ellers.

2.4 Sætninger

Udover inspektion skal en tilstand også kunne *påvirkes*. Dette sker ved hjælp af en *sætning*. Udførelsen af en sætning kan altså ændre tilstanden. De fleste sætninger involverer i deres beskrivelse værdi- og variabeludtryk.

• Den simpleste sætning vi har er

skip

hvis udførelse ikke har nogen effekt; tilstanden ændres ikke. Af samme slags er også

wait

der standser udførelsen indtil brugeren trykker på `Return`. Sætningen

stop

afbryder udførelsen normalt, hvorimod sætningen

abort

afbryder udførelsen med en fejlrapportering.

• En *tilordning* er en sætning af formen

$x := u$

Her er x et variabeludtryk og u er et værdiudtryk af samme type. Når denne sætning udføres, bliver værdien af udtrykket u først udregnet i den aktuelle tilstand. Derefter ændres tilstanden således, at denne værdi bliver det nye indhold af variablen angivet ved x. Fx vil sætningen

```
x:=3*value[x]+value[y]-value[z]
```

ændre tilstanden

N	V
x: v_1	v_1: 7
y: v_2	v_2: 2
z: v_3	v_3: -4

til tilstanden

N	V
x: v_1	v_1: 27
y: v_2	v_2: 2
z: v_3	v_3: -4

- En *ombytning* er en sætning af formen

$$x\texttt{:=:}y$$

Her er x og y begge variabeludtryk af samme type. Når sætningen udføres, ombyttes indholdet af de tilsvarende variabler. Ombytningssætningen vil ændre tilstanden

N	V
x: v_1	v_1: 105
y: v_2	v_2: 87

til

N	V
x: v_1	v_1: 87
y: v_2	v_2: 105

- En *sekvens* er en sætning af formen

$$S_1 S_2 \ldots S_n$$

hvor hvert S_i er en sætning. Når sekvensen udføres, vil de indgående sætningerne alle blive udført i rækkefølge.

- En *simultan tilordning* er en sætning af formen

$$x_1, x_2, \ldots, x_k := u_1, u_2, \ldots, u_k$$

hvor alle x_i'erne er variabeludtryk, u_i'erne er værdiudtryk og hvert par x_i og u_i har samme type. Først udregnes alle udtrykkene i den aktuelle tilstand og derefter tilordnes værdierne til de respektive variabler i en ikke nærmere angivet rækkefølge. Denne sætning er *forskellig* fra sekvensen

$$\begin{aligned} &x_1 := u_1 \\ &x_2 := u_2 \\ &\cdots \\ &x_k := u_k \end{aligned}$$

Hvis vi fx starter i tilstanden

N	V
a: v_1	v_1: 7
b: v_2	v_2: 5

så vil sætningen

```
a,b := value[b]+1,value[a]+1
```

bringe os til tilstanden

N	V
a: v_1	v_1: 6
b: v_2	v_2: 8

hvorimod sekvensen

```
a := value[b]+1
b := value[a]+1
```

bringer os til tilstanden

N	V
a: v_1	v_1: 6
b: v_2	v_2: 7

Det er ofte mere bekvemt at bruge en simultan tilordning.

• Hvis en proces ikke kunne kommunikere med omverdenen, ville vi aldrig kunne drage nytte af dens anstrengelser. Processen kan derfor foretage *eksterne kommunikationer*, som sætningen

write(u)

hvor u er et værdiudtryk. Sætningen bevirker, at der udskrives en tekstuel repræsentation af værdien af udtrykket u i processens vindue. Omvendt vil sætningen

read[x]

hvor x er et variabeludtryk, bevirke, at der indlæses en tekst fra tastaturet. Denne fortolkes som en værdi (jfr. afsnit 3.10), der tilordnes den variabel, der angives af x. Der er to specielle udtryk, der ogs kan bruges til indlsning af enkelte tegn. Udtrykket

readchar

giver som resultat det nste tegn, der trykkes p tastaturet. Udtrykket

trychar

giver som resultat det tegn, der netop er trykket p tastaturet, og ?-Char, hvis et sdant ikke findes. Disse to udtryk kan bruges til interaktive programmer.

• Vi mangler endnu at se, hvordan variabler kan blive skabt. Når følgende sætning, en såkaldt *indskudt sætning*, udføres

```
(+ Var n1 : T1
   Var n2 : T2
   ...
   Var nk : Tk
   S
+)
```

oprettes der k variabler med navne $n_1, n_2, \ldots, n_k$ og typer $T_1, T_2, \ldots, T_k$. Derefter udføres sætningen S; inde i S kan man benytte de nyskabte variabler. Når sætningen er afsluttet, bliver variablerne nedlagt igen.

Med variabeldefinitioner til rådighed kan vi skrive fuldstændige processer – omend de på nuværende tidspunkt må blive meget enkle. Følgende sætning indlæser et tal og udskriver summen af dets nulte, første og anden potens

```
(+ Var x, psum: Int
   read[x]
   psum := 1
   psum := value[psum]*value[x]+1
   psum := value[psum]*value[x]+1
   write(value[psum])
+)
```

Det er let at overbevise sig om, at klassen af processer, som vi på nuværende tidspunkt kan beskrive, er stærkt begrænset. Vi kan fx ikke skrive en sætning, der indlæser to tal x og n og udskriver potenssummen

$$x^n + x^{n-1} + \cdots + x + 1$$

Hvad vi mangler er mere komplicerede sætningsstrukturer, der tillader os at *udvælge* og *gentage* sætninger.

2.5 Implicit value

Inden vi introducerer disse nye sætninger skal vi imidlertid slippe af med value[ ] som eksplicit signal til at aflæse *værdien* af et variabeludtryk. Ethvert variabeludtryk kan også opfattes som en værdiudtryk, og det fremgår altid af sammenhængen, hvornår det skal opfattes som hvad. TRINE-systemet er derfor indrettet så det accepterer variabeludtryk på steder,

hvor der egentligt skulle stå et værdiudtryk, idet det automatisk indsætter value[ ] de relevante steder.

Man kan således skrive

```
x:=y
x:=x+y
```

som automatisk vil blive konverteret til

```
x:=value[y]
x:=value[x]+value[y]
```

Vi skal i det følgende altid bruge denne kortere skrivemåde.

2.6 Selektion

En simpel *selektion* er en sætning af formen

if $b \ \rightarrow S$ **fi**

hvor b er et udtryk af type Bool og S er en sætning. Når selektionen udføres beregnes værdien af b; hvis resultatet er true, udføres sætningen S, ellers sker der intet videre. Hvis x er en heltalsvariabel, vil sætningen

if x $<$ 0 $\ \rightarrow$ x:=-x **fi**

ændre indholdet af x til dets absolutværdi. En selektion kan udvides til

if $b_1 \ \rightarrow S_1$
& $b_2 \ \rightarrow S_2$
& $\cdots$
& $b_n \ \rightarrow S_n$
fi

Denne sætning vil beregne alle b_i'erne efter tur og udføre det første S_i hvis betingelse er sand, det vil sige, hvis b_i har værdien true.

Med introduktionen af selektioner har vi allerede udvidet mængden af mulige beregninger. Vi kan fx beregne hvor mange dage, der er i februar i et givent år. Her husker vi, at et år er skudår, hvis det er deleligt med 4, medmindre det er deleligt med 100, medmindre det er deleligt med 400. Sætningen bliver

```
(+ Var year, days: Int
   read [year]
   if year mod 400 = 0 → days := 29
   & year mod 100 = 0 → days := 28
   & year mod 4 = 0 → days := 29
   & true → days := 28
   fi
   write (days)
+)
```

Rækkefølgen af de forskellige betingelser er afgørende for korrektheden.

2.7 Iteration

En simpel *iteration* er en sætning af formen

do $b \ \rightarrow S$ **od**

hvor b er et udtryk af type Bool og S er en sætning. Hvis b er falsk, det vil sige, hvis b har værdien false i den aktuelle tilstand, gør sætningen ingenting. Hvis b er sand, udføres S, og derefter udføres iterationen igen. Således udfører **do**-sætningen S igen og igen, sålænge b er sand.

Nu kan vi skrive den før omtalte sætning, der udregner summen af de $n+1$ første potenser af x

```
(+ Var x, psum, n: Int
   read [x, n]
   if n<0 → abort fi
   psum := 1
   do n>0 →
        psum := psum*x+1
        n := n-1
   od
   write(psum)
+)
```

Følgende sætning udskriver alle ASCII tegnene

```
(+ Var i: Int
   i := 0
   do i<256 →
        write(ic(i))
        i := i+1
   od
+)
```

Med indførelsen af iterationer er der sket en dramatisk ændring i, hvordan en proces kan opføre sig. Betragt sætningen

```
do true → skip od
```

Konstanten true er en betingelse, der altid er sand, så en proces der udfører denne sætning vil aldrig stoppe! Dette vil normalt være en uønsket situation, som vi fremover er nødt til at tage med i vores overvejelser. En uendelig løkke (ulykke!) behøver dog ikke at være helt så uproduktiv som ovenstående eksempel viser. Følgende sætning indlæser et tal n og udskriver alle decimalerne i reciprokværdien $1/n$; den er således både uendelig og produktiv.

```
(+ Var n: Int
    read[n]
    if n<2 → abort fi
    write('0', '.')
    (+ Var j: Int
        j := 10
        do true →
              write(j/n)
              j := 10*(j mod n)
        od
    +)
+)
```

Her er et eksempel, der kombinerer **if**- og **do**-sætninger. Sætningen udskriver primfaktoropløsningen af et indlæst tal

```
(+ Var n, p: Int
    read[n]
    if n<2 → abort fi
    p := 2
    do n ≠ 1 →
          if n mod p = 0 →
                write(p, eol)
                n := n/p
          & true → p := p+1
          fi
    od
+)
```

Korrektheden hviler på, at den mindste faktor i et tal nødvendigvis er et primtal.

2.8 Indskudte sætninger

Betragt følgende sætning, der indlæser to tal n og p og udskriver potensen n^p

```
(+ Var n, p, i, res: Int
    read [n, p]
    if p<0 → abort fi
    i, res := 0, 1
    do i < p → res, i := res*n, i+1 od
    write(res)
+)
```

De fire variabler n, p, i og res er alle definerede på samme niveau, men deres roller er faktisk forskellige, idet i kun optræder som hjælpevariabel for at få **do**-sætningen til at "løbe rundt". Programmet logiske struktur er egentligt

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

hvor det er klart, at variablen i er lokal til sætningen

```
≪Beregn res = n^p≫
```

Dette tilhørsforhold kan beskrives meget direkte, da vi jo kan skrive en indskudt sætning i en sekvens på lige fod med enhver anden sætning

```
(+ Var n, p, res: Int
    read [n, p]
    if p<0 → abort fi
    (+ Var i: Int
        res, i := 1, 0
        do i < p → res, i := res*n, i+1 od
    +)
    write(res)
+)
```

Sådanne *indskudte sætninger* rejser spørgsmålet om hvilke variabler, der er til rådighed hvornår. Betragt som eksempel følgende lidt simplere sætning

```
          (+ Var i, j: Int
               i := 0
(*1*)          (+ Var k: Int
                    k := i+1
                    i := k+1
(*2*)          +)
               j := i+1
(*3*)     +)
```

Som tidligere nævnt opstår variabler ved indgangen til en indskudt sætning, og de forsvinder ved udgangen. Det betyder, at ved (*1*) findes kun variablerne i og j, ved (*2*) findes i, j og k, og ved (*3*) er k forsvundet igen. Vi kan beskrive sådanne situationer ved at udvide vore tilstandstabeller på følgende måde

N	N	V
$n_1 : v_1$	$n_3 : v_3$	$v_1 : e_1$
$n_2 : v_2$		$v_2 : e_2$
		$v_3 : e_3$

Nu har vi flere N-søjler i tabellen. Ideen er, at de lokale variabelnavne i en indskudt sætning angives i en ekstra N-søjle til højre for de øvrige. Når den indskudte sætning er færdig, fjernes dens N-søjle, og de variabler den introducerede fjernes fra V-søjlen. Tilstandene ved de tre markerede steder kan nu angives på følgende måde

(*1*)

N	V
i: v_1	v_1: 0
j: v_2	v_2: ?

(*2*)

N	N	V
i: v_1	k: v_3	v_1: 2
j: v_2		v_2: ?
		v_3: 1

(*3*)

N	V
i: v_1	v_1: 2
j: v_2	v_2: 3

Bemærk, hvordan variablen v_3 bliver skabt og fjernet. Da der ikke er nogen grænse for hvor mange sætninger, der kan indskydes i hinanden, kan en tilstandstabel have vilkårligt mange N-søjler, som fx

N	N	N	N	V

Meningen med en indskudt sætning er, at den skal fungere som en *mellemregning*; vi vil tænke på dens udførelse som et enkelt skridt i en større beregning. De *lokale* variabler, den indeholder, er kun relevante for den selv og ville blot forvirre, hvis de var tilgængelige i hele beskrivelsen.

Vi har stadig en komplikation at overveje. Der kan være *navnesammenfald*, som i eksemplet

```
        (+ Var x, j: Int
           j := 0
           (+ Var x: Char
              x := 'A'
(*1*)         j := 7
           +)
        +)
```

Tilstanden ved (*1*) er

N	N	V
x: v_1	x: v_3	v_1: ?
j: v_2		v_2: 0
		v_3: A

Inde i den indskudte sætning er x en Char-variabel, udenfor er den en Int-variabel. En mulighed var at sige, at Int-variablen blev fjernet, når

Char-variablen blev skabt. Men det er ikke en rimelig situation. En mellemregning skal bestemt ikke fjerne en del af vores tilstand. Løsningen er at lade alle variablerne blive og indføre reglen, at man finder ud af hvilken variabel et navn angiver, ved at lede fra højre mod venstre i N-søjlerne. Når vi i eksemplet står ved `(*1*)` er variablen v_1 *parkeret* udenfor den indskudte sætning. Vi kan først få fat på den igen, når den indskudte sætning er færdig; dens navn er *overskygget* af det lokale navn. Vi ville stadig have et problem, hvis en enkelt N-søjle skulle indeholde flere forekomster af det samme navn, men den situation optræder ikke, fordi det er forbudt at erklære flere variabler med samme navne i den samme indskudte sætning.

2.9 Nondeterminisme

De processer, vi kan beskrive indtil nu, har alle den egenskab, at deres opførsel er *deterministisk*. Det vil sige, at når alle indlæsninger er foretaget, så kan processen kun opføre sig på én eneste måde. Det virker måske også som en ønskværdig situation, men den er faktisk ikke særligt realistisk, da mange naturlige processer er *nondeterministiske*.

I et stort system med mange samtidige processer bliver de indgående faktorer, der bestemmer systemets opførsel, så indviklede, at processerne mest behændigt kan opfattes som nondeterministiske. Man programmerer derfor ikke ved at forsøge at finde alle de faktorer, der deterministisk bestemmer handlingsforløbet – en iøvrigt helt uoverkommelig opgave. Man prøver i stedet at være så fleksibel, at man kan klare de forskellige situationer, der måtte opstå. Tilsvarende vil vi introducere nondeterministiske processer for at lære at håndtere dem fornuftigt. Vi vælger at indføre nondeterminismen på en måde, der også giver os andre fordele; den tillader os nemlig at undgå at træffe unødvendige beslutninger. Den simple **if**-sætning udvides til

$$
\begin{array}{l}
\textbf{if}\ b_1 \ \rightarrow S_1 \\
\ \mid\ b_2 \ \rightarrow S_2 \\
\ \mid\ \cdots \\
\ \mid\ b_n \ \rightarrow S_n \\
\textbf{fi}
\end{array}
$$

Når denne sætning udføres, bliver alle betingelserne beregnet. Blandt de b_i'er, der er sande, vælges der *nondeterministisk* et tilhørende S_i, der bliver udført. I TRINE er denne nondeterminisme implementeret som en tilfældigt valg blandt de givne muligheder.

Tilsvarende generaliseres **do**-sætningen til

$$\begin{array}{l} \textbf{do}\ b_1 \ \rightarrow S_1 \\ \quad |\ \ b_2 \ \rightarrow S_2 \\ \quad |\ \ \cdots \\ \quad |\ \ b_n \ \rightarrow S_n \\ \textbf{od} \end{array}$$

Her udføres den tilsvarende **if**-sætning gentagent, indtil alle betingelserne er falske.

Disse mekanismer tillader os at være meget præcise, når vi beskriver løsninger. Betragt følgende problem: at beskrive en byttepengeautomat, der ud fra en (ubegrænset) beholdning af 1-, 5- og 10-kroner skal udbetale forskellige beløb. Vi kan skrive denne nondeterministiske sætning

```
(+ 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 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)
   +)
+)
```

Beløbet indlæses og antallet af forskellige mønter udskrives. Der er mange forskellige måder at udbetale et beløb som 27 kr på, og dette afspejles af vores proces, der grundet nondeterminismen potentielt kan vælge blandt dem alle; vi er således ikke gået ud over problemspecifikationen ved at

træffe et umotiveret valg. Vi kunne senere beslutte, at der skal udbetales så få mønter som muligt, men dermed laver vi om på problemet; i så fald kunne vi naturligvis skrive en anden, deterministisk proces, der levede op til denne strengere specifikation.

2.10 Generelle valg

I det helt generelle tilfælde kan man blande determinisme og nondeterminisme i selektioner og iterationer. Man kan skrive et valg som fx

$$
\begin{array}{rl}
b_1 & \rightarrow S_1 \\
\mid b_2 & \rightarrow S_2 \\
\mid b_3 & \rightarrow S_3 \\
\& \, b_4 & \rightarrow S_4 \\
\& \, b_5 & \rightarrow S_5 \\
\mid b_6 & \rightarrow S_6 \\
\& \, b_7 & \rightarrow S_7
\end{array}
$$

hvor & angiver en *sekventiel* sammensætning af de nondeterministiske "grupper". Sætningen udføres på følgende måde. Hvis nogle af b_1, b_2 eller b_3 er sand, så vælges nondeterministisk en af de tilsvarende sætninger; ellers, hvis b_4 er sand, så vælges S_4; ellers, hvis enten b_5 eller b_6 er sand, så vælges en tilsvarende sætning; ellers, hvis b_7 er sand, så vælges S_7. En **if**-sætning udfører valget en enkelt gang; en **do**-sætning udfører valget, sålænge det går godt.

2.11 Pseudoreelle tal

Typen Int er utilstrækkelig hvis man skal repræsentere værdier, der ikke er heltal. Det ville derfor i mange situationer være praktisk at have værdier i stil med

$$17.234, \; \sqrt{7}, \; \log 11, \; \pi$$

som vi kender fra de *reelle* tal.

Der er uendeligt mange heltal, så ved typen Int har vi begrænset os til et endeligt interval. Situationen er dog endnu værre ved de reelle tal, idet der

jo er uendeligt mange reelle tal i ethvert interval med mere end ét punkt. Vi må derfor tynde yderligere ud, for at kunne repræsentere dem.

Disse overvejelser leder os til de såkaldte *pseudoreelle* tal, der faktisk blot er en *endelig* mængde af *rationelle* tal, der udgør værdierne af den atomare type Real i TRINE.

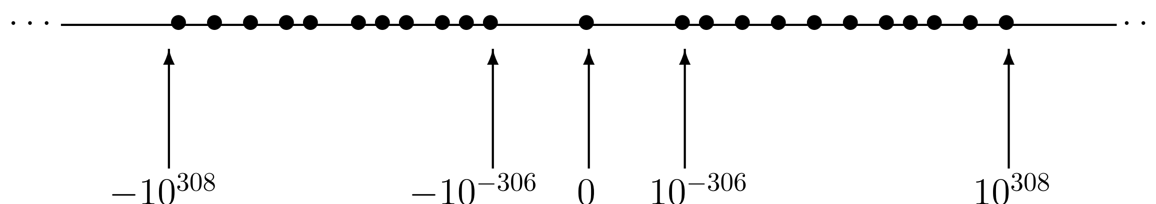
Hvordan man skal vælge denne mængde, samt hvilke egenskaber den har, er et avanceret studie, som vi ikke skal forfølge i denne bog. Det er tilstrækkeligt at erkende, at alle regneoperationer kun kan være approximative, fordi man er nødt til at vælge resultatet blandt de givne pseudoreelle tal.

Vi vil fx observere, at ligningen $\sqrt{7}\sqrt{7} = 7$ *ikke* gælder for pesudoreelle tal. For at få pålidelige resultater, er man derfor nødt til at lære sig et helt nyt sæt regler, der vil blive præsenteret senere i kurset. For helt naive anvendelser kan man dog med sindsro bilde sig ind, at man opererer med de rigtige reelle tal.

Konstanter af typen Real skrives i sædvanlig matematisk notation, så som

```
2
17.234
-11.5
6.023E24
-11.5E214
14E-21
-33.33333E-300
```

Notationen `6.023E24` betyder $6.023 \cdot 10^{24}$. I alt er der ca. $4 \cdot 10^{9}$ tal; absolutværdien af et pseudoreelt tal er enten 0 eller det ligger i intervallet fra 10^{-306} til 10^{308}. De pseudoreelle værdier af typen Real ligger altså blandt de reelle tal på følgende måde



Nedenstående sætning benytter typen Real til at beregne en pseudoreel approximation til π (% angiver division af af pseudoreelle tal, idet vi har reserveret / til heltalsdivision)

```
(+ Var pn, sn, pg, sg: Real
   Var k: Int
   sg, sn, pg, pn, k := 0, 1, 0, 6, 6
   do pn>pg →
        sg, pg := sn, pn
        sn := sn%sqrt(2+sqrt((2+sn)*(2-sn)))
        k := 2*k
        pn := k*sn
   od
   write("Approximation of pi: ", pn%2, eol)
+)
```

2.12 Standardværdier

Alle typer i TRINE har en standardværdi, der bruges til flere forskellige formål.

For det første er standardværdien altid begyndelsesindholdet af en variabel. Det betyder, at man ikke har problemer med udefinerede variabler.

Derudover er standardværdien praktisk i mange situationer, hvor man har brug for en speciel værdi. I forskellige sammenhænge kan det fx give mening at opfatte ?-Int på følgende måder

- hvis h skal angive højden af en person, så kan h=?-Int angive, at højden er ukendt.
- hvis i skal angive nummeret på et element i en liste, så kan i=?-Int angive, at elementet ikke fandtes i listen.
- hvis p skal angive prisen på en vare, så kan p=?-Int angive, at varen ikke kan leveres.

Sådanne situationer optræder meget ofte.

Det er ikke oplagt, hvordan standardværdier kan blandes med “rigtige” værdier. Hvad betyder fx

    2 + ?-Int

I Trine følges det enkle princip, at hvis en operation er mulig for værdier af *alle* typer, så er den også mulig for standardværdier. I modsat fald vil operationen give en fejl, hvis et af dens argumenter er en standardværdi. Det vil sige, at ovenstående addition giver en fejl, hvorimod sammenligningen

    2 = ?-Int

er lovlig (og giver false).

2.13 Oversigt

I det følgende gives en oversigt over operationer på værdier af simple typer.

2.13.1 Heltal

Værdier af type Int skal ligge mellem −2147483647 og +2147483647. Der er fem binære regneoperationer på heltal.

- i+j beregner summen af i og j.
- i-j beregner differensen af i og j.
- i*j beregner produktet af i og j.
- i/j beregner den hele del af i divideret med j.
- i mod j beregner resten af i divideret med j.

Operationerne +, -, *, / og mod kan give fejl ved overløb; / og mod giver desuden fejl, hvis deres andet argument er nul. Der er en enkelt unær regneoperation på heltal.

- `-i` skifter fortegn på i.

Der er fire specielle udtryk af type Int.

- maxint angiver 2147483647.
- `abs(i)` beregner den absolutte værdi af heltallet i.
- `random(i,j)` beregner et pseudotilfældigt heltal i intervallet fra og med i til (men ikke med) j.
- clock angiver antallet af millisekunder, som systemet har kørt.

2.13.2 Sandhedsværdier

Der er to binære regneoperationer på Bool værdier.

- $a \wedge b$ beregner konjunktionen af a og b.
- $a \vee b$ beregner disjunktionen af a og b.

Operationerne $\wedge$ og $\vee$ beregner kun deres andet argument, hvis det er nødvendigt for at afgøre udtrykkets værdi. Der er en enkelt unær regneoperation på Bool værdier.

- $\neg\, b$ beregner negationen af b.

Værdier af alle typer kan sammenlignes indbyrdes med følgende operationer, der giver Bool resultater.

- $x = y$ afgør: lig med.
- $x \neq y$ afgør: forskellig fra.

Værdier af typerne Int, Char og Real kan sammenlignes indbyrdes med følgende operationer, der giver Bool resultater.

- $x < y$ afgør: mindre end.

- $x > y$ afgør: større end.
- $x \leq y$ afgør: mindre end eller lig med.
- $x \geq y$ afgør: større end eller lig med.

Sammenligninger kan skrives i "kæder" så som $a \leq b < c{=}d$, der fortolkes som $(a \leq b) \wedge (b < c) \wedge (c = d)$.

2.13.3 Pseudoreelle tal

Pseudoreelle tal følger IEEE Double Precision standard. Der er fire binære regneoperationer på pseudoreelle tal.

- r+s approximerer summen af r og s.
- r−s approximerer differensen af r og s.
- r*s approximerer produktet af r og s.
- r%s approximerer r divideret med s.

Operationerne `+`, `-`, `*` og `%` kan give fejl ved overløb; `%` giver desuden fejl, hvis dets andet argument er nul. Der er en enkelt unær regneoperation på pseudoreelle tal.

- `-`r skifter fortegn på r.

Der er elleve specielle udtryk der involverer typen `Real`.

- abs`(`r`)` angiver den absolutte værdi af det pseudorelle tal r.
- floor`(`r`)` angiver det største hele pseudoreelle tal, der er mindre end det pseudoreelle tal r.
- ceil`(`r`)` angiver det mindste hele pseudoreelle tal, der er større end det pseudoreelle tal r.
- round`(`r`)` angiver det heltal, der er nærmest det pseudoreelle tal r.

- sqrt(r) angiver en pseudoreel approximation til kvadratroden af det pseudoreelle tal r.
- exp(r) angiver en pseudoreel approximation til den naturlige exponentialfunktion på det pseudoreelle tal r.
- ln(r) angiver en pseudoreel approximation til den naturlige logaritmefunktion på det pseudoreelle tal r.
- sin(r) angiver en pseudoreel approximation til sinus af det pseudoreelle tal r.
- cos(r) angiver en pseudoreel approximation til cosinus af det pseudoreelle tal r.
- arctan(r) angiver en pseudoreel approximation til arctangens af det pseudoreelle tal r.
- rrandom(r,s) angiver et pseudotilfældigt pseudoreelt tal mellem de pseudoreelle tal r og s.

2.14 Beregningsregler

Et almindeligt regneudtryk som

```
2*x+87-17+2
```

er i virkeligheden ret tvetydigt. Nedenfor er nogle af de forskellige mder, det kan fortolkes p.

```
2*(x+87-17)+2
(2*x)+87-(17+2)
2*(x+87)-17+2
(((2*x)+87)-17)+2
```

Vi er dog ikke i tvivl om, at den sidstnvnte mulighed er den rigtige fortolkning. Det skyldes to implicitte forudstninger, som vi her skal gre eksplicitte: *operatorhierarki* og *operatorassociering*.

Operatorerne er ordnet i et hierarki, sledes en operator hjt i hierarkiet binder strkere end alle lavere. I TRINE er hierarkiet organiseret som flger.

*, /, mod, ∧, %, ++
+, -, ∨
<, >, =, ≠

Sledes kan udtrykket

```
2+3*4<17/5
```

nu kun lses som

```
(2+(3*4))<(17/5)
```

Konflikter mellem operatorer p samme niveau i hierarkiet lses ved hjlp af *venstreassociering*, idet man altid skal stte parenteser fra venstre mod hjre. Det vil sige, at udtrykket

```
2+3-4+5-6
```

kun kan lses som

```
(((2+3)-4)+5)-6
```

Endeligt kan man sprge, i hvilken rkkeflge de forskellige deludtryk udregnes? Her er reglen, at udregninger sker fra venstre mod hjre. Det fr betydning, nr vi senere skal se p udtryk med *sideeffekter*.

2.15 Katekismus

△ Værdier klassificeres af typer.

△ De atomare typer er Int, Bool, Char og Real.

△ Enhver type indeholder en standardværdi.

△ Tilstanden for en proces beskrives med en tabel.

- △ Tilstandstabellen knytter navne til variabler (N-søjlen), og variabler til værdier (V-søjlen).
- △ Værdiudtryk angiver værdier relativt til tilstanden.
- △ Variabeludtryk angiver variabler relativt til tilstanden.
- △ Sætninger ændrer tilstanden.
- △ Der er tilordninger, ombytninger, sekvenser, eksterne kommunikationer, selektioner, iterationer og indskudte sætninger.
- △ Indskudte sætninger arbejder med en midlertidigt større tilstand, hvilket vises i tilstandstabellen med en ekstra N-søjle.
- △ Selektioner og iterationer kan være nondeterministiske.
- △ Udtryk beregnes venstreassociativt efter et operatorhierarki.

3 Regulære typer

De fire simple typer fra forrige kapitel er klart utilstrækkelige til at skrive alle de programmer, som vi er interesserede i. Vores ønskeseddel omfatter hensigtmæssige repræsentationer af tekster, vektorer, matricer, mængder, tupler, databaser, bingoplader, selvangivelser, skakbrætter, hierarkier, tabeller, grafer og meget mere. Vi kunne forsøge at tilfredsstille alle disse behov separat, men det ville blive et endeløst arbejde; vi kunne til stadighed udvide sproget med nye typer.

En mere lovende fremgangsmåde er at stille et antal *typekonstruktører* til rådighed, med hvilke man selv kan definere nye typer. I TRINE findes der tre sådanne konstruktører: *lister*, *produkter* og *summer*.

3.1 Navngivne typer

Vi får brug for at kunne definere nye navne på allerede eksisterende typer. Det er i sig selv nyttigt af mnemotekniske årsager, men der er også behov for denne mekanisme til at navngive *nye* typer. Vi indfører definitioner af formen

Type $N = T$

hvor N er et navn og T er en type. Resultatet er, at N bliver et (nyt) navn på T. **Type**-definitioner skrives sammen med **Var**-definitionerne, i begyndelsen af en indskudt sætning. Et eksempel på navngivne typer ses i sætningen

```
(+ Type Penge = Int
   Type Gæld = Penge
   Type Studiegæld = Gæld

   Var s: Studiegæld

   ≪beregn studiegæld≫
+)
```

Her er både Penge, Gæld og Studiegæld synonymer for typen Int. I almindelighed kan **Type**- og **Var**-definitioner blandes vilkårligt.

3.2 Strukturerede mængder

De typekonstruktører, vi indfører, skal bruges til at danne nye typer, som i lighed med de atomare typer skal tilknyttes værdimængder. Disse defineres ved hjælp af de såkaldte *regulære* operationer på mængder, som vi introducerer i dette afsnit.

- Hvis M er en mængde, så er

$$M^*$$

en mængde, der består af endelige *følger* af elementer fra M af formen

$$\begin{array}{rcl} () & - & \text{den tomme følge} \\ (m_0) & - & \text{følger af længde 1} \\ (m_0, m_1) & - & \text{følger af længde 2} \\ (m_0, m_1, m_2) & - & \text{følger af længde 3} \\ & \vdots & \end{array}$$

hvor $\forall i : m_i \in M$. Hvis $M = \emptyset$ så indeholder M^* kun ét element, nemlig den tomme følge; ellers er M^* altid uendelig. Hvis M er endelig, så er der $|M|^i$ følger af længde i.

- Hvis $M_1, M_2, \ldots, M_k$ er mængder, så er

$$M_1 \times M_2 \times \cdots \times M_k$$

deres *produkt*, som er en mængde, der består af elementer af formen

$$(m_1, m_2, \ldots, m_k)$$

hvor $\forall i (1 \leq i \leq k) : m_i \in M_i$. Hvis alle M_i'erne er endelige, så indeholder produktet

$$|M_1| \times |M_2| \times \cdots \times |M_k|$$

elementer.

- Hvis $M_1, M_2, \ldots, M_k$ er mængder, så er

$$M_1 + M_2 + \cdots + M_k$$

deres *sum*, som er en mængde, der består af par af formen

$$(i : m_i)$$

hvor $1 \leq i \leq k$ og $m_i \in M_i$. Summen kaldes også den *disjunkte forening*. Hvis alle M_i'erne er endelige, så indeholder summen

$$|M_1| + |M_2| + \cdots + |M_k|$$

elementer.

3.2.1 Eksempler

Lad $X = \{a\}$ og $Y = \{b, c\}$. Vi har så

$$\begin{aligned}
X^* &= \{(), (a), (a, a), (a, a, a), \ldots, \} \\
Y^* &= \{(), (b), (c), (b, c), (c, b), (b, b, c), \ldots\} \\
X \times Y &= \{(a, b), (a, c)\} \\
Y \times Y &= \{(b, b), (b, c), (c, b), (c, c)\} \\
X + X &= \{(1 : a), (2 : a)\} \\
X + Y &= \{(1 : a), (2 : b), (2 : c)\} \\
Y + Y &= \{(1 : b), (1 : c), (2 : b), (2 : c)\}
\end{aligned}$$

Lad $X = \{d, e\}$, $Y = \{f\}$ og $Z = \{g, h\}$. Vi har så

$$\begin{aligned}
X \times Y \times Z &= \{(d, f, g), (d, f, h), (e, f, g), (e, f, h)\} \\
X + Y + Z &= \{(1 : d), (1 : e), (2 : f), (3 : g), (3 : h)\}
\end{aligned}$$

Lad $X = \{i, j, k\}$ og $Y = \{\}$. Vi har så

$$\begin{aligned}
X \times Y &= \{\} \\
X + Y &= \{(1 : i), (1 : j), (1 : k)\} \\
Y + X &= \{(2 : i), (2 : j), (2 : k)\}
\end{aligned}$$

3.3 Lister

Hvis T er en type, så definerer

Type $L = \mathbf{List}(T)$

en type L, hvis værdimængde er følger af elementer fra T, det vil sige hvor

$$\text{Val}(L) = \text{Val}(T)^* \cup \{?\}$$

Standardværdien af type L angives som ?-L.

Listetypen er en af de vigtigste typer i et programmeringssprog. En liste består af et (dynamisk varierende) antal *indicerede* variabler, hvis identitet angives ved hjælp af listens navn og et *indeks*, der er et ikke-negativt heltal. Hvis en liste x på et tidspunkt indeholder 4 elementer, findes der 4 delvariabler, som angives ved

$x.(0)$, $x.(1)$, $x.(2)$ og $x.(3)$

Bemærk, at nummereringen af listens elementer starter med 0. Selve listens længde angives af Int-udtrykket

$|x|$

3.3.1 Eksempler

Følgende sætning illustrerer en simpel anvendelse af lister med heltal; den indlæser et antal heltal og udskriver det største

```
(+ Type L = List(Int)

    Var x: L
    Var max: Int

    read[x]
    if | x | = 0 → write("The list is empty", eol)
    & true →
        (+ Var i: Int
            i, max := 1, x.(0)
            do i<| x | →
                if x.(i)>max → max := x.(i) fi
                i := i+1
            od
        +)
        write("Max element: ", max, eol)
    fi
+)
```

I eksemplet skabes listens delvariabler i sætningen

```
read[x]
```

som indlæser en række heltal (i standardformatet for lister, jrf. afsnit 3.10), opretter et tilsvarende antal (del)variabler af x og sætter disse til at indeholde de indlæste værdier.

Man kan også oprette delvariabler ved hjælp af en tilordning af formen

x := u

hvor u er et udtryk af type L. Der er flere muligheder for sådanne udtryk. Sætningen

```
x := L(7,9,13)
```

skaber en liste med tre delvariabler, hvis værdier er henholdsvis 7, 9 og 13. Sætningen

x := L(87 | 5)

skaber en liste med 5 delvariabler, der alle har værdien 87. Man kan også sætte lister sammen, hvilket sker med sætningen

x := u_1++u_2

hvor u_1 og u_2 er udtryk af type L. Endeligt kan man udtage en del af en liste på følgende måde

x := u(3 .. 8)

Her bliver x en liste med 5 delvariabler, hvis værdier er komponenterne med indeks 3 til 7 i følgen angivet af u.

Følgende sætninger giver eksempler på anvendelser af disse faciliteter på lister af heltal. Vector er et foruddefineret navn på **List**(Int).

- Variablen x sættes til at indeholde værdien $(0, 1, 2, \ldots, n-1)$

```
(+ Var x: Vector
   Var n: Int
   read [n]
   if n<0 → abort fi
   x := Vector (0 | n)
   (+ Var i: Int
      i := 0
      do i < | x | →
            x. (i) := i
            i := i+1
      od
   +)
   write (x, eol)
+)
```

• Som ovenfor, men ved hjælp af sammensætninger

```
(+ Var x: Vector
   Var n: Int
   read [n]
   if n<0 → abort fi
   x := Vector()
   (+ Var i: Int
      i := 0
      do i < n →
           x := x++Vector(i)
           i := i+1
      od
   +)
   write(x, eol)
+)
```

• En liste indlæses og deles i to lige store dele, samt et eventuelt midterelement, hvis listen har ulige længde

```
(+ Var x, xv, xm, xh: Vector
   read [x]
   xv, xm, xh := x(0 .. | x |/2), Vector(), x(| x |/2 .. | x |)
   if | xv | < | xh | → xm, xh := xh(0 .. 1), xh(1 .. | xh |) fi
   write("venstre halvdel: ", xv, eol)
   write("midterste del: ", xm, eol)
   write("højre halvdel: ", xh, eol)
+)
```

3.3.2 Tilstandstabeller

Vi kan ved brug af tilstandstabeller give en præcis beskrivelse af, hvordan mekanismen med delvariabler virker. Antag, at x er navn på en Vector-variabel, der rummer værdien (11,12,13,14). Vi kan *ikke* beskrive situationen ved hjælp af tilstandstabellen

N	V
x: v_0	v_0: (11,12,13,14)

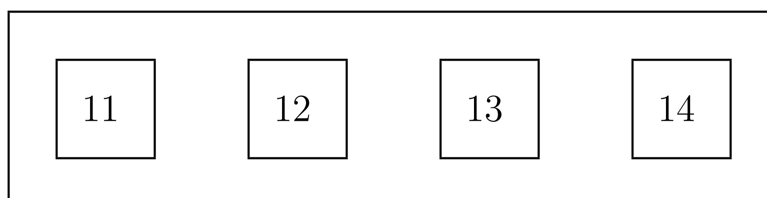
Dette er FORKERT!!

fordi x jo skal indeholde et antal delvariabler, og vi kun kan ændre indholdet af en variabel ved at skifte det helt ud. For at give den ønskede adgang til de enkelte komponenter gemmer vi dem i selvstændige variabler, så tilstanden i stedet bliver

N	V
x: v_0	v_0: (v_1, v_2, v_3, v_4)
	v_1: 11
	v_2: 12
	v_3: 13
	v_4: 14

Dette er RIGTIGT!!

Her ses det klart, at v_0 indeholder fire *delvariabler*, der hver indeholder en af listens komponenter. Når vi aflæser en variabels værdi, skal denne således sammensættes af værdierne af dens delvariabler. I kassografi vil variablen x have følgende udseende



Tilsvarende, når en variabel tilordnes en sammensat værdi, skal der oprettes en samling delvariabler, der kan rumme (kopier af) komponenterne. I tilstanden

N	V
x: v_0	v_0: ?
y: v_1	v_1: (v_2, v_3)
	v_2: 17
	v_3: 22

har y værdien (17,22). Hvis vi udfører sætningen

```
x := y
```

får vi tilstanden

N	V
x: v_0	v_0: (v_4, v_5)
y: v_1	v_1: (v_2, v_3)
	v_2: 17
	v_3: 22
	v_4: 17
	v_5: 22

Bemærk, at der blev oprettet to nye delvariabler, v_4 og v_5. Hvis vi nu udfører sætningen

```
x:=Vector(4)++y
```

bliver resultatet

N	V
x: v_0	v_0: (v_6, v_7, v_8)
y: v_1	v_1: (v_2, v_3)
	v_2: 17
	v_3: 22
	v_6: 4
	v_7: 17
	v_8: 22

Her er værdien af x lig med $(4, 17, 22)$.

3.3.3 Oversigt

Vi kan opsummere listemanipulationerne som følger, idet vi nu betragter den generelle liste

$$\textbf{Type } L = \textbf{List}(T)$$

Standardværdien angives som ?-L. Hvis $u_0, u_1, \ldots, u_k$ er udtryk af type T, så er

$$L(u_0, u_1, \ldots, u_k)$$

et udtryk af type L, hvis værdi er listen af værdier angivet af u_i'erne. Hvis x er et variabeludtryk af type L og u er et udtryk af type Int, så er

$$x.(u)$$

et variabeludtryk af type T; hvis x angiver variablen v og u angiver værdien i, så angiver $x.(u)$ den delvariabel af v, der rummer komponenten med indeks i. Det kræves naturligvis, at listen har en sådan komponent. Man kan benytte forkortelsen

$$x.(u_1, u_2, \ldots, u_n)$$

i stedet for

$$x.(u_1).(u_2).\ldots.(u_n)$$

i forbindelse med lister i flere niveauer. Følgende operationer involverer listeudtryk. Antag, at a er et udtryk af type L med værdi $(a_0, \ldots, a_n)$ og b tilsvarende med værdi $(b_0, \ldots, b_m)$. Så har vi

- |a| er et udtryk af type Int med værdi $n + 1$.
- a$(u_1 \,..\, u_2)$ er et udtryk af type L med værdi $(a_i, \ldots, a_{j-1})$, hvis u_1 og u_2 er Int-udtryk med værdi henholdsvis i og j.
- a++b er et udtryk af type L med værdi $(a_0, \ldots, a_n, b_0, \ldots, b_m)$. Vi har således, at |a++b|=|a|+|b|.
- $L(t \mid u)$ er et udtryk af type L med værdi $(e, e, \ldots, e)$ (n repetitioner), hvis t er et udtryk af type T med værdi e, og u er et Int-udtryk med værdi n.

3.3.4 Typen Text

En bestemt listetype er specielt interessant, nemlig

Type Text = **List**(Char)

Da typen Text anvendes igen og igen, tillades det at anvende den naturlige forkortelse for tekstkonstanter, hvor man kan skrive

```
"Trine"
```

i stedet for udtrykket

```
Text('T','r','i','n','e')
```

Vi kan slutteligt give et lidt større eksempel med tekster. Følgende sætning indlæser en tekst og udskriver alle de *cifferblokke*, som den indeholder. En cifferblok er en maksimal delfølge af listen, der kun indeholder cifre.

```
(+ Var lin: Text
   write("Write a line: ")
   read[lin]
   lin:=lin++Text(eol)
   (+ Var i, d: Int
      i, d:=0, 0
      do 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
   +)
+)
```

3.4 Produkter

Hvis $T_1, T_2, \ldots, T_k$ er typer og $n_1, n_2, \ldots, n_k$ er navne så definerer

$$\textbf{Type}\ P = \textbf{Prod}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k)$$

en type, hvis værdimængde er

$$\mathrm{Val}(P) = \mathrm{Val}(T_1) \times \mathrm{Val}(T_2) \times \cdots \times \mathrm{Val}(T_k) \cup \{?\}$$

Vi kan tænke på et produkt som en *både-og* type. Standardværdien af type P angives som ?-P. Hvis u_i er et udtryk af type T_i, så er

$$P(u_1, u_2, \ldots, u_k)$$

et udtryk af type P, hvis værdi er vektoren af værdier angivet af u_i'erne. Hvis x er et variabeludtryk af type P så, så består x af k delvariabler, og

$$x.n_i$$

er et variabeludtryk af type T_i, der angiver den delvariabel, der rummer n_i-komponenten. Det kræves naturligvis, at indholdet af variablen angivet ved x er forskelligt fra ?-P, således at komponenten faktisk findes. Delvariablerne af en produktvariabel opstår, når den tilordnes en værdi, helt i analogi med lister.

3.4.1 Eksempel

Følgende eksempel anvender en produkttype til at repræsentere tupler i en RASMUS-relation. På basis heraf udregnes gennemsnittet af elevernes eksamenskarakterer.

```
(+ Type Elev = Prod(id, karakter: Int, afgang, egnet: Text)
   Type Relation = List(Elev)

   Var fs: Relation
   Var total: Int

   load[fs]("folkeskole")
   (+ Var i: Int
      i, total := 0, 0
      do i<| fs | →
            total := total+fs.(i).karakter
            i := i+1
      od
write("Gennemsnit: ", total/| fs |, eol)
   +)
+)
```

3.4.2 Tilstandstabeller

Vi skal også i dette afsnit præcisere betydningen af den nye type ved hjælp af tilstandsdiagrammer. Betragt som eksempel definitionen

Type Par = **Prod**(i: Int, b: Bool)

Vi starter med en tom tilstand

N	V

Hvis vi udfører definitionen

Var x: Par

bliver x navnet på en ny variabel, og vi får tilstanden

N	V
x: v_0	v_0: ?

Hvis vi udfører tilordningen

x := Par(87,true)

får vi tilstanden

N	V
x: v_0	v_0: (v_1, v_2)
	v_1: 87
	v_2: true

Nu vil tilordningen

x.i := 555

give tilstanden

N	V
x: v_0	v_0: (v_1, v_2)
	v_1: 555
	v_2: true

og sætningen

x := Par(100, false)

leder os til

N	V
x: v_0	v_0: (v_3, v_4)
	v_3: 100
	v_4: false

Her er værdien af x lig med (100,false). Bemærk, at v_1 og v_2 forsvandt og blev erstattet af nye delvariabler, v_3 og v_4.

3.4.3 Oversigt

Betragt det generelle produkt

$$\textbf{Type}\ P = \textbf{Prod}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k)$$

Standardværdien af type P skrives som ?-P. Hvis u_i er et værdiudtryk af type T_i, så er

$$P(u_1, u_2, \ldots, u_k)$$

et værdiudtryk af type P, hvis værdi er produktet af værdierne angivet af u_i'erne. Hvis x er et variabeludtryk af type P, så er

$$x.n_i$$

et variabeludtryk af type T_i; hvis x angiver variablen v, så angiver $x.n_i$ den delvariabel af v, der rummer n_i-komponenten. Det kræves naturligvis, at indholdet af v ikke er ?-P.

3.5 Summer

Hvis $T_1, T_2, \ldots, T_k$ er typer og $n_1, n_2, \ldots, n_k$ er navne så definerer

$$\textbf{Type } S = \textbf{Sum}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k)$$

en type, hvis værdimængde er

$$\text{Val}(S) = \text{Val}(T_1) + \text{Val}(T_2) + \cdots + \text{Val}(T_k) \cup \{?\}$$

Vi kan tænke på en sum som en *enten-eller* type. Standardværdien af type S skrives som ?-S. Hvis u_i er et udtryk af type T_i, der angiver værdien e_i, så er

$$S(n_i : u_i)$$

et udtryk af type S, hvis værdi er parret $(i : e_i)$. Hvis x er et variabeludtryk af type S, så består x af præcis to delvariabler, hvoraf den første indeholder *varianten* (et heltal mellem 1 og k) og den anden indeholder selve værdien. Således er

$$x.n_i$$

et variabeludtryk af type T_i, der angiver den anden delvariabel. Dette forudsætter naturligvis, at x indeholder en værdi af formen $(i : e_i)$, det vil sige, at den er af *variant* n_i. For at afgøre dette, findes der et Bool-udtryk

$$\textbf{is}(x, n_i)$$

der er sandt, hvis x er af variant n_i og falsk ellers.

3.5.1 Eksempel

En sætning, der bruger summer, er følgende modifikation af gennemsnitseksemplet, hvor der tages højde for, at et eksamensresultat ikke nødvendigvis er en karakter.

```
(+ Type Eksamen = Sum(afmeldt, syg: Unit, karakter: Int)
   Type Elev = Prod(id: Int, resultat: Eksamen, afgang, egnet: Text)
   Type Relation = List(Elev)

   Var fs: Relation
   Var total: Int

   load[fs]("folkeskole")
   (+ Var i,k: Int
      i, total, k := 0, 0, 0
      do i<| fs | →
           if is(fs.(i).resultat, karakter) →
                total := total+fs.(i).karakter
                k := k+1
           fi
           i := i+1
      od
   +)
   write("Gennemsnit: ", total/k, eol)
+)
```

3.5.2 Tilstandstabeller

Betragt typen

Type Eksamen = **Sum**(afmeldt, syg: Unit, karakter: Int)

der beskriver et muligt eksamensresultat. Studenten kan *enten* være afmeldt, *eller* syg, *eller* være gået op og have fået en karakter. Summen tillader os at skelne mellem disse tilfælde og knytte forskellig information til dem. Unit er en type der kun har en enkelt værdi, som altså er standardværdien ?-Unit (se afsnit 13.3 for nærmere detaljer). En sådan énværdi type kan bruges i sumtyper, hvor man kun er interesseret i varianten. Konstanten ?-Unit kan også angives som #.

Lad nu a og b være variabler, der beskriver hvordan Anne og Børge klarede sig til en eksamen. Vi kunne have følgende tilordninger

a := Eksamen(afmeldt:#)
b := Eksamen(karakter:5)

I dette tilfælde meldte Anne fra og Børge dumpede med et femtal. Tilstanden ser ud som følger

N	V
a: v_0	v_0: (v_1, v_2)
b: v_3	v_1: 1
	v_2: #
	v_3: (v_4, v_5)
	v_4: 3
	v_5: 5

Bemærk, at der ikke findes noget variabeludtryk der giver os adgang til delvariablerne v_1 og v_4, der indeholder nummeret på de aktuelle varianter. Disse er specielle, da man kun har adgang til dem på indirekte vis. De kan aflæses gennem **is**-udtrykket, og de kan ændres i forbindelse med en tilordning til hele **sum**-variablen. Disse restriktioner sikrer, at vi altid kan stole på, at varianten angiver den korrekte type af den anden delvariabel.

Hvis vi nu beslutter os til alligevel at lade Børge bestå, kan vi blot udføre tilordningen

b.karakter := 8

Så er tilstanden

N	V
a: v_0	v_0: (v_1, v_2)
b: v_3	v_1: 1
	v_2: #
	v_3: (v_4, v_5)
	v_4: 3
	v_5: 8

hvor vi har de samme delvariabler. Hvis Anne også skal bestå, har vi det problem, at a ikke har nogen karakter-komponent. Vi må i stedet udføre tilordningen

a := Eksamen(karakter:10)

og få tilstanden

N	V
a: v_0	v_0: (v_6, v_7)
b: v_3	v_6: 3
	v_7: 10
	v_3: (v_4, v_5)
	v_4: 3
	v_5: 8

Her blev v_0's delvariabler skiftet ud.

3.5.3 Oversigt

Betragt den generelle sum

$$\textbf{Type}\ S = \textbf{Sum}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k)$$

Standardværdien af type S angives som ?-S. Hvis u_i er et værdiudtryk af type T_i, så er

$$S(n_i : u_i)$$

et værdiudtryk af type S af *variant* n_i. Hvis x er et variabeludtryk af type S, så er

$$x.n_i$$

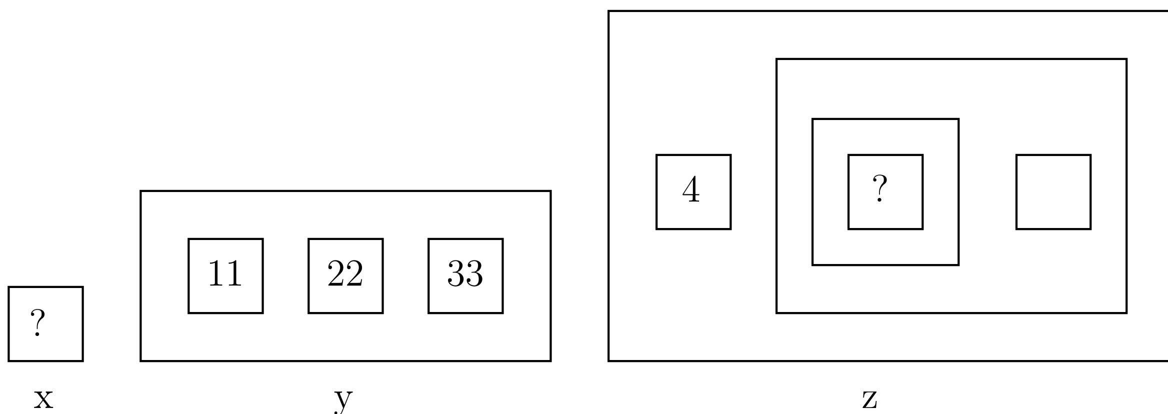
et variabeludtryk af type T_i. Dette giver kun mening hvis værdien af x ikke er ?-S og er af variant n_i. Dette kan man afgøre med Bool-udtrykket **is**(x,n_i).

3.6 Kassografi

Tidligt i kapitel 2 opgav vi at tegne variabler som kasser. Lad os for én gangs skyld se, hvordan en tilstandstabel ser ud i kassografi. Tabellen

N	V
x: v_1	v_1: ?
y: v_2	v_2: (v_4, v_5, v_6)
z: v_3	v_3: (v_7, v_8)
	v_4: 11
	v_5: 22
	v_6: 33
	v_7: 4
	v_8: (v_9, v_{10})
	v_9: (v_{11})
	v_{10}: ()
	v_{11}: ?

bliver til



Der er netop én kasse for hver (del)variabel. Det betyder, at i denne tilstand vil ethvert variabeludtryk angive en af disse 11 mulige kasser.

3.7 Modellering med typer

En vsentlig del af programmering bestr i at benytte typesystemet til at skabe en model af den tilstand, i hvilken handlingen skal foreg. Denne modellering kan vre mere eller mindre detaljeret afhngigt af de aspekter, man lgger vgt p. I de foregende afsnit har vi set, hvordan man gradvist kan forfine typen for en relation.

Her skal vi se en simpel, intuitiv teknik til at komme fra en sproglig beskrivelse af en tilstand til en prcis typedefinition. Man skal i al enkelhed blot bemrke, at produkter svarer til *bde* ... *og* ..., at summer svarer til *enten* ... *eller* ..., og at lister svarer til *et antal* ....

Lad os fx sige, at en studenterrelation bestr af *et antal* tupler, der hver indeholder *bde* et rskort (der er et tal) *og* et navn (der er en tekst) *og* et eksamensresultat, der *enten* er en afmelding, *eller* et sygefravr, *eller* en karakter (der er et tal). I s fald er det en simpel opgave direkte at overstte denne forklaring til typerne

> **Type** Eksamen = **Sum**(afmeldt, syg: Unit, karakter: Int)
> **Type** Student = **Prod**(årskort: Int, navn: Text, res: Eksamen)
> **Type** Relation = **List**(Student)

Vi kunne gre denne model mere detaljeret ved at uddybe, at en afmelding skal have tilknyttet en dato og et sygefravr skal dokumenteres med en diagnose

> **Type** Eksamen = **Sum**(afmeldt: Dato, syg: Diagnose, karakter: Int)

Hvordan beskriver vi en dato? Det simpleste valg er blot som en tekst. En mere detaljeret lsning er at sige, at en dato bestr af *bde* en dag *og* en mned *og* et r, der alle er heltal. Det giver s

> **Type** Dato = **Prod**(dag, mned, r: Int)

En diagnose kunne blot vre *et antal* linjer, der hver er tekster.

> **Type** Diagnose = **List**(Text)

Man kunne dog ogs forestille sig, at en yderligere struktur var pkrvet. Det er oplagt, at man evigt kan blive ved med at forfine sine beskrivelser og samtidigt opn mere og mere indviklede typer. Hvor langt man skal g er helt afhngigt af den aktuelle anvendelse.

3.8 Skabeloner for kontrol

Man opdager hurtigt, at typekonstruktørerne giver anledning til de samme programstumper igen og igen. De kan udtrykkes som nedenstående *skabeloner* for kontrolstrukturer. Vi antager hele tiden, at vi kigger p en variabel, der ikke indeholder en standardvrdi.

3.8.1 Listeskabelon

Hvis x er en generel listevariabel, så får man fat i alle dens komponenter ved at udføre sætningen

```
(+Var i: Int
   i := 0
   do i < | x | →
         ≪behandl x.(i)≫
         i := i+1
   od
+)
```

Hvis man ønsker at behandle listens elementer i *modsat* rækkefølge, så bruges sætningen

```
(+ Var i: Int
    i := | x |
    do i > 0 →
          i := i−1
          ≪behandl x.(i)≫
    od
+)
```

3.8.2 Produktskabelon

Hvis x er en generel produktvariabel, så er det problemfrit at få adgang til dens komponenter. Man angiver dem som $x.n_1, x.n_2, \ldots, x.n_k$.

3.8.3 Sumskabelon

Hvis x er en generel sumvariabel, så er problemet at vide hvilken variant, den er af. Det klares med sætningen

if is(x,n_1) $\rightarrow$ $\ll$behandl $x.n_1$$\gg$
| **is**(x,n_2) $\rightarrow$ $\ll$behandl $x.n_2$$\gg$
| $\cdots$
| **is**(x,n_k) $\rightarrow$ $\ll$behandl $x.n_k$$\gg$
fi

3.8.4 Akkumulatorvariabler

Listeskabelonen har en ofte set familie af specialtilflde, hvor man for en listevrdi

$$(u_0, u_1, \ldots, u_k)$$

nsker at beregne resultatet

$$(\ldots(\phi(u_0) \oplus \phi(u_1)) \oplus \ldots \oplus \phi(u_k))$$

hvor ϕ er en funktion fra vrdimngden af listens elementtype til vrdimngden af en type T, og $\oplus$ er en binr operator p T-vrdier. Hvis $\oplus$ er associativ, s kan ovenstende udtryk skrives uden paranteser. Lad os frst antage, at $\oplus$ har et *neutralt* element ε, sledes at $a \oplus \varepsilon = \varepsilon \oplus a = a$. I s fald kan det nskede udtryk beregnes i variablen a af type T for listen L som

```
(+ Var i: Int
   Var a: T
   i, a:=0, ε
   do i< | L | →
        a:=a+φ(L.(i))
        i:=i+1
   od
+)
```

Bemrk, at den tomme liste giver resultat ε, hvilket stemmer overens med de sdvanlige konventioner. Variablen a kaldes for en *akkumulatorvariabel.*

Hvis $\oplus$ ikke har et neutralt element, s har beregningen ingen vrdi for den tomme liste. Vi m i stedet skrive variationen

```
if | L | = 0 → abort fi
(+ Var i: Int
   Var a: T
   i, a := 1, φ(L.(0))
   do i< | L | →
        a := a+φ(L.(i))
        i := i+1
   od
+)
```

Det flgende er en liste over de mest sdvanlige operatorer og deres neutrale elementer

type	**operator**	**neutralt element**
Int	+	0
Int	*	1
Int	min	maxint
Int	max	-maxint
Bool	$\wedge$	true
Bool	$\vee$	false
List(T)	++	""

Lad os benytte skabelonen i et par tilflde. Hvis vi nsker at beregne lngden af listen, s kan vi vlge $\phi(x) = 1$ og $\oplus =$ +. Hvis vi nsker at vide, om vrdien y optrder i listen, s vlger vi $\phi(x) = (x = y)$ og $\oplus = \vee$.

3.8.5 Eksempel: eksamensresultater

Et eksempel på brugen af samtlige disse skabeloner er følgende sætning, der arbejder med en databaserelation over en årgang studerende, hvis eksamensresultater skal forbedres.

```
(+ Type Eksamen = Sum(afmeldt, syg: Unit, karakter: Int)
   Type Student = Prod(årskort: Int, navn: Text, res: Eksamen)
   Type Relation = List(Student)

   Var r: Relation

   load[r]("rgang97")
   (+ Var i: Int
      i := 0
      do i < | r | →
            if is(r.(i).res, karakter) →
                  if r.(i).res.karakter = 0 →
                        r.(i).res.karakter := 3
                   | r.(i).res.karakter = 3 →
                        r.(i).res.karakter := 5
                   | 5 ≤ r.(i).res.karakter ≤ 10 →
                        r.(i).res.karakter := r.(i).res.karakter+1
                   | r.(i).res.karakter = 11 →
                        r.(i).res.karakter := 13
                  fi
            fi
            i := i+1
      od
   +)
   dump[r]("rgang97")
+)
```

Vi skal senere se, hvordan man kan skrive dette program på en mere overskuelig måde.

3.9 Anonyme konstanter

Konstanter af de regulære typer kan for den erfarne TRINE programmør virke noget omstændelige at angive, på grund af de mange *navne*, man skal skrive. Derfor findes der en simplere notation, de *anonyme* konstanter. Princippet er, at man mere direkte angiver værdierne i den form, der

beskrives i afsnit 3.2. Som vi skal se, kan sådanne udtryk i almindelighed tilskrives mange forskellige typer.

3.9.1 Anonyme standardværdier

Hvis L, P og S er liste-, produkt- og sumtyper, så kan standardværdierne ?-L, ?-P og ?-S også skrives som ?-List, ?-Prod og ?-Sum.

3.9.2 Anonyme lister

Hvis u_i'erne alle er udtryk af samme type T, så er

$$\text{List}\texttt{(}u_0, u_1, \ldots, u_k\texttt{)}$$

et udtryk af type **List**(T). Typen er her entydigt bestemt, bortset fra, at

$$\text{List}\texttt{()}$$

angiver den tomme liste af *enhver* type. Man kan også skrive

$$\text{List}\texttt{(}u\,\texttt{|}\,k\texttt{)}$$

i stedet for

$$\text{List}\texttt{(}u, u, \ldots, u\texttt{)}$$

af længde k.

3.9.3 Anonyme produkter

Hvis u_i er et udtryk af type T_i, så er

$$\text{Prod}\texttt{(}u_1, \ldots, u_k\texttt{)}$$

et udtryk af type **Prod**$(n_1 : T_1, \ldots, n_k : T_k)$ for *ethvert* valg af navne $n_1, \ldots, n_k$.

3.9.4 Anonyme summer

Hvis u er et udtryk af type T, og i er et konstant Int-udtryk, så er

$$\text{Sum}(i : u)$$

et udtryk af *enhver* **Sum**-type, hvis i'te variant er af type T.

3.9.5 Anvendelser

Med den nye notation kan et udtryk som

Relation(Student(971234, "Hugo", Eksamen(afmeldt: #)))

i stedet angives som

List(Prod(971234, "Hugo", Sum(1: #)))

En anden vigtig konsekvens er, at man nu direkte kan angive værdier af en type som

List(**Prod**(x, y: Int))

som fx

List(Prod(1, 2), Prod(3, 4))

Med den traditionelle notation var det nødvendigt at navngive både listen og produktet.

3.10 Kanoniske tekstformater

Man kan bemærke, at enhver værdi kan skrives som en anonym konstant. Atomare konstanter, standardværdier samt Unit-værdien skrives som sædvanligt. Dette danner basis for de *kanoniske tekstformater*, der benyttes

til ekstern kommunikation af værdier. Ved read, write, load og dump kan man netop skrive sådanne anonyme konstanter. Der er dog et par lempelser i forbindelse med tekster. En værdi af type Text kan stadigvæk skrives på formen `"tekst"`, og hvis den ikke er del af en større konstant, så undlader man de omsluttende apostroffer. Det tilsvarende gælder for værdier af type Char.

Det følgende er eksempler på lovlige tekstformater

```
radise
List(0,1,2,3,4)
Prod(87,"radise")
Sum(4:"radise")
Prod(List(11,22,33),Sum(4:"radise"),true)
#
?-Real
?-Prod
6.023E24
```

Følgende grammatik beskriver de lovlige kanoniske tekstformater

Format	::=	**Standard** | **IntConst** | **BoolConst** | **CharConst** | **RealConst** | **TextConst** **ListConst** | **ProdConst** | **SumConst**
Standard	::=	`?-Int` | `?-Bool` | `?-Char` | `?-Real` | `?-Pointer` | `?-List` | `?-Prod` | `?-Sum` | `#` | `nil`
IntConst	::=	**Sign**$^{\circ}$ **Digit**$^{+}$
Sign	::=	`+` | `-`
Digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
BoolConst	::=	`true` | `false`
CharConst	::=	' **Ascii** '
RealConst	::=	**Sign**$^{\circ}$ **Digit**$^{+}$ **Decimals**$^{\circ}$ **Exponent**$^{\circ}$

Decimals	::=	`.` **Digit**$^+$
Exponent	::=	`E` **Sign**$^\circ$ **Digit**$^+$
TextConst	::=	`"` **Ascii*** `"`
ListConst	::=	`List (` **Format**$^{*\lambda}$ `)`
ProdConst	::=	`Prod (` **Format**$^{*\lambda}$ `)`
SumConst	::=	`Sum (` **IntConst** `:` **Format** `)`

Kategorien **Ascii** svarer til alle ASCII tegnene.

3.11 Konverteringer

Ethvert værdiudtryk af type Int kan også bruges som et værdiudtryk af type Real; der vil implicit blive foretaget en konvertering.

Der findes otte eksplicitte konverteringsfunktioner mellem værdier af forskellige typer.

- ic(i) angiver tegnet med ASCII nummer i; det kræves, at i ligger mellem 0 og 255; udtrykket eol er en forkortelse for linjeskifttegnet.
- ci(c) angiver ASCII nummeret på tegnet c.
- bt(b) giver den tekstuelle repræsentation af sandhedsværdien b.
- tb(t) giver en fejl hvis ikke t er lig med true eller false; i modsat fald angives den tilsvarende sandhedsværdi.
- it(i) angiver den tekstuelle repræsentation af tallet i.
- ti(t) beregner heltallet angivet af teksten t; det giver en fejl, hvis ikke t er en tekstuel repræsentation af et lovligt tal.
- rt(r,d,w) angiver den tekstuelle repræsentation af det pseudoreelle tal r med d cifre og bredde w.
- tr(t) beregner det pseudoreelle tal angivet af teksten t; det giver en fejl, hvis ikke t er en tekstuel repræsentation af et lovligt tal.

3.12 Værdier i filer

Der findes fem sætninger, der giver adgang til værdier i filer.

- dump[x](f) skriver værdien af variablen x i kanonisk tekstformat på filen med navn f.
- load[x](f) læser indholdet af filen f som en værdi i variablen x. Dette vil give en fejl, hvis ikke værdien er skrevet i kanonisk tekstformat.
- dumppage[x](f) skriver værdien af variablen x af type List(Text) på filen med navn f. Formatet er som en almindelig tekstfil.
- loadpage[x](f) læser indholdet af filen med navn f som en værdi i variablen x af type List(Text). Hver tekstlinje i filen bliver til et element i listen.
- append(t,f) udvider filen med navn f med en tekstlinje, der indeholder værdien af t af type Text.

3.13 Katekismus

△ Typer kan navngives.

△ Man kan opbygge nye typer ved hjælp af **List**, **Sum** og **Prod**.

△ Værdier af disse typer er sammensatte.

△ En liste giver sekvenser, en sum giver disjunkte foreninger, et produkt giver tupler.

△ En liste svarer til *et antal*.

△ Et produkt svarer til *både-og*.

△ En sum svarer til *enten-eller*.

△ En sammensat værdi gemmes altid i en sammensat variabel.

△ Man kan tilordne værdier til delvariabler individuelt.

△ Alle værdier kan skrives som anonyme konstanter.

△ Ekstern kommunikation foregår ved hjælp af anonyme konstanter.

4 Typeækvivalens

I forbindelse med

- sammenligninger $u_1 = u_2$
- tilordninger $x := u$
- ombytninger $x_1 :=: x_2$

har vi tidligere krævet, at højre- og venstresiderne skulle have *samme* type. Sålænge vi kun bekymrede os om de simple typer, var dette krav naturligt og letforståeligt. Efter at have introduceret de regulære typer, er det ikke længere klart, hvorvidt dette krav er rimeligt – endsige hvad det betyder. Betragt fx følgende definitioner

Type A = Int
Type B = A
Type C = **Prod**(x: A, y: B)
Type D = **Prod**(x: Int, y: A)

Var a: A
Var b: B
Var c: C
Var d: D

Man kan med god ret spørge, om følgende udtryk og sætninger skal være lovlige

- c :=: d
- d := C(b,a)
- b=7

Svaret er i alle tilfælde *ja*, idet vi skal indtage det *liberale* standpunkt, at typer, der tillader samme værdi- og variabeludtryk, så vidt muligt skal være *ækvivalente*. Hermed menes, at de skal kunne erstatte hinanden i alle sammenhænge. Vi angiver at to typer T_1 og T_2 er ækvivalente ved at skrive $T_1 \approx T_2$.

4.1 En kongruensrelation

Vi kan opnå den ønskede ækvivalens ved at beslutte at ignorere de navne på typer, som programmøren indfører i sine definitioner. Det vil sige, at hvis

$$\textbf{Type}\ T = E$$

er en definition, så er $T \approx E$. Det følger heraf, at A $\approx$ Int og at B $\approx$ A. Vi har dog stadig ikke noget belæg for at sige, at B $\approx$ Int. Til dette formål indfører vi den regel, at hvis $T_1 \approx T_2$ og $T_2 \approx T_3$, så vil også $T_1 \approx T_3$; denne regel kaldes *transitivitet*. En anden naturlig regel er, at hvis $T_1 \approx T_2$, så vil også $T_2 \approx T_1$; denne regel kaldes *symmetri*. I samme linje beslutter vi også, at en type altid er ækvivalent med sig selv, det vil sige $T \approx T$; denne regel kaldes *refleksivitet*. En relation, der opfylder disse tre regler, kaldes for en *ækvivalensrelation*.

I vores eksempel ovenfor kan vi stadig ikke konkludere, at C $\approx$ D. Reglerne er ikke stærke nok til at klare dette simple tilfælde. Vi mangler at vide, at de tre typekonstruktører altid bevarer ækvivalens. Det kan vi udtrykke med de følgende ekstra regler. Hvis $T, T_1, T_2, \ldots, T_k$ og $S, S_1, S_2, \ldots, S_k$ er typeudtryk, hvor $T \approx S$ og $T_i \approx S_i$, så har vi også

$$\begin{aligned}
\textbf{List}(T) &\approx \textbf{List}(S)\\
\textbf{Prod}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k) &\approx \textbf{Prod}(n_1 : S_1, n_2 : S_2, \ldots, n_k : S_k)\\
\textbf{Sum}(n_1 : T_1, n_2 : T_2, \ldots, n_k : T_k) &\approx \textbf{Sum}(n_1 : S_1, n_2 : S_2, \ldots, n_k : S_k)
\end{aligned}$$

En relation, der også opfylder disse regler, kaldes for en *kongruens*.

Det er en simpel øvelse at benytte de ovenstående regler til at vise, at C $\approx$ D:

Fra definitionerne har vi at

1) A $\approx$ Int

2) B $\approx$ A

Med kongruensreglen for **Prod** kan vi således slutte at

3) $\textbf{Prod}(\text{x: A, y: B}) \approx \textbf{Prod}(\text{x: Int, y: A})$

Fra definitionen har vi ligeledes

4) $\text{C} \approx \textbf{Prod}(\text{x: A, y: B})$

Fra 3) og 4) kan vi med transitivitet slutte, at

5) $\text{C} \approx \textbf{Prod}(\text{x: Int, y: A})$

Den sidste definition giver os

6) $\text{D} \approx \textbf{Prod}(\text{x: Int, y: A})$

Med symmetri slutter vi

7) $\textbf{Prod}(\text{x: Int, y: A}) \approx \text{D}$

Fra 5) og 7) får vi med transitivitet det ønskede resultat

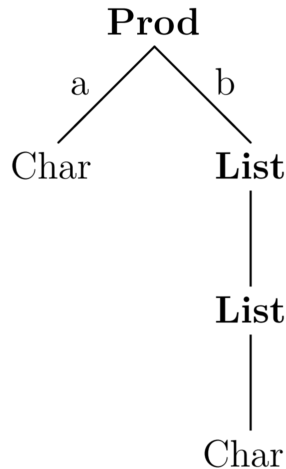
8) $\text{C} \approx \text{D}$

4.2 Normalformer

Vi behøver heldigvis i almindelighed ikke at gennemgå så tunge overvejelser som ovenstående eksempel antyder. For at afgøre om to typer er ækvivalente kan vi blot beregne deres *normalformer*; typerne er så ækvivalente hvis normalformerne er *ens*. Normalformen fremkommer ved at man *udfolder* typedefinitionerne og skriver dem op som træer, i hvilke også rækkefølgen af grenene har betydning. I normalformen optræder der ingen navne på typer. Fx har typen

$$\textbf{Type}\ \text{T} = \textbf{Prod}(\text{a: Char, b: } \textbf{List}(\textbf{List}(\text{Char})))$$

normalformen



I almindelighed beregnes for en type T normalformen $\mathrm{nf}(T)$ på følgende måde

$$\begin{aligned}
&\mathrm{nf}(T) = \mathrm{nf}(E), \text{ hvis } \textbf{Type } T = E\\
&\mathrm{nf}(\mathrm{Int}) = \mathrm{Int}\\
&\mathrm{nf}(\mathrm{Bool}) = \mathrm{Bool}\\
&\mathrm{nf}(\mathrm{Char}) = \mathrm{Char}\\
&\mathrm{nf}(\mathrm{Real}) = \mathrm{Real}
\end{aligned}$$

$$\mathrm{nf}(\mathbf{List}(T)) = \begin{array}{c}\mathbf{List}\\ | \\ \mathrm{nf}(T)\end{array}$$

$$\mathrm{nf}(\mathbf{Prod}(n_1 : T_1, \ldots, n_k : T_k)) = \begin{array}{ccc} & \mathbf{Prod} & \\ n_1 \swarrow & & \searrow n_k \\ \mathrm{nf}(T_1) & \cdots & \mathrm{nf}(T_k)\end{array}$$

$$\mathrm{nf}(\mathbf{Sum}(n_1 : T_1, \ldots, n_k : T_k)) = \begin{array}{ccc} & \mathbf{Sum} & \\ n_1 \swarrow & & \searrow n_k \\ \mathrm{nf}(T_1) & \cdots & \mathrm{nf}(T_k)\end{array}$$

Med denne teknik kan man nu se, at med definitionerne

Type Tekst = **List**(Char)
Type Tegn = Char
Type Streng = **List**(Bogstav)
Type Bogstav = Tegn
Type Page = **List**(Tekst)
Type Blad = **Prod**(a: Tegn, b: **List**(Streng))
Type Ark = **Prod**(a: Bogstav, b: Page)
Type Side = Blad
Type Kra = **Prod**(b: **List**(Streng), a: Bogstav)

så er fx Side og Ark ækvivalente, idet de begge har normalformen angivet ovenfor. Derimod er Kra ikke ækvivalent med Ark (eller Side), fordi b-komponenten nu kommer før a-komponenten.

Man kan let overbevise sig om at denne definition af typeækvivalens har følgende konsekvenser

1) Ækvivalente typer har præcist de samme variabel- og værdiudtryk.

2) Ækvivalente typer har de samme værdimængder.

3) Ækvivalensen er en kongruens.

4.3 Katekismus

△ Typeækvivalens er en kongruensrelation på typer.

△ Ved ombytninger, tilordninger og sammenligninger skal typerne af de to argumenter være ækvivalente.

△ To typer er ækvivalente, hvis deres normalformer er ens.

△ En normalform er et træ, der svarer til en udfoldning af typen.

△ Navne på typer er ikke signifikante, men det er navne på komponenter.

5 Trinvis forfinelse

Efterhånden som programmer opnår en vis størrelse, bliver de hurtigt sværere at læse. Det er imidlertid muligt at skrive et Trine-program i en stil, der klart angiver dets tiltænkte virkemåde; det er i almindelighed en stor hjælp for læseren.

5.1 Ubestemte stumper

I et Trine-program kan man lade forskellige programstumper forblive *ubestemte*. En ubestemt stump ser ud som

≪ubestemt stump≫

og kan stå for enten et *værdiudtryk*, en *sætning*, eller en *type*. Et program, der indeholder ubestemte stumper, kan stadig oversættes og køres. Systemet må dog naturligvis give op, hvis det bliver nødt til at udføre en sådan ubestemt stump.

Man kan senere hen angive indholdet af en ubestemt stump. Det sker i teksten efter selve programmet, ved at man skriver

where ≪ubestemt stump≫ **is** $\cdots$

Indholdet kan igen indeholde ubestemte stumper, der senere kan defineres, og så videre.

Denne programmeringsstil rummer fordele udover en øget læselighed af programteksten. For det første leder den til en meget anbefalelsesværdig teknik til programudvikling, der kaldes *trinvis forfinelse*: et problem løses i mange små og overskuelige skridt, hvor man udtrykker løsningen ved at introducere andre, mere specifikke problemer. For det andet er det praktisk at bevare denne udviklingsstruktur i programteksten, hvis man senere skal modificere programmets virkemåde.

5.2 Eksempel: sikker indlæsning

Som et eksempel på (overdreven) brug af denne teknik vil vi nu udvikle et program, der tillader en sikker indlæsning af et heltal. Vi starter ud med følgende.

```
≪read a number≫
```

Vi skal bruge en heltalsvariabel til at modtage det indlæste tal.

```
where ≪read a number≫ is
        (+ Var n: Int
           ≪read n≫
        +)
```

Hvis vi blot læser tallet direkte, så standser programmet, hvis brugeren skriver andet end et tal. Vi må derfor først indlæse en tekst, der består af cifrene, og derefter selv beregne det angivne tal.

```
where ≪read n≫ is
        (+ Var t: Text
           ≪read digits in t≫
           ≪n:=the number t≫
        +)
```

Vi kører i en løkke, indtil vi har indlæst en tekst, der kun består af cifre.

```
where ≪read digits in t≫ is
        (+ Var digits: Bool
           digits := false
           do ¬ digits →
                read [t]
                ≪digits:=t consists of digits≫
           od
        +)
```

Teksten må ikke være tom, og alle dens elementer skal være cifre.

```
where ≪digits:=t consists of digits≫ is
      if | t | > 0 →  ≪digits:=all t.(i) are digits≫ fi
```

En simpel **do**-løkke checker tekstens elementer.

```
where ≪digits:=all t.(i) are digits≫ is
      digits := true
      (+ Var i: Int
         i := 0
         do i < | t | →
               digits := digits ∧ ≪t.(i) is a digit≫
               i := i+1
         od
      +)
```

Et element er et ciffer, hvis det ligger i det rigtige interval.

```
where ≪t.(i) is a digit≫ is ('0' ≤ t.(i) ≤ '9')
```

Tallet angivet af t flyttes gradvist over til n.

```
where ≪n:=the number t≫ is
      ≪initialize n≫
      ≪move digits from t to n≫

where ≪initialize n≫ is n := 0

where ≪move digits from t to n≫ is
      (+ Var i: Int
         i := 0
         do i < | t | →
               ≪move i'th digit from t to n≫
               i := i+1
         od
      +)
```

Et ciffer flyttes på sædvanlig vis.

where ≪move i'th digit from t to n≫ **is**
n := 10*n+≪t.(i) as a number≫

Tallet angivet af et ciffer kan findes med en simpel formel.

where ≪t.(i) as a number≫ **is** ci(t.(i))-ci('0')

Til sammenligning vil en ekspanderet version af programmet se ud som følger

```
(+ Var n: Int
   (+ Var t: Text
      (+ Var digits: Bool
         digits := false
         do ¬ digits →
               read[t]
               if | t |>0 →
                     digits := true
                     (+ Var i: Int
                        i := 0
                        do i<| t | →
                              digits := digits ∧ ('0' ≤ t.(i) ≤ '9')
                              i := i+1
                        od
                     +)
               fi
         od
      +)
      n := 0
      (+ Var i: Int
         i := 0
         do i<| t | →
               n := 10*n+ci(t.(i))-ci('0')
               i := i+1
         od
      +)
   +)
   write(n)
+)
```

Det er noget mindre overskueligt, og denne forskel vil blive mere markant med større programmer. Hvis vi nu senere opdagede, at TRINE faktisk har

en indbygget facilitet, der oversætter tekster til tal, så kunne vi simplificere vores program på følgende facon

```
where ≪n:=the number t≫ is n:=ti(t)
```

I den ekspanderede version af programmet kunne det tænkes at være sværere at finde det rigtige sted at rette.

5.3 Eksempel: orddeling

Dette afsnit præsenterer et noget større eksempel, der i højere grad demonsterer det reelle behov for trinvis forfinelse.

Problemet er at angive de lovlige delinger af et ikke-sammensat dansk ord. Dialogen kan skitseres som følger

```
(+ Var word: Text
   ≪read word≫
   do | word | ≠ 0 →
        ≪hyphenate≫
        ≪read word≫
   od
+)

where ≪read word≫ is
          write("Write a word: ")
          read[word]
```

Vi kan oversætte denne skitse for at sikre os, at der ikke er nogen fejl på nuværende tidspunkt.

Til selve orddelingen beslutter vi nu at benytte følgende simple regelsæt, der dog ikke er uden undtagelser. Et (ikke-sammensat og ikke-afledt) dansk ord kan deles midt imellem hvert par af nabovokaler. Hvis der står et ulige antal konsonanter mellem to vokaler, så får den sidste flest konsonanter. Følgende er eksempler på anvendelsen af denne regel, idet delingerne angives med bindestreg

tekst: tekst (kan ikke deles)
type: ty–pe
korrekt: kor–rekt
kompleks: kom–pleks
datalogi: da–ta–lo–gi
algoritme: al–go–rit–me
program: prog–ram (undtagelse)

Vi beslutter at finde delepunkterne ved først at opsamle positionerne af ordets vokaler, derefter at beregne de mulige steder der kan deles, og endeligt at komponere resultatet, der skal udskrives til brugeren. Det giver følgende forfinelse

```
where ≪hyphenate≫ is
        (+ Var vowel, hyp: Vector
           Var result: Text
           ≪find positions of vowels≫
           ≪find positions of hyphenations≫
           ≪compute result≫
           write("Hyphenation: ", result, eol)
        +)
```

Positionerne af vokalerne findes med et simpelt **do**-gennemløb af ordet

```
where ≪find positions of vowels≫ is
        (+ Var i: Int
           vowel, i := Vector(), 0
           do i < | word | →
               if ≪word.(i) is a vowel≫ →
                   vowel := vowel++Vector(i)
               fi
               i := i+1
           od
        +)
```

Der er desværre ingen elegant måde at afgøre, om et bogstav er en vokal

```
where ≪word.(i) is a vowel≫ is
        (word.(i) = 'a') ∨ (word.(i) = 'e') ∨
        (word.(i) = 'i') ∨ (word.(i) = 'o') ∨
        (word.(i) = 'u') ∨ (word.(i) = 'y') ∨
        (word.(i) = '') ∨ (word.(i) = '') ∨
        (word.(i) = '')
```

Næst skridt består i at beregne (i vektoren hyp), hvor der kan deles. Vi beslutter at angive de positioner i ordet *før* hvilke, der kan deles. De er positionerne midt mellem to nabovokaler, rundet op, hvilket kan findes som heltalsgennemsnittet af vokalernes positioner plus 1.

```
where ≪find positions of hyphenations≫ is
        (+ Var i: Int
            hyp, i := Vector(), 1
            do i < | vowel | →
                (+ Var mid: Int
                    mid := (vowel.(i-1)+vowel.(i)+1)/2
                    hyp := hyp++Vector(mid)
                +)
                i := i+1
            od
        +)
```

Det sidste skridt er at opbygge resultatet: det delte ord med indsatte bindestreger.

```
where ≪compute result≫ is
        hyp := hyp++Vector(| word |)
        (+ Var i: Int
            result, i := word(0 .. hyp.(0)), 1
            do i < | hyp | →
                result := result++"-"++word(hyp.(i-1) .. hyp.(i))
                i := i+1
            od
        +)
```

De fleste af disse skridt er ikke trivielle. Brugen af trinvis forfinelse er en støtte til både udviklingen og præsentationen af programmet.

5.4 Katekismus

- △ Et program kan indeholde ubestemte stumper.
- △ En ubestemt stump kan stå i stedet for en sætning, et værdiudtryk eller et typeudtryk.
- △ Et program kan oversættes og udføres, selv om det indeholder ubestemte stumper.
- △ En ubestemt stump kan senere blive defineret.
- △ Dette kan benyttes til trinvis forfinelse af programskitser.

6 Eksempel: beregninger på relationer

Dette eksempel viser, hvordan man kan foretage beregninger på relationer, som de kendes fra RASMUS.

6.1 Relationer som typer

Som tidligere set er det let at repræsentere RASMUS relationer i TRINE. Skemaer bliver til produkter, og i stedet for en *mængde* af tupler bruges en *liste* af tupler, hvor man sørger for at undgå dubletter.

Betragt fx RASMUS relationer med følgende skemaer

R:

Name: Text	Age: Int	Id: Int

S:

Id: Int	Employed: Bool

De kan repræsenteres som værdier af følgende TRINE-typer

Type Rtup = **Prod**(Name: Text, Age: Int, Id: Int)
Type Rrel = **List**(Rtup)

Type Stup = **Prod**(Id: Int, Employed: Bool)
Type Srel = **List**(Stup)

Bemærk, at der ikke benyttes en type, hvis værdier er generelle relationer med forskellige skemaer. Vi benytter en speciel type for hvert skema.

6.2 Gamle og nye beregninger

Vi kan nu efterligne de relationelle beregninger, som vi kender fra RASMUS, ved at skrive tilsvarende TRINE-programmer. Til eksempel viser vi beregningen af *join*, der på mange måder er den enkleste, da der ikke er problemer med dubletter af tupler.

Et join af de to ovennænte relationer får følgende type

```
Type Ttup = Prod(Name: Text, Age: Int, Id: Int, Employed: Bool)
Type Trel = List(Ttup)
```

Antag, at variablerne R og S indeholder relationer af typerne Rrel og Srel. Følgende algoritme vil beregne deres join og gemme resultatet i variablen T af type Trel

```
T := Trel()
(+ Var i, j: Int
   i := 0
   do i<| R | →
        j := 0
        do j<| S | →
             if R.(i).Id = S.(j).Id →
                  T := T++Trel(Ttup(R.(i).Name,
                                    R.(i).Age,
                                    R.(i).Id,
                                    S.(j).Employed))
             fi
             j := j+1
        od
        i := i+1
   od
+)
```

Dette er en simpel algoritme, der direkte følger definitionen. Da ++ imidlertid tager lineær tid at udføre, så er dette faktisk en ret ineffektiv implementation. Vi kan gøre det meget bedre ved først at sætte plads af til tuplerne i resultatet.

Det giver i stedet nedenstående program, der først allokerer plads til produktet af antallet af tupler i R med antallet af tupler i S; det er jo en øvre grænse. Derefter konstrueres de joinede tupler, der tælles og anbringes i T. Endeligt skæres T ned til den korrekte størrelse.

```
T := Trel(?-Ttup | | R |*| S |)
(+ Var i, j, antal: Int
   i, antal := 0
   do i<| R | →
      j := 0
      do j<| S | →
         if R.(i).Id = S.(j).Id →
            T.(antal) := Ttup(R.(i).Name,
                              R.(i).Age,
                              R.(i).Id,
                              S.(j).Employed)
            antal := antal+1
         fi
         j := j+1
      od
      i := i+1
   od
   T := T(0 .. antal)
+)
```

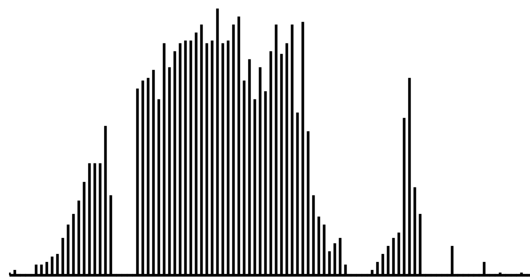
Algoritmen er stadig ikke optimal. Den bedste implementation vil starte med at *sortere* tuplerne og derefter foretage en *fletning*.

Vi kan naturligvis også foretage beregninger, der ikke er mulige i RASMUS. Et eksempel er nedenstående program, der udfra relationen T tegner et simpelt histogram over fordelingen af folk i arbejde i de forskellige aldersklasser fra 0 til 99 år.

```
(+ Var H: Vector
    ≪find height of columns≫
    ≪draw histogram≫
+)

where ≪find height of columns≫ is
            H := Vector (0 | 100)
            (+ Var i: Int
                i := 0
                do i<| T | →
                      if T.(i).Employed →
                            H.(T.(i).Age) := H.(T.(i).Age)+1
                      fi
                      i := i+1
                od
            +)

where ≪draw histogram≫ is
            (+ Var max: Int
                ≪find max height≫
                ≪draw base line≫
                ≪draw columns≫
            +)

where ≪find max height≫ is
            max := 0
            (+ Var i: Int
                i := 0
                do i<| H | →
                      if H.(i) >max → max := H.(i) fi
                      i := i+1
                od
            +)
```

```
where ≪draw base line≫ is
        jumpto(0, 10)
        turnto(0)
        move(2*(| H |-1))

where ≪draw columns≫ is
        turnto(90)
        (+ Var i: Int
           i := 0
           do i<| H | →
                jumpto(2*i, 10)
                move((100*H.(i))/max)
                i := i+1
           od
        +)
```

En kørsel af programmet gav følgende resultat



6.3 Katekismus

- △ Relationer kan systematisk repræsenteres som typer.
- △ Man kan efterlige de sædvanlige relationelle operationer. Derudover kan man lave beregninger, der ikke er mulige i RASMUS.

7 Systematisk afprøvning

Når et program skrives, er det tiltænkt en bestemt opførsel. En typisk programudvikling starter med en mere eller mindre formel specifikation af denne opførsel, det vil sige, en beskrivelse af den ønskede sammenhæng mellem inddata og uddata. Derefter bestemmes en algoritme, der realiserer specifikationen, og endeligt konkretiseres algoritmen som et program.

Selv for tilsyneladende simple specifikationer er der mange fejlkilder i en sådan proces: specifikationen kan være misforstået; algoritmen kan være forkert; en korrekt algoritme kan være implementeret forkert; et rigtigt program kan benytte nogle forkerte primitiver, eller helt banalt være indtastet forkert.

Man er derfor altid nødt til at *afprøve* et nyudviklet program. En afprøvning er essentielt blot et antal kørsler af programmet med forskellige inddata. Man kan derefter sammenholde inddata og uddata for at sikre sig, at de overholder specifikationen.

7.1 Afprøvningens psykologi

Et program skal oftest kunne fungere for *uendeligt* mange forskellige inddata. Enhver afprøvning vil kun kunne køre programmet på *endeligt* mange inddata og vil derfor altid være ufuldstændig. Dette forklarer følgende vigtige regel.

- En afprøvning kan kun påvise *tilstedeværelsen* af fejl – og aldrig *fraværet* af disse.

Man kan derfor sige, at en *succesfuld* afprøvning skal påvise en fejl i programmet. Selv den største mængde af prøvekørsler, der ikke påviser fejl, kan kun i ringe grad underbygge pålideligheden af et program. Statistiske argumenter, som de kendes fra andre slags stikprøvekontrol, kan ikke anvendes her, da fejl ikke behøver at have nogen pæn fordeling over inddata.

Hvis man er sikker på, at programmet er fejlfrit, så er prøvekørsler spild af tid. Hvis man mistænker, at programmet indeholder fejl, så er prøvekørsler

nyttige, men kun hvis de rent faktisk afslører fejlen.

Hvis man selv har skrevet programmet, så er det ikke altid let at dele begejstringen over afsløringen af fejl. Man er tilbøjelig til at håbe, at en given prøvekørsel *ikke* vil afsløre nogen fejl. Der er derfor en stor risiko for, at man, måske ubevidst, er for nænsom under afprøvningen af egne programmer. Fx kan man risikere at vælge et stort antal ensartede prøvedata, der ikke "kommer ud i hjørnerne" af programmet.

En mulighed er at lade andre, mindre følelsesmæssigt involverede, stå for valget af prøvedata. Man kan dog gøre meget selv, ved at anvende en vis objektiv systematik, der præsenteres i de næste afsnit. Det er naturligvis heller ikke korrekt, at fejlfri prøvekørsler ikke i en vis grad kan underbygge tilliden til et program. Det er i hvert fald svært at have tillid til et program, der *aldrig* er blevet afprøvet uden fejl. Det er blot ikke antallet af prøvekørsler, der er afgørende, men snarere at de valgte prøvedata er *repræsentative*. Hvis man ønsker delvist at dokumentere et program ved hjælp af prøvekørsler, så skal en væsentlig del af denne dokumentation være en argumentation for de valgte prøvedata.

7.2 Ekstern afprøvning

Den *eksterne* afprøvning tager sit udgangspunkt i den oprindelige specifikation. Man søger af vælge inddata, der afspejler de forskellige facetter af problemet. Denne afprøvning er uafhængig af det konkrete program og kan genbruges for forskellige implementationer.

Til eksempel kan vi se på programmet fra afsnit 3.3.4, der finder cifferblokke i tekster. Et rimeligt valg af prøvedata kunne være:

- en tom tekst
- en tekst uden cifre: `abcdefg`
- en tekst udelukkende med cifre: `0123456789`
- en tekst, der begynder med en cifferblok: `1234abcd`
- en tekst, der slutter med en cifferblok: `abcd1234`

- en tekst med cifferblokke af længde 1: `a0b1c2d`
- en "almindelig" tekst: `I 1995 er der 14 uger med dProg1`

Bemærk, at der er lagt størst vægt på de "ekstreme" tilfælde, da det erfaringsmæssigt er her de fleste fejl viser sig.

Som et andet eksempel betragter vi et program til beregning af polynomier, der givet en ikke-tom liste af koefficienter $A = (a_n, a_{n-1}, \ldots, a_0)$ og et tal x beregner værdien af $\sum_{i=0}^{n} a_i x^i$. Et rimeligt valg af prøvedata kunne indeholde følgende seks kombinationer af $n = 0$ og $n > 0$ med $x < 0$, $x = 0$ og $x > 0$:

- $A =$ `List(3,2,1)`, $x =$ `0`
- $A =$ `List(3)`, $x =$ `0`
- $A =$ `List(3,2)`, $x =$ `7`
- $A =$ `List(3)`, $x =$ `7`
- $A =$ `List(4,3,2,1)`, $x =$ `-2`
- $A =$ `List(3)`, $x =$ `-2`

7.3 Intern afprøvning

Den *interne* afprøvning tager sit udgangspunkt i det konkrete program. Man søger af vælge inddata, så alle dele af programmet vil blive udført mindst en gang. Det må være et rimeligt minumumskrav for enhver afprøvning. Denne afprøving er helt afhængig af det konkrete program og skal planlægges på ny for hver implementation.

For cifferblokprogrammet er det faktisk nok med en enkelt prøvekørsel med inddata: `ab1234cd`.

Vi betragter derefter et konkret polynomieprogram, der ser ud som følger

```
(+ Var A: Vector
   Var x, r: Int
```

```
    read[A, x]
    if |A| = 0 → abort fi
    if x = 0 → r := A.(| A |-1)
    | x = 1 →
        (+ Var i: Int
           i, r := 1, 0
           do i<| A | →
                r := r+A.(i)
                i := i+1
           od
        +)
    & true →
        (+ Var i: Int
           i, r := 1, A.(0)
           do i<| A | →
                r := x*r+A.(i)
                i := i+1
           od
        +)
    fi
    write(r)
+)
```

Programmet benytter to gunstige specialtilfælde for $x = 0$, hvor resultatet er a_0, og $x = 1$, hvor resultater er $\sum_{i=0}^{n} a_i$, idet multiplikationer her kan undgås. En afprøvning med de ovennævnte eksterne prøvedata vil ikke afsløre nogen fejl. Vi har imidlertid ikke være igennem alle grene af programmet. En intern afprøvning vil også kræve en kørsel med $x = 1$ og fx $A = (1, 2, 3)$. Denne vil give resultatet 5 og dermed afsløre en fejl i programmet: summationen af koefficienterne er forkert; **do**-løkken skal starte med i=0. Dette eksempel viser, hvorfor interne prøvedata er nødvendige som supplement til de eksterne.

Hvis eksterne og interne prøvedata vælges med omhu, så vil der ofte være et stort overlap. En samling prøvedata, der tilfredsstiller disse krav, kan med fordel gemmes til brug ved senere ændringer af programmet.

7.4 Afprøvningens begrænsninger

Der er mange situationer, hvor det at tilrettelægge en rimelig afprøvning er sværere end antydet ovenfor. Det vil blandt andet ske, når programmer opnår realistiske størrelser på hundreder af tusinder af linjer. En komplet intern afprøvning af et sådant program kan være næsten umulig at opnå. Man kan til dels afhjælpe dette ved at afprøve dele af programmet separat.

Der er imidlertid også principielle begrænsninger på afprøvninger. Antag, at vi har skrevet et program til at beregne værdien af summen

$$\sum_{i=1}^{\infty} \frac{1}{i^2}$$

Hvordan kan vi afprøve dette program? Det tager ingen inddata, og vi kan ikke på nogen måde kontrollere, at uddata er korrekt – hvis vi allerede kendte resultatet, så var det jo overflødigt at skrive programmet. En mulig løsning er at lave programmet om, så det i stedet med inddata n beregner

$$\sum_{i=1}^{n} \frac{1}{i^2}$$

Denne sum kan vi for små værdier af n beregne manuelt. Dermed får vi den nødvendige kendte sammenhæng mellem inddata og uddata.

Et andet eksempel er programmer med en nondeterministisk opførsel, som fx veksleprogrammet fra afsnit 2.9. For den eksterne afprøvning er problemet, at uddata ikke afhænger funktionelt af inddata. For den interne afprøvning er problemet, at vi ikke kan garantere, at alle dele af programmet bliver udført. Hvis nondeterminismen er selvforskyldt, som for veksleprogrammet, så kan man vælge at afprøve forskellige deterministiske versioner af programmet. Der kan dog være eksponentielt mange forskellige sådanne versioner. Hvis nondeterminismen påtvinges af programmets eksterne omgivelser, så er problemerne uundgåelige.

Konklusionen bliver, at afprøvninger er et vigtigt værktøj i forbindelse med programudviklinger, men at de aldrig endegyldigt kan garantere et programs korrekthed. Der er endvidere mange situationer, hvor det er vanskeligt overhovedet at tilrettelægge rimelige prøvekørsler. I næste kapitel præsenteres en formel tilgangsvinkel, der kan supplere afprvningerne.

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.
- △ Man kommer langt med lidt sund fornuft.

8 Programmering med udsagn

Programmering ved trinvis forfinelse, som vi s 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 dog stadig 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 mod 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. Udsagnene kan ses som formaliserede kommentarer, der kan benyttes til at eftervise programmets korrekthed *inden* det udfres. P den mde er det ogs et vrdifuldt supplement til den systematiske afprvning.

8.1 Udsagn

Et *udsagn* er enten sand eller falsk i en given tilstand. Som sdan minder det meget om et udtryk af type Bool, der jo netop beregner til true eller false i en aktuel tilstand. Et udsagn er dog mere generelt, da det ikke ndvendigvis skal kunne skrives i TRINE syntaksen. Vi vil faktisk tillade os at bruge al den sdvanlige notation fra matematikken, nr vi formulerer udsagn.

Antag, at vi betragter en tilstand, der beskrives af flgende definitioner

```
(+ Var x, y: Int
   Var t: Text
   ...
+)
```

Dette er s eksempler p udsagn over denne tilstand

a) x+y $\leq$ |t|

b) x=3

c) (x≤y) $\Rightarrow$ |t|=x-1

d) $\exists i \geq 0$: ci(t.(i))=y

e) 2+2<87

Udsagnene er af forskellig karakter. Eksemplerne a) og b) er faktisk blot TRINE-udtryk af type Bool. Eksemplerne c) og navnlig d) er matematiske udsagn, der ikke direkte kan skrives i TRINE. Det er dog stadig klart, at deres sandhedsvrdi afhnger af den aktuelle tilstand. Eksempel e) er et udsagn, hvis sandhedsvrdi (sand) er uafhngig af tilstanden. Sdanne konstante udsagn har vi ikke megen brug for, men de tilfredsstiller dog kravene i vores definition.

Her er en rkke tilstandstabeller og for hver af disse en angivelse af de ovenstende udsagns sandhedsvrdier.

N	V
x: v_0	v_0: 1
y: v_1	v_1: 2
t: v_3	v_3: (v_4, v_5, v_6)
	v_4: a
	v_5: b
	v_6: c

a)	b)	c)	d)	e)
sand	falsk	falsk	falsk	sand

N	V
x: v_0	v_0: 3
y: v_1	v_1: -1
t: v_3	v_3: (v_4, v_5)
	v_4: a
	v_5: b

a)	b)	c)	d)	e)
sand	sand	sand	falsk	sand

N	V
x: v_0	v_0: 0
y: v_1	v_1: 87
t: v_3	v_3: (v_4, v_5, v_6)
	v_4: a
	v_5: W
	v_6: c

a)	b)	c)	d)	e)
falsk	falsk	falsk	sand	sand

8.2 Gyldige udsagn og programmer

Et frste skridt i teknikken er at nedskrive udsagn forskellige steder i vores programmer. Programmer, der indeholder sdanne udsagn, kaldes *dekorerede*. Et sdan udsagn vil altid udtale sig om den tilstanden det pgldende sted i programmet.

Vi kalder et udsagn i et dekoreret program for *gyldigt*, hvis det altid er sandt i enhver tilstand det mder i enhver udfrelse af programmet. Denne definition er mere subtil end man umiddelbart skulle tro. I de flgende sm eksempler har vi sledes indrammet de gyldige udsagn. Frst ser vi to simple eksempler p gyldige og ugyldige udsagn.

```
(+ Var x: Int
   x := 0
   { x = 0 }
+)
```

```
(+ Var x: Int
   x := 1
   { x = 0 }
+)
```

Vi observerer dernst, at vi ikke kan sige noget specifikt om indlste vrdier, medmindre de eksplicit checkes.

```
(+ Var x: Int
   read [x]
   { x = 0 }
+)
```

```
(+ Var x: Int
   read [x]
   if x ≠ 0 → abort fi
   { x = 0 }
+)
```

Gyldigheden kan ogs pvirkes af, om udsagnet kun kan ns i bestemte tilstande.

```
(+ Var x: Int                (+ Var x: Int
     read [x]                     read [x]
     if x = 0 →                   do x ≠ 0 → skip od
          skip                    { x = 0 }
          { x = 0 }          +)
     fi
+)
```

Endeligt er et udsagn, der ikke kan ns, altid gyldigt.

```
(+ Var x: Int                (+ Var x: Int
     read [x]                     read [x]
     do true → skip od            if true →
     { x = 0 }                         skip
+)                                     { x = 0 }
                                  | false →
                                       skip
                                       { x = 0 }
                                  fi
                             +)
```

Vi kalder et dekoreret program for *gyldigt*, hvis alle dets udsagn er gyldige. Hvis et gyldigt program indeholder interessante udsagn, s er det ofte en smal sag at udtale sig om programmets korrekthed. Antag fx, at den flgende skitse af et sorteringsprogram kan vises at vre gyldig.

```
(+ Var L: List (Int)
     read [L]
     ⋮
     {∀i, j : 0 ≤ i ≤ j < |L| ⇒ L.(i) ≤ L.(j)}
     write (L)
+)
```

Da udsagnet lige inden write-stningen er gyldigt, s ved vi, at hvis der udskrives noget, s er det en sorteret liste. Problemet er nu naturligvis at bevise, at sdanne dekorerede programmer er gyldige.

8.3 Bevis af gyldighed

At vise gyldighed af et program kan reduceres til at foretage almindelige matematiske beviser. Vi udvikler her denne teknik i tre skridt.

8.3.1 Bevisteknikken

At et program kan vises gyldigt, er kun interessant, hvis det indeholder udsagn der siger noget om dets virkemde. Betragt flgende program, der lser andengradsligninger.

```
(+ Var a, b, c, d, x1, x2: Real
   read[a, b, c]
   if a = 0 → abort("ledende koefficient er nul") fi
   d := b*b-4*a*c
   if d<0 → abort("imaginre rdder") fi
   x1 := (-b+sqrt(d))%(2*a)
   x2 := (-b-sqrt(d))%(2*a)
   write(x1, eol, x2)
+)
```

Vi vil gerne kunne vise gyldighed af den dekorerede udgave

```
(+ Var a, b, c, d, x1, x2: Real
   read[a, b, c]
   if a = 0 → abort("ledende koefficient er nul") fi
   d := b*b-4*a*c
   if d<0 → abort("imaginre rdder") fi
   x1 := (-b+sqrt(d))%(2*a)
   x2 := (-b-sqrt(d))%(2*a)
   { ax1²+bx1+c = 0 ∧ ax2²+bx2+c = 0 }
   write(x1, eol, x2)
+)
```

Men som vi udmrket ved, s kan den anvendte formel kun etablere dette, hvis determinanten ikke er negativ og den ledende koefficient ikke er nul.

Derfor m vi arbejde med flgende udgave, hvor slutudsagnet kun skal vises under en betingelse, der udtrykkes ved et tidligere udsagn

```
(+ Var a, b, c, d, x1, x2: Real
   read[a, b, c]
   if a = 0 → abort("ledende koefficient er nul") fi
   d:= b*b-4*a*c
   if d<0 → abort("imaginre rdder") fi
   { d = b²-4ac ∧ d ≥ 0 ∧ a ≠ 0 }
   x1:= (-b+sqrt(d))%(2*a)
   x2:= (-b-sqrt(d))%(2*a)
   { ax1²+bx1+c = 0 ∧ ax2²+bx2+c = 0 }
   write(x1, eol, x2)
+)
```

Dette tidligere udsagn kan igen vises under betingelse af et tidligere udsagn, og s videre. Det fuldt dekorerede program ser ud som flger.

```
(+ Var a, b, c, d, x1, x2: Real
   read[a, b, c]
   if a = 0 → abort("ledende koefficient er nul") fi
   { a ≠ 0 }
   d:= b*b-4*a*c
   { d = b²-4ac ∧ a ≠ 0 }
   if d<0 → abort("imaginre rdder") fi
   { d = b²-4ac ∧ d ≥ 0 ∧ a ≠ 0 }
   x1:= (-b+sqrt(d))%(2*a)
   x2:= (-b-sqrt(d))%(2*a)
   { a*x1²+b*x1+c = 0 ∧ a*x2²+b*x2+c = 0 }
   write(x1, eol, x2)
+)
```

I de flgende afsnit ser vi, hvordan man p denne mde kan kde udsagn sammen og udtale sig om programmers virkemde.

8.3.2 Sekvenser

Betragt en sekvens af simple stninger, omgivet af udsagn

$$\{U\}\ S_1\ S_2 \ldots S_n\ \{V\}$$

Denne sekvens er *betinget* gyldig, sfremt der glder, at hvis U er gyldig, s er V ogs gyldig.

Antag for simpelheds skyld, at tilstanden omfatter variablerne x, y og z. Stningerne S_1 S_2 $\ldots S_n$ vil fre frem til en ny stilstand, i hvilken disse variabler antager nye vrdier. I denne teknik vil vi benytte notationen x', y' og z' til at angive variablernes vrdier i denne nye tilstand. Hver af disse mrkede variabler afhnger af x, y og z p en mde, som vi kan regne ud ved at betragte stningerne i den givne sekvens. Nr denne sammenhng er bestemt, s skal vi nu blot vise

$$U(\mathrm{x},\mathrm{y},\mathrm{z}) \Rightarrow V(\mathrm{x}',\mathrm{y}',\mathrm{z}')$$

Men da x', y' og z' er bestemt af x, y og z, kan vi indstte i formlen og ende med et almindeligt matematisk bevis, som vi forhbentlig kan gennemfre. Betragt flgende lille eksempel

```
{ x+y = z }
x := x+y
y := x+z
x := : z
{ y = 2*z }
```

For at vise gyldighed skal vi frst bestemme sammenhngen mellem tilstandene fr og efter sekvensen. Det er ret let at se, at den er

$$\mathrm{x}'=\mathrm{z},\ \mathrm{y}'=\mathrm{x+y+z},\ \mathrm{z}'=\mathrm{x+y}$$

Vi skal derfor vise

$$\mathrm{x+y} = \mathrm{z} \Rightarrow \mathrm{y}' = 2*\mathrm{z}'$$

Ved indsttelse fr vi

$$x+y = z \Rightarrow x+y+z = 2*(x+y)$$

der ved almindelige udregninger let ses at glde.

8.3.3 Selektioner

Hvis vi skal vise gyldighed af et program, der indeholder en selektion, s er teknikken fra sidste afsnit ikke tilstrkkelig. Vi kan jo ikke vide, hvilken gren, der vil blive udfrt i flgende dekorerede stning

$$\begin{array}{l} \{U\} \\ \textbf{if}\ b_1 \ \rightarrow S_1 \\ \ |\ b_2 \ \rightarrow S_2 \\ \ |\ \cdots \\ \ |\ b_n \ \rightarrow S_n \\ \textbf{fi} \\ \{V\} \end{array}$$

Lsningen er at gennemfre et separat bevis for hver gren for sig. Vi kan dog udnytte, at en stning kun kan blive udfrt, hvis den tilsvarende betingelse er sand. Vores **if**-stning er derfor betinget gyldig, sfremt flgende n individuelle stninger alle er betinget gyldige

$$\{U \wedge b_i\}\ S_i\ \{V\}$$

og

$$U \wedge \neg b_1 \wedge \neg b_2 \wedge \ldots \wedge \neg b_n \Rightarrow V.$$

De frste kan s klares med teknikken for sekvenser. Den sidste tager hensyn til det tilflde, hvor alle betingelser er falske og intet S_i bliver udfrt. Betragt flgende eksempel, der udskriver maksimum af to indlste tal

```
(+ Var x, y, z: Int
    read[x, y]
    if x ≥ y → z := x
    | x ≤ y → z := y
    fi
    { z = max(x, y) }
    write(z)
+)
```

Vi skal vise den betingede gyldighed af stningen

```
{ true }
if x ≥ y → z := x
| x ≤ y → z := y
fi
{ z = max(x, y) }
```

hvilket reducerer til at vise betinget gyldighed af stningerne

$$\{x \geq y\}\ z := x\ \{z = \max(x, y)\}$$
$$\{x \leq y\}\ z := y\ \{z = \max(x, y)\}$$

For den frste har vi, at x'=x, y'=y og z'=x. Vi skal derfor vise

$$x \geq y \Rightarrow z' = \max(x', y')$$

der efter indsttelse er det samme som

$$x \geq y \Rightarrow x = \max(x, y)$$

der ungteligt er sandt. Den anden sekvens vises betinget gyldig p tilsvarende vis.

8.3.4 Iterationer

For programmer med iterationer fr vi endnu en komplikation, da vi ikke ved hvor mange gange iterationer, der udfres. Betragt flgende stning

$\{U\}$
do $b_1 \ \rightarrow S_1$
$\quad | \ \ b_2 \ \rightarrow S_2$
$\quad | \ \ \cdots$
$\quad | \ \ b_n \ \rightarrow S_n$
od
$\{V\}$

Det betingede gyldighedsbevis krver denne gang en strre grad af indsigt. Vi har brug for at forst **do**-stningens virkemde og beskrive den med en *invariant*, som er et udsagn, der et sandt fr og efter ethvert gennemlb af lkken. Antag, at vi har fundet en sdan invariant $\{I\}$. Vi skal nu gennemfre flgende samling af beviser

$U \ \Rightarrow \ I$ (basis)

$\{I \wedge b_i\} \ S_i \ \{I\}$ er betinget gyldig (invarians)

$I \wedge \neg b_1 \wedge \ldots \wedge \neg b_n \ \Rightarrow \ V$ (konklusion)

Kunsten er at finde p en invariant, der gr ovenstende muligt at vise. I eksempler vil vi skrive invarianten umiddelbart efter ngleordet **do**, hvilket indikerer, at den mdes fr og efter ethvert gennemlb.

Lad os gennemfre tre mindre gyldighedsbeviser for **do**-stninger. Det frste er det lille simple program fra afsnit 2.7 til beregning af potenssummen $\mathrm{x}^{\mathrm{n}} + \mathrm{x}^{\mathrm{n}-1} + \cdots + \mathrm{x} + 1$. Her er invarianten meget simpel: p et givet tidspunkt har vi beregnet en del af det nskede polynomie. I dekoreret form ser programmet ud som flger

```
(+ 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)
+)
```

Hvis dette program er gyldigt, s har vi beregnet det rigtige resultat inden udskrivningen. Udsagnet $\{ n \geq 0 \}$ er oplagt gyldigt p grund af **if**-stningen lige inden. Lad os gennemg de tre delbeviser for **do**-stningens gyldighed.

For at etablere basis, m vi i dette tilflde vise betinget gyldighed af sekvensen

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

Tilordningen har effekten, at x'=x, n'=n, psum'=1 og m'=0. Vi skal med andre ord vise, at

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

hvilket efter indsttelse er det samme som

$$n \geq 0 \Rightarrow (1 = 1) \wedge (0 \leq 0 \leq n)$$

der oplagt glder. For at etablere invariansen, skal vi nu vise betinget gyldighed af

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

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

Effekten af tilordningerne er x′=x, n′=n, psum′=psum*x+1 og m′=m+1. Vi skal sledes vise

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

hvilket efter indsttelse er det samme som

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

der kan regnes om til

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

der let ses at vre sandt. For slutteligt at etablere konklusionen skal vi vise

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

men det er blot

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

der jo ogs er let at se. Her har vi vret meget detaljerede omkring et ganske simpelt program. I almindelighed kan man lgge et noget hjere abstraktionsniveau.

Vi prver nu med veksleprogrammet fra afsnit 2.9. Her er det i en dekoreret udgave, hvor invarianten siger, at vi aldrig smider penge vk.

```
(+ Var amount: Int
   read[amount]
   if amount<0 → abort fi
   { amount ≥ 0}
   (+ 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)
   +)
+)
```

Vi gr lidt hurtigere frem denne gang. Basis kommer fra

```
{ amount ≥ 0 }
k1, k5, k10, rest:=0, 0, 0, amount
{ (amount=k10*10+k5*5+k1+rest) ∧ (rest≥0) }
```

Effekten af tilordningen er k_1'=0, k_5'=0, k_{10}'=0 og rest′=amount, s efter indsttelse skal vi vise

$$\text{amount} \geq 0$$
$$\Downarrow$$
$$(\text{amount} = 0*10+0*5+0+\text{amount}) \wedge (\text{amount} \geq 0)$$

hvilket er sandt. Invariansen skal vi vise i tre tilflde. Vi gennemgr her det frste

$$\{\ (\texttt{amount} = k_{10}*10+k_5*5+k_1+\texttt{rest}) \wedge (\texttt{rest} \geq 0) \wedge\ (10 \leq \texttt{rest})\ \}$$
$$k_{10},\ \texttt{rest} := k_{10}+1,\ \texttt{rest}-10$$
$$\{\ (\texttt{amount} = k_{10}*10+k_5*5+k_1+\texttt{rest}) \wedge (\texttt{rest} \geq 0)\ \}$$

Effekten er $k_{10}'=k_{10}+1$ og $\texttt{rest}'=\texttt{rest}-10$ (vi nvner fremover kun de variabler, der ndres). Vi skal derfor efter indsttelse vise

$$(\texttt{amount} = k_{10}*10+k_5*5+k_1+\texttt{rest}) \wedge (\texttt{rest} \geq 0) \wedge (10 \leq \texttt{rest})$$
$$\Downarrow$$
$$(\texttt{amount} = (k_{10}+1)*10+k_5*5+k_1+\texttt{rest}-10) \wedge (\texttt{rest}-10 \geq 0)$$

der flger af et par simple omskrivninger. Konklusionen krver

$$(\texttt{amount} = k_{10}*10+k_5*5+k_1+\texttt{rest}) \wedge (\texttt{rest} \geq 0) \wedge$$
$$\neg(10 \leq \texttt{rest}) \wedge\ \neg(5 \leq \texttt{rest}) \wedge\ \neg(1 \leq \texttt{rest})$$
$$\Downarrow$$
$$\texttt{amount} = k_{10}*10+k_5*5+k_1$$

men det er det samme som

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

hvilket er sandt, s vi er frdige.

Endeligt kan vi betragte flgende program, der sorterer k indlste tal

```
(+ Var x_1, x_2, ... x_k: Int
   read[x_1, x_2, ... x_k]
   do { true }
       x_1>x_2 → x_1:=:x_2
    & x_2>x_3 → x_2:=:x_3
      ⋮
    & x_{k-1}>x_k → x_{k-1}:=:x_k
   od
   {x_1 ≤ x_2 ≤ ... ≤ x_k}
   write(x_1, " ", x_2, " ",...," ", x_k)
+)
```

Bemrk, at vi tilsyneladende slet ikke behver at beskrive tilstanden, da invarianten jo er true! Derfor er beviserne for basis og invarians trivielle. Konklusionen flger imidlertid af

$$\text{true} \wedge \neg(x_1 > x_2) \wedge \neg(x_2 > x_3) \wedge \ldots \wedge \neg(x_{k-1} > x_k)$$

$$\Downarrow$$

$$(x_1 \leq x_2) \wedge (x_2 \leq x_3) \wedge \ldots \wedge (x_{k-1} \leq x_k)$$

Dette bevis, der er helt efter reglerne, virker lidt mistnkeligt, idet vi ikke engang har skelet til hjresiderne af **do**-stningen. I nste afsnit skal vi se, hvilket aspekt vi mangler at overveje.

8.4 Bevis af terminering

De tre beviser for **do**-stninger i det foregende afsnit er alle mangelfulde som argumenter for programmernes korrekthed. Vi har nemlig kun vist, at *hvis* programmerne terminerer, s opns der en nskvrdig tilstand. Men vi kan rent faktisk ikke vide, at de terminerer i alle situationer. I dette afsnit introducerer vi en teknik til at bevise denne egenskab for **do**-stninger.

8.4.1 Termineringsfunktioner

Betragt en **do**-stning med en gyldig invariant

$$
\begin{array}{l}
\textbf{do}\ \{\ I\ \} \\
\quad\ \ b_1\ \rightarrow S_1 \\
\ \ |\ \ b_2\ \rightarrow S_2 \\
\ \ |\ \ \cdots \\
\ \ |\ \ b_n\ \rightarrow S_n \\
\textbf{od}
\end{array}
$$

Antag for simpelheds skyld, at tilstanden omfatter variablerne x, y og z. En *termineringsfunktion* for **do**-stningen er en heltallig funktion μ(x,y,z), med flgende egenskaber

$$I \Rightarrow \mu(\mathrm{x,y,z}) \geq 0$$
$$I \wedge b_i \Rightarrow \mu(\mathrm{x',y',z'}) < \mu(\mathrm{x,y,z})$$

hvor x′, y′og z′ er effekten af stningen S_i. Vi bruger sledes μ til at mle, hvor langt den aktuelle tilstand er fra sluttilstanden. De to egenskaber siger, at mlet aldrig kan komme under nul men samtidig aftager ved hvert gennemlb. Det flger, at **do**-stningen m terminere.

8.4.2 Tre eksempler

Vi viser nu terminering af de tre **do**-stninger fra det foregende afsnit. For programmet til beregning af potenssummen benytter vi termineringsfunktionen

$$\mu(\mathrm{x,psum,n,m}) = \mathrm{n-m}$$

Vi skal vise de to egenskaber, der reducerer til

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

og

$$
\begin{array}{ll}
 & (\mathrm{psum} = \mathrm{x}^{\mathrm{m}} + \mathrm{x}^{\mathrm{m}-1} + \cdots + \mathrm{x} + 1) \wedge (0 \leq \mathrm{m} \leq \mathrm{n}) \wedge \mathrm{m} \neq \mathrm{n} \\
\Downarrow & \\
 & (\mathrm{n}-(\mathrm{m}+1) < \mathrm{n}-\mathrm{m})
\end{array}
$$

der let kan ses at glde. Vi kan derfor slutte, at programmet terminerer og dermed beregner det korrekte resultat.

Veksleprogrammet er ligeledes enkelt, nr vi vlger termineringsfunktion

$$\mu(\mathrm{amount},\mathrm{k}_1,\mathrm{k}_5,\mathrm{k}_{10},\mathrm{rest}) = \mathrm{rest}$$

For sdanne simple situationer vil man sjldent give et fuldstndigt formelt bevis, men blot observere, at vrdien af rest bliver reduceret i hvert af **do**-stningens tre grene.

Det er dog ikke altid s simpelt, at vise terminering. For sorteringsprogrammet er det klart, at hele kompleksiteten m ligge gemt i dette bevis. Vi skal da ogs vlge en noget mere indviklet termineringsfunktion

$$\mu(\mathrm{x}_1,\mathrm{x}_2,\ldots,\mathrm{x}_k) = |\ \{(i,j) \mid i < j\ \wedge\ \mathrm{x}_i > \mathrm{x}_j\}\ |$$

Det, vi mler, er antallet af *inversioner*, det vil sige par af tal, der str forkert i forhold til hinanden. Da dette er strrelsen af en mngde, er det oplagt af μ altid er positiv. Tilbage er at vise, at hver gren reducerer vrdien af μ. Et typisk tilflde er

$$\mathrm{x}_i > \mathrm{x}_{i+1} \Rightarrow \mu(\mathrm{x}_1,\ldots,\mathrm{x}_i,\mathrm{x}_{i+1},\ldots,\mathrm{x}_k) > \mu(\mathrm{x}_1,\ldots,\mathrm{x}_{i+1},\mathrm{x}_i,\ldots,\mathrm{x}_k)$$

Det er oplagt, at vi fjerner en inversion, nemlig $(i, i+1)$. Samtidig laver vi alle inversioner af formen (γ, i) om til $(\gamma, i+1)$ og alle af formen $(i+1, \gamma)$ om til (i, γ). Alt i alt falder antallet af inversioner med en. Vi konkluderer, at sorteringsprogrammet terminerer og derfor er korrekt.

8.5 Korrekthed

Vi kan nu give den formelle definition af *korrekthed*, der stemmer overens med vore intuitive forstelse. Et (dekoreret) program er korrekt, hvis det er gyldigt og terminerer.

8.6 Eksempel: Euklids algoritme

I dette afsnit beviser vi korrektheden af en algoritme, der ikke er helt triviel. Vi illustrerer samtidig den noget lsere stil, som man mere naturligt vil benytte i praksis.

Det handler om den udvidede Euklids Algoritme, der beregner største fælles divisor og mindste fælles multiplum af to tal.

```
(+ Var n, m: Int
   write("Indtast n og m: ")
   read[n, m]
   if (n<1) ∨ (m<1) → abort("Kun positive tal!") 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
      { (p = q = sfd(n,m)) ∧ (2*mfm(n, m) = s+t) }
      write("Strste flles divisor: ", p, eol)
      write("Mindste flles multiplum: ", (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)

$$I\colon\ (p*s+q*t = 2*sfd(n,\ m)*mfm(n,\ m)) \land$$
$$(sfd(p,\ q) = sfd(n,\ m)) \land (p,\ q \geq 1)$$

og det er den tilhørende termineringsfunktion også

$$\mu(n,m,p,q,s,t) = p+q$$

Vi viser frst basis. Det følger af definitionen på største fælles divisor og mindste fælles multiplum, at der for vilkårlig positive heltal n og m gælder

$$\texttt{n*m} = \texttt{sfd(n, m)*mfm(n, m)}$$

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

For invariansen starter vi med at observere, at $\texttt{p}', \texttt{q}' \geq 1$ følger af, at vi altid subtraherer det mindste af p og q fra det største.

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

At vise, at $\texttt{sfd(p}',\texttt{q}'\texttt{)} = \texttt{sfd(p,q)}$ er det samme som at vise (vi betragter igen tilfældet q>p), at 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

$$\texttt{sfd(p,q)} \leq \texttt{sfd(p,q-p)}$$

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

$$\texttt{sfd(p,q)} \geq \texttt{sfd(p,q-p)}$$

Heraf følger gyldigheden af invarianten. For konklusionen har vi, at

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

følger direkte af $I \wedge (\texttt{p} \geq \texttt{q}) \wedge (\texttt{q} \geq \texttt{p})$. Så mangler vi kun at vise, at μ er en termineringsfunktion. Men det følger af

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

Vi konkluderer, at programmet er korrekt.

8.7 Programmering med udsagn

Invarianten for Euklids algoritme kan virke frygtindgydende, hvis man skal opdage den ved at betragte det frdige program. Sandheden er da ogs, 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 omvendte udgave af den beskrevne teknik gr begreberne udsagn, gyldighed og termineringsfunktion til elementer i en nyttig programmeringsteknik. Vi viser i dette afsnit nogle eksempler på denne teknik.

Vi betragter frst potensopløftningsprogrammet fra afsnit 2.7. Vi kan formulere opgaven som 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**-stning, så invarianten og negationen af den kontrollerende betingelse tilsammen medfrer 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 illustrere 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
```

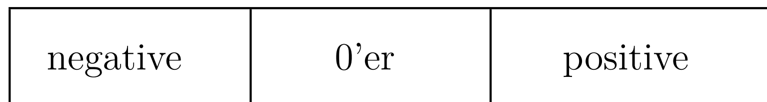
```
(+ Var i: Int
    ≪Initialiser i og x≫
    do { (V.(x) ≥ V(0..i)) ∧ (0 ≤ i ≤ |V|) }
        i ≠ | V | → ≪Opdater i og x≫
    od
+)
```

Men så er det let at se, at følgende virker

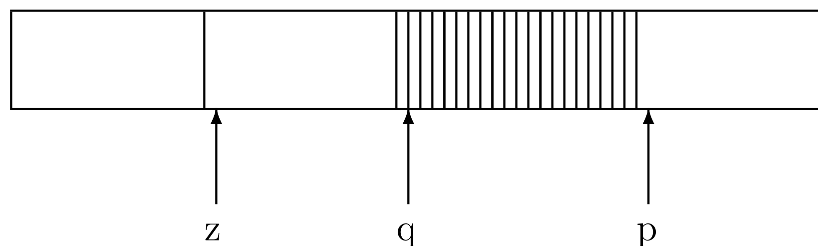
```
where ≪Initialiser i og x≫ is x, i:=0, 1

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 nul, 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 "approksimationer" 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.

8.8 Brug og begrnsninger

Begreberne i dette kapitel kan, anvendt med sund fornuft, give en uvurderlig hjlp i konstruktionen af korrekte algoritmer. Ekstreme standpunkter er dog heller ikke i denne situation ret nyttige.

Man kan oplagt ikke give et detaljeret formelt bevis for korrektheden af et program p flere millioner linjer. P den anden side vkker det dyb mistro, hvis en programmr ikke kan beskrive bare en uformel invariant for en vigtig **do**-stning.

Det er sikkert de frreste invarianter, der bliver beskrevet i formel logik. Der er til gengld rigtigt mange, der i programmrens hverdag udtrykkes ved hjlp af tegninger eller meningsfulde kommentarer. Endvidere er alle de klassiske og bermte algoritmer bevist korrekte i pinlig detalje, hvilket er baggrunden for den store tiltro, de nyder.

Teknikken i dette kapitel er meget fleksibel, da den stadig er nyttig, hvis man tager sm formelle skridt de kritiske steder og store intuitive spring i resten af programmet.

8.9 Katekismus

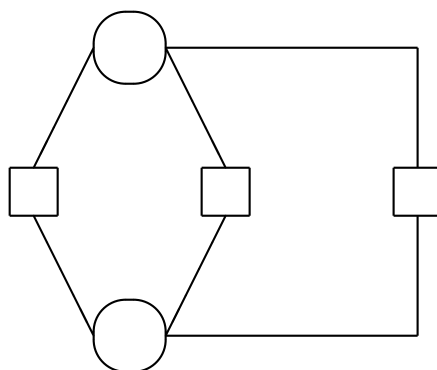
- △ Et udsagn er enten sand eller falsk i en given tilstand.
- △ Et program kan dekoreres med udsagn.
- △ Et udsagn er gyldigt i et program, hvis det er sandt i den aktuelle tilstand hver gang det nås under programmets udførelse.
- △ Et program er gyldigt, hvis alle dets udsagn er gyldige.
- △ Ved hjælp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

9 Procedurer

På nuværende tidspunkt er TRINE allerede et ganske slagkraftigt programmeringssprog, der sætter én i stand til at løse interessante algoritmiske problemer. Hvis vi standsede udviklingen af sproget her, ville det imidlertid lide af den afgørende mangel, at man kun kunne udtrykke sig ved hjælp af primitiver på et forholdsvist lavt niveau. Det bliver hurtigt nødvendigt at kunne betragte sammensatte sætninger som nye primitiver på et højere, mere problemnært niveau. Denne mulighed stilles til rådighed af *procedurer*, som er en mekanisme, der tillader konstruktion af *navngivne* og *parametriserede* programstumper. Disse kan fx placeres i et *bibliotek*, som kan *inkluderes* i et program, der således får stillet en ny samling primitiver til rådighed. Følgende afsnit viser et eksempel på dette, der involverer *grafik*.

9.1 Eksempel: Petrinet tegninger

Som det fremgår af manualen findes der et specielt grafikvindue, som TRINE-processer kan tegne i. Der findes nogle primitive tegnefaciliteter (move, moveto, turn, turnto og så videre), hvormed man kan tegne rette linjer i første kvadrant af et passende koordinatsystem. Vi ønsker at skrive en proces, der kan tegne *Petrinet* på følgende måde



Hvis vi kun kan tegne enkelte linjestykker, så virker opgaven besværlig, og hvis der er tale om et stort net, så ser det nærmest uoverkommeligt ud. Antag derimod, at vi har følgende typedefinitioner

Type Point = **Prod**(x, y: Int)
Type Figure = **Prod**(ctr: Point, r: Int)

som repræsenterer henholdsvis *punkter* (givet ved deres koordinater) og *figurer* (givet ved et centrum og en radius). Antag yderligere, at vi har følgende primitiver til rådighed

- Right(p)
- Left(p)
- Between(p,q)
- Above(p)
- Below(p)

der beregner positionen af et nyt punkt, der ligger i det angivne forhold til parametrene. Between(p,q) beregner således koordinaterne til et punkt, der ligger midt på det linjestykke, der forbinder p og q. Antag, at vi også har

- Circle(f)
- Square(f)

der tegner figuren f som henholdsvis en cirkel og et kvadrat (en figur med radius 0 tegnes som et punkt), samt

- Connect(p_1,p_2)

der tegner en forbindelseslinje mellem de to punkter p_1 og p_2. Da figurerne skal forbindes i enten deres nordlige, sydlige, østlige eller vestlige endepunkt, er følgende fire primitive *udtryk* af type Punkt også nyttige

- North(f)
- South(f)
- East(f)

- West(f)

Hver af dem beregner for en figur f det tilsvarende punkt. Nu er det simpelt at tegne nettet ovenfor på følgende måde (besværgelsen @"···" forklares senere)

```
(+ @"figures.tri"

    Var p1, p2, p3, p4, p5, p6, p7: Point
    Var f1, f2, f3, f4, f5, f6, f7: Figure
    p2 := Point(150, 150)
    p1 := Left(p2)
    p3 := Right(p2)
    p4 := Above(Between(p1, p2))
    p5 := Below(Between(p1, p2))
    p6 := Above(p3)
    p7 := Below(p3)
    f1 := Figure(p1, 15)
    f2 := Figure(p2, 15)
    f3 := Figure(p3, 15)
    f4 := Figure(p4, 20)
    f5 := Figure(p5, 20)
    f6 := Figure(p6, 0)
    f7 := Figure(p7, 0)
    Square(f1)
    Square(f2)
    Square(f3)
    Circle(f4)
    Circle(f5)
    Connect(West(f4), North(f1))
    Connect(East(f4), North(f2))
    Connect(South(f1), West(f5))
    Connect(South(f2), East(f5))
    Connect(East(f5), West(f7))
    Connect(North(f7), South(f3))
    Connect(North(f3), South(f6))
    Connect(West(f6), East(f4))
+)
```

De nævnte primitiver kan netop konstrueres som procedurer. Square vil fx få følgende udseende

```
Proc Square(k: Figure)
    jumpto(k.ctr.x-k.r, k.ctr.y-k.r)
    turnto(0)
    (+ Var a: Int
        a := 0
        do a < 4 →
            move(k.r*2)
            turn(90)
            a := a+1
        od
    +)
end Square
```

Denne procedure har et *navn* (Square), en *formel parameter* (k) af type Figure, og en *krop* (sætningen mellem **Proc** og **end**). Et *kald* af proceduren er en sætning, der har formen

```
Square(f)
```

hvor f er en *aktuel parameter*. Den aktuelle parameter skal være et udtryk, hvis type er ækvivalent med Figure. Ved udførelsen af kaldet, udføres kroppen af proceduren med værdien af den aktuelle parameter som argument; parametermekanismen vil senere blive beskrevet præcist.

Betragt nu følgende samling af relevante typer og procedurer

```
@"graphics.tri"

Type Point = Prod(x, y: Int)
Type Figure = Prod(ctr: Point, r: Int)

Proc Right(p: Point) → (Point)
    return Point(p.x+100, p.y)
end Right
```

```
Proc Left(p: Point) → (Point)
    return Point(p.x-100, p.y)
end Left

Proc Between(p, q: Point) → (Point)
    return Point((p.x+q.x)/2, (p.y+q.y)/2)
end Between

Proc Above(p: Point) → (Point)
    return Point(p.x, p.y+100)
end Above

Proc Below(p: Point) → (Point)
    return Point(p.x, p.y-100)
end Below

Proc Circle(c: Figure)
    arc(c.ctr.x, c.ctr.y, c.r, 0, 360)
end Circle

Proc Square(k: Figure)
    jumpto(k.ctr.x-k.r, k.ctr.y-k.r)
    turnto(0)
    (+ Var a: Int
        a := 0
        do a<4 →
            move(k.r*2)
            turn(90)
            a := a+1
        od
    +)
end Square

Proc North(f: Figure) → (Point)
    return Point(f.ctr.x, f.ctr.y+f.r)
end North
```

```
Proc South(f: Figure) → (Point)
    return Point(f.ctr.x, f.ctr.y-f.r)
end South

Proc East(f: Figure) → (Point)
    return Point(f.ctr.x+f.r, f.ctr.y)
end East

Proc West(f: Figure) → (Point)
    return Point(f.ctr.x-f.r, f.ctr.y)
end West

Proc Connect(p, q: Point)
    jumpto(p.x, p.y)
    moveto(q.x, q.y)
end Connect
```

Det er dem, der befinder sig på filen `figures.tri`. Notationen @"···" betyder, at indholdet af den tilsvarende fil inkluderes i programteksten på dette sted. Denne facilitet er en meget håndfast måde at stille biblioteker til rådighed på. Bemærk, at der her anvendes biblioteker i to niveauer, da `figures.tri` selv inkluderer grafikbiblioteket. På denne måde kan man systematisk opbygge mere og mere slagkraftige værktøjer.

Ved hjælp af procedurer skabte vi en samling primitiver, der gjorde programmeringen af nettet til næsten blot en verbal beskrivelse. Alternativet ville være direkte at kalde de tilgængelige faciliteter i grafikbiblioteket. Til sammenligning ville det tilsvarende program for nettet se ud som følger

```
jumpto(35,135)
turnto(0)
move(30)
turn(90)
move(30)
turn(90)
move(30)
turn(90)
move(30)
```

```
turn(90)
jumpto(135,135)
turnto(0)
move(30)
turn(90)
move(30)
turn(90)
move(30)
turn(90)
move(30)
turn(90)
⋮
```

Det skulle være indlysende, at løsningen med procedurer er lettere både at læse, skrive og modificere. Prøv fx at flytte et af kvadraterne i begge de to programmer.

9.2 Procedurer og tilstandstabeller

I dette afsnit beskrives nærmere, hvordan et procedurekald foregår. Da beskrivelsen (naturligvis) foretages ved hjælp af tilstandstabeller, er det mest bekvemt at se på en procedure, der manipulerer tal. Betragt derfor følgende procedure, der udregner *tværsummen* af et tal, det vil sige summen af cifrene; fx er tværsummen af 297 lig med 18.

```
Proc TSum[s: Int](m: Int)
    s := 0
    do m ≠ 0 →
        s := s+(m mod 10)
        m := m/10
    od
end TSum
```

Proceduren har to formelle parametre s og m, der begge er af type Int. De to parametre er imidlertid forskellige, idet s, der står i []-parenteser, er en *variabelparameter*, medens m, der står i ()-parenteser, er en *værdiparameter*. Hvis vi betragter et kald af TSum

TSum[x](u)

så skal x være et variabeludtryk og u skal være et værdiudtryk, begge af type Int. Ved kaldet udføres procedurens krop i en tilstand, hvor den formelle variabelparameter er *synonym* med den aktuelle variabelparameter, og den formelle værdiparameter angiver en *ny* variabel, hvis indhold er værdien af den aktuelle værdiparameter. Nu kan vi analysere følgende sætning, der indeholder et kald af proceduren TSum

```
          (+ Proc TSum[s: Int](m: Int)
(*2*)          s:=0
               do m ≠ 0 →
                    s:=s+(m mod 10)
                    m:=m/10
               od
(*3*)        end TSum

             Var n, t: Int

             n:=783
(*1*)        TSum[t](n)
(*4*)     +)
```

Ved udførelsen af sætningen går vi gennem følgende tilstande. Umiddelbart før kaldet er situationen

(*1*)

N	V
n: v_1	v_1: 783
t: v_2	v_2: ?

Parametermanipulationerne består i at gøre s til et synonym for t samt sætte m til at angive en ny variabel, v_3, med samme værdi som v_1.

(*2*)

N	N	V
n: v_1	s: v_2	v_1: 783
t: v_2	m: v_3	v_2: ?
		v_3: 783

Her er vi ved at udføre procedurens krop. Dette sker i en tilstand hvor der kun er adgang til de formelle parametre, og hvor omgivelsernes variabler er skjulte. Vi angiver dette med en dobbelt streg i tilstandstabellen.

(*3*)

N	N	V
n: v_1	s: v_2	v_1: 783
t: v_2	m: v_3	v_2: 18
		v_3: 0

Når procedureudførelsen er slut, sker der det samme som ved afslutningen af en indskudt sætning, det vil sige, den lokale N-søjle og dens variabler forsvinder.

(*4*)

N	V
n: v_1	v_1: 783
t: v_2	v_2: 18

I det generelle tilfælde har en procedure med navn P følgende udseende

Proc P [a: A] (b: B)
    S
end P

hvor a og b er navne, A og B er typeudtryk og S er en sætning. Procedurens *formelle parametre* står i parenteser efter dens navn; i []-parenteserne står *variabel*parametrene og i ()-parenteserne står *værdi*parametrene. I dette eksempel har vi kun to parametre, men deres antal kan være vilkårligt. Sætningen S kaldes *kroppen* af proceduren.

Et *kald* af P er en sætning af formen

P [x] (u)

hvor x er et variabeludtryk og u er et værdiudtryk. x og u kaldes de *aktuelle parametre*, og deres typer skal være ækvivalente med de formelle parametres. Når procedurekaldet udføres i tilstanden

N	V
$n_1 : v_1$	$v_1 : e_1$
$\cdots$	$\cdots$
$n_i : v_i$	$v_i : e_i$
$\cdots$	$\cdots$
$n_k : v_k$	$v_k : e_k$

bliver x og u først beregnet. Antag, at resultatet af x er variablen v_i og at resultatet af u er værdien e. Så udføres procedurekroppen S i denne nye tilstand

N	N	V
$n_1 : v_1$	$a : v_i$	$v_1 : e_1$
$\cdots$	$b : v$	$\cdots$
$n_i : v_i$		$v_i : e_i$
$\cdots$		$\cdots$
$n_k : v_k$		$v_k : e_k$
		$v : e$

Den dobbelte streg betyder, at vi ikke kan bruge de N-søjler, der ligger til venstre for denne. Inde i S kender vi således kun navnene på de formelle parametre. Den formelle variabelparameter a er blevet et synonym på den aktuelle variabelparameter. Den formelle værdiparameter b er blevet navn på en *ny* variabel v, hvis indhold er værdien af den aktuelle værdiparameter. Når kroppen er afsluttet er tilstanden ændret til

N	N	V
$n_1 : v_1$	$a : v_i$	$v_1 : e_1$
$\cdots$	$b : v$	$\cdots$
$n_i : v_i$		$v_i : e'_i$
$\cdots$		$\cdots$
$n_k : v_k$		$v_k : e_k$
		$v : e'$

Procedurekaldet afsluttes ved, at N-delen af tilstanden får sit udseende fra før kaldet, og variabler der rummede værdiparametre fjernes

N	V
$n_1 : v_1$	$v_1 : e_1$
$\cdots$	$\cdots$
$n_i : v_i$	$v_i : e'_i$
$\cdots$	$\cdots$
$n_k : v_k$	$v_k : e_k$

Som vi kan se, er nettoresultatet på tilstanden ved et procedurekald, at indholdet af de aktuelle variabelparametre kan være blevet ændret, mens der ikke er "sket" noget med værdiparametrene.

Procedurer kan også indeholde kald af andre procedurer, som i dette eksempel, hvor der udskrives den *reducerede* tværsum af et indlæst tal; den reducerede tværsum af 987 er 6.

```
(+ Proc TSum[s: Int](m: Int)
       s:=0
       do m≠0 →
           s:=s+(m mod 10)
           m:=m/10
       od
   end TSum

   Proc RSum[r: Int]
       do r>9 → TSum[r](r) od
   end RSum

   Var n: Int

   read[n]
   RSum[n]
   write(n)
+)
```

I en udførelse, hvor der er indlæst 783 vil tilstandstabellen på et tidspunkt (hvilket?) have udseendet

N	N	N	V
n: v_0	r: v_0	s: v_0	$v_0 : 0$
		m: v_1	$v_1 : 783$

I almindelighed kan den samlede effekt af procedurekald og indskudte sætninger blive, at tilstandstabellerne får et vilkårligt kompliceret udseende, som fx

N	N	N	N	N	N	N	V

9.3 Return sætningen

Et procedurekald afsluttes når kroppen er blevet udført. Nogle gange kan det være nyttigt at standse kaldet på et tidligere tidspunkt. Til det formål findes der en sætning

return

der afslutter det aktuelle procedurekald. Et eksempel på dens anvendelse er følgende procedure, der indlæser en række tal og udskriver deres løbende sum; proceduren standser, når man skriver 0

```
Proc Add
    (+ Var s: Int
       s := 0
       do true →
            (+ Var i: Int
               read [i]
               if i = 0 → return fi
               s := s+i
               write (s)
            +)
       od
    +)
end Add
```

9.4 Værdiprocedurer

Procedurer tillader os at skrive parametriserede *sætninger*. Det kan imidlertid også være nyttigt at kunne definere *parametriserede udtryk*. Til det formål indfører vi en klasse af *værdiprocedurer* med udseendet

> **Proc** P[a: A](b: B) → (C)
> S
> **end** P

Ideen er nu, at et kald af P er et *udtryk* af type C. Et udtryk skal angive en værdi; det klarer vi ved at **return**-sætninger i kroppen S efterfølges af et udtryk af type C. Dette udtryks værdi bliver således værdien af procedurekaldet. Vi må samtidig kræve, at enhver udførelse af et kald af en værdiprocedure afsluttes med udførelsen af en sådan **return**-sætning.

Et simpelt eksempel er følgende maximum-funktion, der returnerer værdien af det største af sine argumenter.

> **Proc** Max(x, y: Int) → (Int)
>   **if** x ≥ y → **return** x
> | x ≤ y → **return** y
>   **fi**
> **end** Max

Følgende lidt mere interessante eksempel returnerer den største værdi i en ikke-tom vektor

```
Proc VecMax[v: Vector] → (Int)
    (+ Var i, m: Int
        i, m := 1, 0
        do { V.(m) ≥ V(0 .. i) ∧ (0 ≤ i ≤ |V|) }
            i < | v | →
              if v.(i) > v.(m) → m := i fi
              i := i+1
        od
        return v.(m)
    +)
end VecMax
```

9.5 Variabelprocedurer

Vi er nu ledt til også at indføre *variabelprocedurer*, hvis kald er variabeludtryk. En simpel generalisering giver

```
Proc P[a: A](b: B) → [C]
    S
end P
```

hvor **return**-sætninger nu efterfølges af et variabeludtryk. Vi kan fx omskrive vores maximum-funktion til

```
Proc Max[x, y: Int] → [Int]
    if x ≥ y → return x
    | x ≤ y → return y
    fi
end Max
```

Max returnerer nu den af sine to parametre, der har den største værdi. Således vil sætningen

```
Max[a,b] := 87
```

ændre den af variablerne a og b, der har den største værdi, til at have værdien 87.

VecMax som variabelprocedure returnerer den variabel i en (ikke-tom) vektor, der har den største værdi

```
        Proc VecMax[v: Vector] → [Int]
           (+ Var i, m: Int
              i, m := 1, 0
              do i < | v | →
                 if v.(i) > v.(m) → m := i fi
                 i := i+1
              od
(*1*)         return v.(m)
           +)
        end VecMax
```

Det vil sige, at vi med sætningen

```
VecMax[V1] :=: VecMax[V2]
```

kan ombytte det største element i V_1 med det største element i V_2. Vi kan se på tilstandstabellen under en udførelse af kaldet

```
VecMax[S]
```

Antag, at vi starter i tilstanden

N	V
S: w	$w : (v_0, v_1, v_2, v_3)$
	$v_0 : 14$
	$v_1 : 8$
	$v_2 : 22$
	$v_3 : 17$

Ved mærket (*1*) står vi i tilstanden

N	N	N	V
S: w	v: w	i: v_4	$w : (v_0, v_1, v_2, v_3)$
		m: v_5	$v_0 : 14$
			$v_1 : 8$
			$v_2 : 22$
			$v_3 : 17$
			$v_4 : 4$
			$v_5 : 2$

Her kan vi se, at vi skal returnere v.(m), der er det samme som v.(2), der igen er v_2. Når kaldet af VecMax er afsluttet, så bringes vi tilbage til den oprindelige tilstand

N	V
S: w	$w : (v_0, v_1, v_2, v_3)$
	$v_0 : 14$
	$v_1 : 8$
	$v_2 : 22$
	$v_3 : 17$

hvor v_2 jo netop giver mening og udpeger det største element i S. Andre tilfælde er mere prekære. Betragt variabelproceduren

```
Proc Pip(i: Int) → [Int]
    (+ Var j: Int
        j := i
        return j
    +)
end Pip
```

Her returnerer vi en *lokal* variabel j, men det giver problemer, fordi sådanne som bekendt bliver fjernet så snart kaldet afsluttes. Dette er en penibel situation, som vi vil undgå ved at kræve, at variabler, der returneres, også skal findes blandt de variabler, der eksisterede før proceduren blev kaldt. Dette betyder, at en returneret variabel skal være en aktuel variabelparameter eller en delvariabel heraf.

Vi skal senere betragte andre (og nok så interessante) variabelprocedurer.

9.6 Eksempel: opdatering af database

Procedurer kan ofte give en mere overskuelig struktur af programmer, der arbejder med komplicerede typer. Som eksempel kan vi betragte følgende variant af programmet fra afsnit 3.8.

```
(+ Type Eksamen = Sum(afmeldt, syg: Unit, karakter: Int)
   Type Student = Prod(rskort: Int, navn: Text, res: Eksamen)
   Type Relation = List(Student)

   Proc OpKarakter [k: Int]
      if k = 0 → k := 3
       | k = 3 → k := 5
       | 5 ≤ k ≤ 10 → k := k+1
       | k = 11 → k := 13
      fi
   end OpKarakter

   Proc OpEksamen [e: Eksamen]
      if is(e, karakter) → OpKarakter [e.karakter] fi
   end OpEksamen

   Proc OpStudent [s: Student]
      OpEksamen [s.res]
   end OpStudent

   Proc OpRelation [r: Relation]
      (+ Var i: Int
         i := 0
         do i < | r | →
               OpStudent [r.(i)]
               i := i+1
         od
      +)
   end OpRelation
```

```
    Var r: Relation

    load[r]("rgang97")
    OpRelation[r]
    dump[r]("rgang97")
+)
```

Her kan man blandt andet se, at man med passende brug af procedurer kan undgå lange udvælgelser af formen r.(i).res.karakter.

9.7 Aspekter af variabelparametre

Tilstedeværelsen af variabelparametre har flere konsekvenser. Vi skal omtale *effektivitet*, *sideeffekter* og *deling*.

9.7.1 Effektivitet

Hensigtsmæssig brug af variabelparametre kan have afgørende betydning for et programs effektivitet, fordi man kan undgå eventuelle kostbare *kopieringer* af aktuelle parametre. Dette er en væsentlig begrundelse for at have variabelparametre. Betragt som eksempel følgende procedure, der returnerer en tilfældig værdi i en vektor.

```
Proc Some(x: Vector) → (Int)
    return x.(random(0, | x |))
end Some
```

En udførelse af kaldet

```
Some(S)
```

i tilstanden

N	V
S: v_0	$v_0 : (v_1, v_2, v_3, v_4, v_5, v_6)$
	$v_1 : 13$
	$v_2 : 19$
	$v_3 : 205$
	$v_4 : 4711$
	$v_5 : 6$
	$v_6 : 38$

bevirker ifølge reglerne for parametermekanismen at værdien af S udregnes, hvilket betyder at der foretages en kopiering. Selve procedurekroppen udføres derfor i tilstanden

N	N	V
S: v_0	x: v_7	$v_0 : (v_1, v_2, v_3, v_4, v_5, v_6)$
		$v_1 : 13$
		$v_2 : 19$
		$v_3 : 205$
		$v_4 : 4711$
		$v_5 : 6$
		$v_6 : 38$
		$v_7 : (v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13})$
		$v_8 : 13$
		$v_9 : 19$
		$v_{10} : 205$
		$v_{11} : 4711$
		$v_{12} : 6$
		$v_{13} : 38$

Hvis proceduren i stedet havde haft vektoren som variabelparameter, det vil sige, hvis procedurehovedet havde været

Proc Some [x: Vector] → (Int)

så ville procedurekroppen være blevet udført i tilstanden

N	N	V
S: v_0	x: v_0	$v_0 : (v_1, v_2, v_3, v_4, v_5, v_6)$
		$v_1 : 13$
		$v_2 : 19$
		$v_3 : 205$
		$v_4 : 4711$
		$v_5 : 6$
		$v_6 : 38$

Som det fremgår, bevirker værdiparametre, at der oprettes lige så mange nye variabler, som den aktuelle parametre "fylder". Dette er kostbart og kan erstattes med variabelparametre i de situationer, hvor man ikke ændrer parameteren.

9.7.2 Sideeffekter

Betragt følgende situation

```
(+ Var i: Int

   Proc P [j: Int] → (Int)
      j := j+1
      return j
   end P

   i := 0
   if P [i] ≠ P [i] → abort("Det var underligt") fi
+)
```

Betingelsen i **if**-sætningen burde altid være falsk. Men en undersøgelse af proceduren P afslører, at betingelsen faktisk altid er *sand*, fordi P har den *sideeffekt* at ændre værdien af i; situationen er bizar. Der findes talrige andre eksempler på denne uheldige opførsel. Betingelser i **if**- og **do**-sætninger bør derfor aldrig have sideeffekter. I nondeterministiske sætninger, hvor vi ikke har nogen fast rækkefølge at udregne betingelserne efter, ville vi have svært ved overhovedet at sige hvad der sker, når sætningen udføres i tilstedeværelsen af sideeffekter.

9.7.3 Deling

Variabelparametre ændrer på subtil måde de mulige indhold af tilstandstabeller. Betragt sætningen

```
(+ Proc Q[x, y: Int]
   (*2*) x := 87 (*3*) y := 13 (*4*)
   end Q

   Var i: Int
   (*1*) Q[i, i] (*5*)
+)
```

Tilstandene ved de fem mærker er henholdsvis

(*1*)

N	V
i: v_1	v_1: ?

(*2*)

N	N	V
i: v_1	x: v_1 y: v_1	v_1: ?

(*3*)

N	N	V
i: v_1	x: v_1 y: v_1	v_1: 87

(*4*)

N	N	V
i: v_1	x: v_1 y: v_1	v_1: 13

(*5*)

N	V
i: v_1	v_1: 13

Her kan vi se, at x og y er *synonymer*, det vil sige, navne på samme variabel, og at de begge er kendte på samme tid. Dette fænomen kaldes *deling*.

9.8 Katekismus

- △ En procedure er et parametriseret stykke program.
- △ Der findes variabel- og værdiparametre.
- △ En procedure kaldes med aktuelle parametre, der skal være typeækvivalente med de formelle parametre.
- △ Et kald af en procedure kan fungere som en sætning, et værdiudtryk eller et variabeludtryk.
- △ Procedurekald kan ses med specielle N-søjler i tilstandstabellen.
- △ Den formelle variabelparameter er et synonym for den aktuelle variabelparameter.
- △ Den formelle værdiparameter er en kopi af den aktuelle værdiparameter.
- △ Variabelparametre kan give anledning til sideeffekter og deling.

10 Beskyttelse, datatyper og polymorfi

Vi har defineret en typeækvivalens, der sikrer, at så mange typer som muligt er ækvivalente. Nogle gange kunne vi imidlertid have brug for at *beskytte* en type, så den ikke kan blandes sammen med andre.

10.1 Beskyttelse

Antag, at vi skal skrive en beregning, der involverer fysiske størrelser benævnte i *SI-systemets* enheder, som er *Joule* for energi, *Sekund* for tid og *Watt* for effekt. Vi kunne fx beregne energiforbruget af en 60W pære i et tidsrum med sætningen

```
(+ Var e, w, t: Int
   w := 60
   read [t]
   e := w*t
   write(e)
+)
```

hvor e er energien i Joule, w er effekten i Watt og t er tiden i sekunder. Vi kunne gøre systemets enheder tydeligere ved at skrive

```
(+ Type Joule = Int
   Type Watt = Int
   Type Sekund = Int

   Var e: Joule
   Var w: Watt
   Var t: Sekund

   w := 60
   read [t]
   e := w*t
   write(e)
+)
```

Da alle de optrædende typer er ækvivalente, er der intet der forhindrer os i at misbruge enhederne således

```
e:=w*w
```

hvilket fejlagtigt antyder, at Joule = Watt×Watt. Hvis vi ønsker at undgå dette, så kan vi pakke definitionerne ind i en såkaldt *box*, som er en definition af formen

Box B
⋮
end B

Inde i boxen står et antal definitioner; boxens funktion er at beskytte indholdet. Man kan kun referere til navne på procedurer og typer inde i en box ved at skrive boxens navn og en apostrof foran. En type T, der er defineret inde i B, opfattes udefra som typen

B'T

Tilsvarende kan en procedure

$P[x{:}X](y{:}Y)\rightarrow(Z)$

defineret i boxen, udefra opfattes som

B'$P[x{:}B$'$X](y{:}Y)\rightarrow(B$'$Z)$

hvis typerne X og Z er defineret inden i boxen, mens Y er kendt udenfor. Beskyttelse realiseres nu gennem typeækvivalensen, idet vi vedtager at typen B'T er sin egen normalform; den er med andre ord ikke ækvivalent med andre typer end sig selv.

Med denne beskyttelse kan vi nu skrive et *sikkert* SI-system, der ser ud som følger

```
Box SI
    Type Joule = Int
    Type Watt = Int
    Type Sekund = Int

    Proc TilWatt(i: Int) → (Watt)
        return i
    end TilWatt

    Proc ReadSekund[s: Sekund]
        read[s]
    end ReadSekund

    Proc WriteJoule(j: Joule)
        write(j, "J")
    end WriteJoule

    Proc Energi(w: Watt, s: Sekund) → (Joule)
        return w*s
    end Energi
end SI
```

Inde i boxen er Joule, Watt og Sekund alle ækvivalente, men udenfor er de alle forskellige, fordi boxen er ækvivalent med følgende definitioner

```
Type SI'Joule = ≪Joule≫
Type SI'Watt = ≪Watt≫
Type SI'Sekund = ≪Sekund≫

Proc SI'TilWatt(i: Int) → (SI'Watt)
    ≪ returner i≫
end SI'TilWatt

Proc SI'ReadSekund[s: SI'Sekund]
    ≪indls s≫
end SI'ReadSekund
```

```
Proc SI'WriteJoule(j: SI'Joule)
    ≪udskriv j≫
end SI'WriteJoule

Proc SI'Energi(w: SI'Watt, s: SI'Sekund) → (SI'Joule)
    ≪returner w∗s≫
end SI'Energi
```

Bemærk, at det kun er de type- og procedurenavne, der defineres inde i boxen, der udefra får boxens navn som præfiks. Forekomster af allerede kendte navne i procedurelistens parameterliste (som Int i TilWatt) betegner den udenfor eksisterende type og eksporteres uden videre.

Energiberegningen kan nu skrives som

```
(+ Var e: SI'Joule
   Var w: SI'Watt
   Var t: SI'Sekund
   w := SI'TilWatt(60)
   SI'ReadSekund[t]
   e := SI'Energi(w, t)
   SI'WriteJoule(e)
+)
```

SI'boxen kunne udvides til at omfatte alle slags enheder, konverteringer og formler og ville give en meget solid basis for fysiske beregninger. Man kunne aldrig blande enhederne sammen, og de basale formler ville altid være korrekte.

10.2 Datatyper

En *datatype* er en type med et antal tilhørende *operationer*, der kan skrives som procedurer.

De medfødte typer kan naturligvis med denne definition også opfattes som datatyper, men i almindelighed tænker man på mere specialiserede konstruktioner med forskellige skræddersyede operationer.

Et eksempel er en datatype, hvis værdier skal opfattes som mængder af heltal. De krævede operationer er

- at konstruere den tomme mængde
- at indsætte et tal i mængden
- at afgøre om et givet tal er i mængden

Vi kan bruge en box til at implementere denne datatype på følgende måde

```
Box S
    Type Set = List(Int)

    Proc Null[s: Set]
        s := Set()
    end Null

    Proc Insert[s: Set] (i: Int)
        if ¬ Member[s] (i) → s := s++Set(i) fi
    end Insert

    Proc Member[s: Set] (i: Int) → (Bool)
        (+ Var j: Int
            j := 0
            do j<| s | →
                if s.(j) = i → return true fi
                j := j+1
            od
            return false
        +)
    end Member
```

```
    Proc Print [s: Set]
        write("{")
        (+ Var j: Int
            j := 0
            do j<| s | →
                    write(s.(j))
                    if j<| s |-1 → write(",") fi
                    j := j+1
            od
        +)
        write("}")
    end Print
end S
```

Som nævnt ovenfor er denne box ækvivalent med følgende definition (der dog ikke kan skrives direkte i TRINE)

```
Type S'Set = ≪mængde af heltal≫

Proc S'Null [s: S'Set]
    ≪s := ∅≫
end S'Null

Proc S'Insert [s: S'Set] (i: Int)
    ≪s := s ∪ {i}≫
end S'Insert

Proc S'Member [s: S'Set] (i: Int) → (Bool)
    return ≪i∈s≫
end S'Member

Proc S'Print [s: S'Set]
    ≪udskriv s≫
end S'Print
```

Boxen S kan derfor bruges i følgende sætning, der indlæser en række tal, afsluttet med 0, og udskriver de af tallene, der forekommer mere end én gang.

Vi antager, at boxen S er skrevet på filen `set.tri`, så den kan importeres ved hjælp af @"`set.tri`".

```
(+ @"set.tri"

    Var tal, dub: S'Set
    S'Null[tal]
    S'Null[dub]
    (+ Var i: Int
        read[i]
        do i ≠ 0 →
            if S'Member[tal](i) → S'Insert[dub](i) fi
            S'Insert[tal](i)
            read[i]
        od
        S'Print[dub]
    +)
+)
```

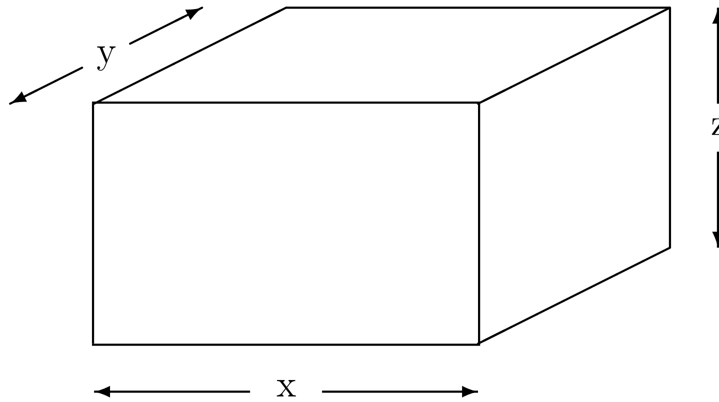
Det er klart, at der findes mange måder at realisere S'Set og de tilhørende procedurekroppe. Det er også klart, at dubletfinderen (og alle andre programmer, der benytter boxen) er *uafhængige* af den konkrete realisation, da de jo kun har kendskab til boxen på "operationsniveau". Dette betyder, at datatypen er *abstrakt*, og at implementationen er *beskyttet*, hvilket vil sige, at den kun kan manipuleres gennem boxens procedurer. Hvis vi senere fik lyst til at forbedre søgetiden i Member kunne vi uden videre ændre implementationen ved fx at forlange, at listen skal være sorteret i ikke-aftagende orden. Dette er faktisk en *invariant*, der påtrykkes repræsentationen, og på grund af beskyttelsesmekanismen kan dens gyldighed garanteres ved at sikre, at alle boxens procedurer respekterer den. For at udnytte de sorterede lister skal vi ændre Insert og Member som følger

```
Proc Insert [s: Set] (i: Int)
    (+ Var j: Int
       j := 0
       do (j < | s |) ∧ (s.(j) < i) → j := j+1 od
       if (j = | s |) ∨ (s.(j) > i) →  s := s(0 .. j)++Set(i)++ s(j .. | s |) fi
    +)
end Insert

Proc Member [s: Set] (i: Int) → (Bool)
    (+ Var j: Int
       j := 0
       do (j < | s |) ∧ (s.(j) ≤ i) →
            if s.(j) = i → return true fi
            j := j+1
       od
       return false
    +)
end Member
```

Abstrakte datatyper er et særdeles nyttigt struktureringsværktøj. De gør det muligt at stille biblioteker med de mest brugte datatyper til rådighed for programmøren. De er også meget nyttige i forbindelse med større projekter, hvor mange programmører skal samarbejde om udviklingen af et stort program. Ved på forhånd at specificere dele af programmet som abstrakte datatyper, kan flere grupper af programmører arbejde på samme tid, uden fare for at "træde hinanden over tæerne". En første gruppe kan skrive en anvendelse, der benytter en datatype, samtidigt med at en anden gruppe skriver datatypens "indmad". På grund af *abstraktionen* behøver den første gruppe ikke at vide noget om den anden gruppes beslutninger, og takket være *beskyttelsen* er det ganske enkelt umuligt for den første gruppes beslutninger at indvirke på den anden gruppes arbejde.

10.3 Eksempel: papkasser

Dette afsnit indeholder et eksempel på, hvordan invarianter på datatyper kan være nyttige. Værdimængden af vores datatype skal være *papkasser*, repræsenteret ved deres sidelængder



På papkasser kan man oplagt definere en ordning, således at $k_1 \leq k_2$, hvis k_1 kan være inde i k_2 idet vi kræver, at siderne holdes parallelle. Vi ser bort fra væggenes tykkelse, så en papkasse kan fx være inde i sig selv. Vi ønsker nu følgende operationer

```
Box Pap
    Type Kasse = Prod(x, y, z: Int)

    Proc Konst(x, y, z: Int) → (Kasse)
        ≪lav en kasse med sider x,y,z≫
    end Konst

    Proc Mindre(k1, k2: Kasse) → (Bool)
        ≪afgør om k1 ≤ k2≫
    end Mindre

    Proc Rumfang(k: Kasse) → (Int)
        ≪beregn kassens rumfang≫
    end Rumfang

    Proc Sammen(k1, k2: Kasse) → (Kasse)
        ≪beregn mindste k, så k1 ≤ k og k2 ≤ k≫
    end Sammen
end Pap
```

Når vi overvejer hvordan fx Mindre skal skrives, så ser vi hurtigt en komplikation. Vi er ikke interesseret i papkasser, der står "på skrå", men alligevel kan en papkasse vendes på 6 essentielt forskellige måder, så det ser ud

som om, vi bliver nødt til at undersøge en hel del tilfælde. Det kan vi dog undgå, hvis vi påtvinger boxen følgende invariant

- Hvis k er af type Kasse, så er k.x ≤ k.y ≤ k.z

Invarianten siger dybest set, at vi lover altid at vende papkasserne på en bestemt måde.

Introduktionen af en sådan invariant har to konsekvenser

1) Hver gang vi får en parameter af type Kasse, kan vi gå ud fra, at den opfylder invarianten.
2) Hver gang vi afleverer et resultat af type Kasse, skal vi sørge for, at den opfylder invarianten.

På grund af den indbyggede beskyttelse ved vi, at værdier af type Kasse kun kan ændres inde i boxen Pap. Det er derfor let at se, at invarianten altid vil blive overholdt, uanset hvordan boxen benyttes.

Vi kommer til følgende implementation

```
Box Pap
    Type Kasse = Prod(x, y, z: Int)

    Proc Konst(x, y, z: Int) → (Kasse)
        do x>y → x:=:y
        | y>z → y:=:z
        od
        return Kasse(x, y, z)
    end Konst

    Proc Mindre(k1, k2: Kasse) → (Bool)
        return (k1.x ≤ k2.x) ∧ (k1.y ≤ k2.y) ∧ (k1.z ≤ k2.z)
    end Mindre
```

```
    Proc Rumfang(k: Kasse) → (Int)
        return k.x*k.y*k.z
    end Rumfang

    Proc Sammen(k₁, k₂: Kasse) → (Kasse)
        (+ Proc max(i, j: Int) → (Int)
               if i ≥ j → return i
               | j ≥ i → return j
               fi
           end max

           return Kasse(max(k₁.x, k₂.x),
                        max(k₁.y, k₂.y),
                        max(k₁.z, k₂.z))
        +)
    end Sammen
end Pap
```

I procedurerne Mindre og Sammen *udnytter* vi, at invarianten gælder. I procedure Konst sorterer vi x, y og z, således at invarianten bliver *etableret*. I procedure Sammen bliver invarianten *bevaret* af beregningen.

Hvis vi ikke opretholdt invarianten, s blev vi ndt til at tage stilling til alle de 6 forskellige mder, man kan orientere en papkasse p. Fx ville operationen Mindre blive til

```
Proc Mindre(k₁, k₂: Kasse) → (Bool)
    return ((k₁.x ≤ k₂.x) ∧ (k₁.y ≤ k₂.y) ∧ (k₁.z ≤ k₂.z)) ∨
           ((k₁.x ≤ k₂.x) ∧ (k₁.z ≤ k₂.y) ∧ (k₁.y ≤ k₂.z)) ∨
           ((k₁.y ≤ k₂.x) ∧ (k₁.x ≤ k₂.y) ∧ (k₁.z ≤ k₂.z)) ∨
           ((k₁.y ≤ k₂.x) ∧ (k₁.z ≤ k₂.y) ∧ (k₁.x ≤ k₂.z)) ∨
           ((k₁.z ≤ k₂.x) ∧ (k₁.x ≤ k₂.y) ∧ (k₁.y ≤ k₂.z)) ∨
           ((k₁.z ≤ k₂.x) ∧ (k₁.y ≤ k₂.y) ∧ (k₁.x ≤ k₂.z))
end Mindre
```

og operationen Sammen kunne blive helt uoverskuelig.

10.4 Polymorfi

Når man begynder at skrive forskellige datatyper, får man hurtigt brug for at kunne lave datatype*konstruktører*, i stil med de indbyggede konstruktører **List**, **Prod** og **Sum**.

Et eksempel er en parametriseret mængde, det vil sige en datatype, som S'Set, men hvor elementtypen ikke er bundet til at være heltal. Dette kan gøres ved hjælp af en *polymorf* box (en *polybox*) på følgende måde

```
Box Set(N)
    Type Set = List(N)
    ⋮
end Set
```

I en polybox har box-navnet en *typeparameter* (N), der kan bruges som et typeudtryk inde i selve boxen. Parametermekanismen i en polybox kan nu bruges til at frembringe en almindelig box. Følgende *instantiering* resulterer i én, der er identisk med S'Set

```
Box S
    Set(Int)
end S
```

Generelt er en polymorf box af formen

```
Box P(N_1, N_2, ..., N_k)
    ⋮
end P
```

hvor N_i'erne er navne. Inde i boxen P kan N_i'erne bruges som *formelle typer*. Det er typer, hvis variabler og værdier kan udsættes for de manipulationer, der er fælles for alle Trine-typer. Disse er

- standardværdier
- definition af variabler

- tilordning `:=` og ombytning `:=:`
- sammenligning $=$ og $\neq$
- ekstern kommunikation med read, write, load og dump
- overførsel som parameter til procedurer

Da man ikke kan vide, hvilken aktuel type der erstatter en formel type ved instantiering, er dette også de eneste manipulationer, der tillades på formelle typer. En formel type er også sin egen normalform og er således ikke ækvivalent med nogen anden type.

En polybox kan *instantieres* med *aktuelle typer* på følgende måde

> **Box** B
> $P(T_1, T_2, \ldots, T_k)$
> **end** B

Her er $T_1, T_2, \ldots, T_k$ typeudtryk, og B svarer til en almindelig box, hvis indhold er det samme som P's, bortset fra, at alle forekomster af N_i systematisk er erstattet med T_i.

Indskrænkningen af manipulationer på formelle typer medfører begrænsninger for polyboxe. Man kan således ikke bruge den sorterede repræsentation i polyboxen Set, fordi formelle typer ikke tillader sammenligning ved hjælp af $\leq$ og $\geq$.

10.5 Eksempel: polymorfe sekvenser

Vi afslutter dette kapitel med et lidt større eksempel, der illustrerer flere af de begreber, som vi har set indtil nu.

Overvej følgende problem: Vi skal indlæse et ukendt antal heltal (afsluttet med 0) og anbringe dem i en liste. Umiddelbart virker problemet trivielt; vi kan blot skrive

```
(+ Type Vector = List(Int)

   Var v: Vector
   Var i: Int

   v := Vector()
   read[i]
   do i ≠ 0 →
       v := v++Vector(i)
       read[i]
   od
+)
```

Men hvad sker hvis vi fx indlæser n tal? Hver gang vi tilføjer et element til listen, udregner vi udtrykket v++Vector(i), hvilket betyder, at der bliver skabt en helt ny liste. Det kræver, at vi opretter nye delvariabler og kopierer listens indhold. Første gang er listen tom, derefter kopierer vi ét element, så to og tre, og så videre. Alt i alt foretager vi kopiering af

$$1 + 2 + 3 + \cdots + (n-2) + (n-1)$$

elementer, hvilket er proportionalt med n^2. Hvis vi vidste hvor mange tal, vi skulle modtage, kunne vi lave plads til dem på én gang, men det gør vi jo netop ikke. Vi ønsker en variant af listebegrebet, der tillader os at forlænge en sådan liste, uden at omkostningerne løber urimeligt i vejret. Vi kan benytte følgende strategi: Hver gang vores liste løber fuld, kan vi *fordoble* listens længde, så vi har lidt plads at bruge, inden vi skal kopiere igen. Hvor mange kopieringer foretager vi på denne måde? Første gang kopierer vi en liste med ét element, næste gang en med 2 elementer, så med 4, og så videre. Alt i alt foretager vi

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2}$$

kopieringer, hvilket er proportionalt med n. Vi ønsker altså en slags særligt effektiv liste, som vi skal kalde en *sekvens*.

Da behovet for denne form for effektivitet dukker op i mange andre sammenhænge, skal vi konstruere sekvensen så den tillader flere operationer end der umiddelbart er behov for i dette eksempel. Mere præcist skal vi

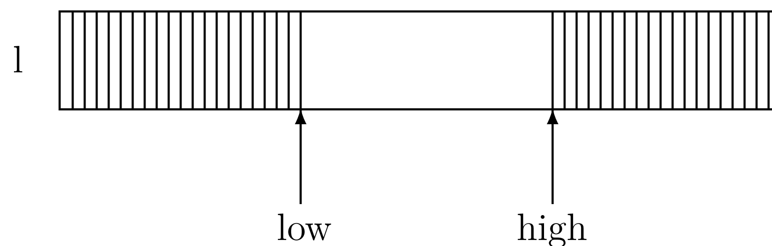
udstyre den, så vi udover initialisering kan tilføje (push), fjerne (pop) og aflæse (top) det yderste element i både venstre (L) og højre (R) ende af listen. Hertil kommer, at vi kan "få fat" i et vilkårligt af sekvensens elementer ved hjælp af en variabelprocedure (Sel).

Hvis elementtypen er heltal, kan en sådan sekvens beskrives på følgende måde

```
Type S'Seq = ≪liste af heltal≫

Proc S'Init [s: S'Seq]
    ≪s := ()≫
end S'Init

Proc S'Con [s: S'Seq] (c: List (Int))
    ≪s := c≫
end S'Con

Proc S'Len [s: S'Seq] → (Int)
    return ≪|s|≫
end S'Len

Proc S'Lpush [s: S'Seq] (i: Int)
    ≪s := (i)++s≫
end S'Lpush

Proc S'Lpop [s: S'Seq]
    ≪s := s(1 .. |s|)≫
end S'Lpop

Proc S'Ltop [s: S'Seq] → [Int]
    return ≪s.(0)≫
end S'Ltop

Proc S'Rpush [s: S'Seq] (i: Int)
    ≪s := s++(i)≫
end S'Rpush
```

```
Proc S'Rpop[s:S'Seq]
    ≪s:=s(0..|s|−1)≫
end Rpop

Proc S'Rtop[s:S'Seq] → [Int]
    return ≪s.(|s|−1)≫
end Rtop

Proc S'Sel[s:S'Seq](i:Int) → [Int]
    return ≪s.(i)≫
end Sel

Proc S'Print[s:S'Seq]
    ≪udskriv s≫
end Print
```

Da sådanne sekvenser er nyttige for andet end heltal, skal vi implementere dem i en polybox. Den konkrete repræsentation er i form af en liste på følgende måde



det vil sige, hvor indholdet af den abstrakte sekvens befinder sig mellem pegepindene low og high i den konkrete liste l, og hvor det stribede område er ubenyttet. Mere præcist er indholdet af den abstrakte sekvens identisk med dellisten l(low..high).

```
Box Sequence(T)
    Type Tlist = List(T)
    Type Seq = Prod(low, high: Int, l: Tlist)

    Proc Init[s: Seq]
        s := Seq(0, 0, Tlist())
    end Init

    Proc Con[s: Seq](c: List(T))
        s := Seq(0, | c |, c)
    end Con

    Proc Len[s: Seq] → (Int)
        return s.high−s.low
    end Len

    Proc Lpush[s: Seq](t: T)
        if | s.l | = 0 → s := Seq(0, 1, Tlist(t))
        & s.low>0 →
             s.low := s.low−1
             s.l.(s.low) :=: t
        & true →
             (+ Var n: Int
                 n := | s.l |
                 s.l := Tlist(?−T | n)++s.l
                 s.l.(n−1) :=: t
                 s.low, s.high := n−1, s.high+n
             +)
        fi
    end Lpush

    Proc Lpop[s: Seq]
        if Len[s] >0 → s.low := s.low+1 fi
    end Lpop
```

```
Proc Ltop[s: Seq] → [T]
    if Len[s] = 0 → abort("Ltop on empty sequence") fi
    return s.l.(s.low)
end Ltop

Proc Rpush[s: Seq] (t: T)
    if | s.l | = 0 → s := Seq(0, 1, Tlist(t))
    & s.high< | s.l |  →
         s.l.(s.high) :=: t
         s.high := s.high+1
    & true →
         (+ Var n: Int
             n := | s.l |
             s.l := s.l++Tlist(?-T | n)
             s.l.(n) :=: t
             s.high := s.high+1
         +)
    fi
end Rpush

Proc Rpop[s: Seq]
    if Len[s]>0 → s.high := s.high-1 fi
end Rpop

Proc Rtop[s: Seq] → [T]
    if Len[s] = 0 → abort("Rtop on empty sequence") fi
    return s.l.(s.high-1)
end Rtop

Proc Sel[s: Seq] (i: Int) → [T]
    if 0 ≤ i<Len[s] → return s.l.(s.low+i)
    & true → abort("Seq index out of range")
    fi
end Sel
```

```
    Proc Print [s: Seq]
        write(s.l(s.low .. s.high))
    end Print
end Sequence
```

Sel er et eksempel på en nyttig variabelprocedure: den giver os adgang til sekvensens delvariabler (og ikke kun til deres værdier) på samme måde som vi kender fra lister, summer og produkter.

Nu kan vi i stedet skrive vores program som

```
(+ Box S
       Sequence(Int)
   end S

   Var s: S'Seq
   Var i: Int

   S'Init [s]
   read [i]
   do i ≠ 0 →
        S'Rpush [s] (i)
        read [i]
   od
+)
```

Denne udgave kan ved prøvekørsler ses at være langt hurtigere. Gevinsten vil vokse kvadratisk med antallet af elementer. Bemærk, at vi ikke har ændret vores indlæsningsalgoritme, der ikke fejlede noget. Datatypen Sequence er heller ikke væsensforskellig fra datatypen **List**. Det er udelukkende effektiviteten af den underliggende implementation, der er ændret.

10.6 Eksempel: polymorfe kataloger

Som et sidste eksempel vil vi skrive en polymorf datatype, hvis værdier er *kataloger*, det vil sige endelige partielle funktioner, med følgende operationer

```
Box Catalog(S, T)
    Type Cat = List(Pair)
    Type Pair = Prod(x: S, y: T)

    Proc Init[c: Cat]
        c := Cat()
    end Init

    Proc Search[c: Cat](s: S) → (Int)
        (+ Var i: Int
            i := 0
            do i<| c | →
                if c.(i).x = s → return i fi
                i := i+1
            od
            return ?-Int
        +)
    end Search

    Proc Defined[c: Cat](s: S) → (Bool)
        return Search[c](s) ≠ ?-Int
    end Defined

    Proc Dom[c: Cat] → (List(S))
        (+ Var d: List(S)
            d := List(?-S | | c |)
            (+ Var i: Int
                i := 0
                do i<| c | →
                    d.(i) := c.(i).x
                    i := i+1
                od
            +)
            return d
        +)
    end Dom
```

```
        Proc Lookup[c: Cat] (s: S) → (T)
            (+ Var i: Int
                i := Search[c] (s)
                if i = ?-Int → abort
                & true → return c.(i).y
                fi
            +)
        end Lookup

        Proc Update[c: Cat] (s: S, t: T)
            (+ Var i: Int
                i := Search[c] (s)
                if i = ?-Int → c := c++Cat(Pair(s, t))
                & true → c.(i).y := t
                fi
            +)
        end Update

        Proc Delete[c: Cat] (s: S)
            (+ Var i: Int
                i := Search[c] (s)
                if i ≠ ?-Int → c := c(0 .. i)++c(i+1 .. | c |) fi
            +)
        end Delete
    end Catalog
```

Hvis man fx skal bruge en telefonbog, der er en partiel funktion fra navne til numre, kan den opnås som

```
    Box Phone
        Catalog(Text, Int)
    end Phone
```

Med samme lethed kan et globalt register, der er en partiel funktion fra lande til telefonbøger, opnås som

```
Box Global
    Catalog(Text, Phone'Cat)
end Global
```

På overdrevets rand kan man få en partiel funktion fra planeter til globale registre som

```
Box InterPlanetary
    Catalog(Text, Global'Cat)
end InterPlanetary
```

10.7 Kanoniske tekstformater

Boxtyper har, på grund af beskyttelsen, ingen konstantudtryk udover standardværdier, og de har derfor i princippet heller ingen kanoniske tekstformater. De har værdier, men disse er *anonyme* og bruges kun som indhold af variabler af den pågældende type.

Man bør derfor ikke indlæse/udskrive værdier til/fra variabler af boxtyper ved hjælp af read/write. Der skal ske en konvertering ved hjælp af en procedure som Print i boxene S og Sequence. Værdier af boxtyper kan dog stadig load'es og dump'es.

10.8 Katekismus

- △ En samling typer og procedurer kan beskyttes i en box.
- △ Typer defineret inde i boxen får udenfor påklistret boxens navn.
- △ Sådanne boxtyper er kun ækvivalente med sig selv.
- △ Datatyper kan implementeres med boxe.
- △ Implementationen af en datatype i en box kan ændres uden konsekvenser for brugeren.
- △ Beskyttelse i en box kan bruges til at garantere en invariant for implementationen af en datatype.

△ En datatype kan laves polymorf med en polybox.

△ En polybox har formelle typeparametre.

△ En formel type tillader kun de operationer, der er fælles for alle typer.

△ Til gengæld kan en formel type i en polybox instantieres med en vilkårlig aktuel type.

11 Definitioner og navne

Gennem tilstandstabellerne har vi givet en nøjagtig beskrivelse af hvilke variabelnavne, der er kendte på et givet tidspunkt af programudførelsen og hvilke variabler de refererer til. Vi har imidlertid også navngivne procedurer, typer og boxe, og her mangler vi en tilsvarende præcis beskrivelse.

11.1 Statiske omgivelser

Den første ting vi kan bemærke er, at hvor variabelnavnes betydning ændrer sig *dynamisk* under udførelsen, så er betydningerne af de øvrige navne *statiske*; de kan læses direkte fra programteksten.

Til enhver lokalitet i programteksten kan vi knytte dens tilstandstabel og *statiske omgivelse*, der er en mængde af procedure-, type- og boxdefinitioner. Betydningen af et navn afgøres ved at konsultere den statiske omgivelse.

Intuitivt omfatter den statiske omgivelse for et punkt i et program de navne, der kan ses ved at læse "opad, nedad og udad" i programmets hierarkiske struktur. Hvis det samme navne optræder flere gange, så vinder det nærmeste.

11.2 Eksempel

Vi betragter nu et eksempelprogram, hvis eneste effekt er at udskrive et C (med meget besvær).

De forskellige områder af programteksten, der har fælles statisk omgivelse, er blevet indrammede. Hvert af disse områder er desuden identificeret med et bogstav. Et områdes udstrækning omfatter *ikke* de indre delområder.

```
Process Citrus
  (+ Box Grape(T)                                              A
       Type Appelsin = List(T)                             B
       Type Citron = T

       Proc Mandarin[a: List(T)] → (Citron)
         (+ Type Citron = Appelsin         C
            Var c: Citron
            c:=a
            return c.(0)
         +)
       end Mandarin
     end Grape

     Proc Citron(t: Text)
       (+ Type Grape = Text                D
          Var g: Grape
          g:=t
          write(Lime'Mandarin[g])
       +)
     end Citron

     Box Lime
       Grape(Char)
     end Lime

     Citron("Clementin")
  +)
end Citrus
```

Vi kan nu vise de statiske omgivelser i de angivne områder. Indholdet af procedurer og boxe er kun skitseret. I område A er den statiske omgivelse

> **Box** Grape · · · **end** Grape
> **Proc** Citron · · · **end** Citron
> **Box** Lime · · · **end** Lime

I område B er den statiske omgivelse

Proc Mandarin $\cdots$ **end** Mandarin
Type Citron = T
Type Appelsin = **List**(T)
Type T
Box Grape $\cdots$ **end** Grape
Box Lime $\cdots$ **end** Lime

I område C er den statiske omgivelse

Type Citron = Appelsin
Proc Mandarin $\cdots$ **end** Mandarin
Type Appelsin = **List**(T)
Type T
Box Grape $\cdots$ **end** Grape
Box Lime $\cdots$ **end** Lime

I område D er den statiske omgivelse

Type Grape = Text
Proc Citron $\cdots$ **end** Citron
Box Lime $\cdots$ **end** Lime

11.3 Formel definition

Til brug for den generelle beskrivelse af statiske omgivelser, skal vi introducere en *overskygningsoperator*. Hvis E_1 og E_2 er statiske omgivelser, så er $E_1 \oslash E_2$ en statisk omgivelse, der er ligesom $E_1 \cup E_2$ bortset fra, at definitioner i E_1, med navne der også optræder i E_2, bliver fjernet; E_2 overskygger således E_1.

De statiske omgivelser kan nu bestemmes efter følgende regler:

- Den statiske omgivelse for en enkelt proces er tom.

- Hvis den indskudte sætning

```
(+ Def₁
   Def₂
   ...
   Defₖ

   S
+)
```

har statisk omgivelse G, så har S og alle Def_i'erne fælles statisk omgivelse, nemlig

$$G \oslash \{\text{Def}_i | \text{Def}_i \text{ er } \textbf{Proc}, \textbf{Type}, \text{eller } \textbf{Box}\}$$

- Hvis proceduren

Proc P `[···] (···)`
  S
end P

har statisk omgivelse G, så har S også statisk omgivelse G.

- Hvis (poly)boxen

Box $B(T_1, T_2, \ldots, T_n)$
  Def_1
  Def_2
  $\cdots$
  Def_k
end B

har statisk omgivelse G, så har hvert Def_i statisk omgivelse

$$G \oslash (\{T_1, \ldots, T_n\} \cup \{\text{Def}_j | \text{Def}_j \text{ er } \textbf{Proc}, \textbf{Type}, \text{eller } \textbf{Box}\})$$

- Hvis sekvensen

$S_1 S_2 \cdots S_k$

har statisk omgivelse G, så har hvert S_i også statisk omgivelse G.

• Hvis en **if**- eller **do**-sætning med betingede sætninger $S_1, S_2, \ldots, S_k$ har statisk omgivelse G, så har hvert S_i også statisk omgivelse G.

11.4 Navneregler

Vi kan nu præcisere de regler, der gælder for anvendelse af navne i et TRINE program.

§1 De eneste navne, der må refereres i en bestemt lokalitet i et program, er navne i lokalitetens statiske omgivelse eller i den dynamiske omgivelse (det vil sige, navne i tilstandstabellen).

§2 Intet navn må optræde i to forskellige **var**-, **type**-, **proc**-, eller **box**-definitioner i samme omgivelse for nogen lokalitet.

11.5 Rekursion

I generelt sprogbrug kaldes et begreb *rekursivt*, hvis dets definition refererer til begrebet selv.

Det er en vigtig observation, at reglerne for statiske omgivelser tillader rekursivitet i vores definitioner. Et eksempel er sætningen

```
(+ Def_1
   Def_2
   ...
   Def_k

   S
+)
```

Hvis Def_i er en **Proc**-, **Type**- eller **Box**-definition, så indeholder dens statiske omgivelse Def_i selv. Definitionen kan således referere til sig selv.

Det kan vi se i eksemplet i første afsnit, hvor den statiske omgivelse i område D inde i proceduren Citron omfattede proceduren Citron selv.

I de næste kapitler skal vi se, at de rekursive definitioner, som hermed er introduceret, er af central betydning i TRINE.

11.6 Katekismus

- △ Til hvert sted i et program kan man knytte en statisk omgivelse.
- △ En statisk omgivelse indeholder definitioner på alle de navne, der er kendte der.
- △ Rækkefølgen på definitioner er ikke signifikant.
- △ Generelt skal man læse indefra og udad.
- △ Disse regler tillader vilkårlige rekursive definitioner.

12 Rekursive procedurer

En *rekursiv* procedure indeholder *kald af sig selv*. Dette begreb er ikke så fremmed, som det måske synes ved første øjekast. Et velkendt eksempel på en rekursivt defineret udregningsmetode udgøres af de sædvanlige regler for differentiation af rationale funktioner (det vil sige funktioner som er summer, produkter eller kvotienter mellem polynomier). Reglerne er som følger; $f(x)$ og $g(x)$ er rationale funktioner og c er en reel konstant

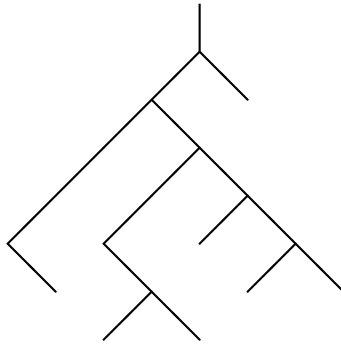
1) $c' = 0$
2) $(x^n)' = nx^{n-1}$
3) $(cf(x))' = cf'(x)$
4) $(f(x) + g(x))' = f'(x) + g'(x)$
5) $(f(x)g(x))' = f(x)g'(x) + f'(x)g(x)$
6) $(\frac{f(x)}{g(x)})' = \frac{g(x)f'(x) - f(x)g'(x)}{g^2(x)}$

Reglerne 1) og 2) angiver direkte resultatet af at differentiere simple udtryk, mens reglerne 3)-6) angiver, at et sammensat udtryk differentieres ved, at det opbrydes i deludtryk, som hver især differentieres ved (rekursiv) anvendelse af reglerne.

I det følgende vises et antal eksempler på rekursive programmer, hvoraf det første faktisk er prototypen på, hvordan en rekursiv definition på naturlig måde leder frem til en rekursiv metode til generering af den pågældende struktur.

12.1 Binære træer

Et (formelt) *træ* er en idealiseret version af et "rigtigt træ"; det er en struktur, som består af en rod, hvorfra der udgår forgreninger, som så igen forgrener sig, og så videre. I den bedste datalogiske tradition skal vi tegne træer på en sådan måde, at de vokser nedad



Ovenstående er et eksempel på et *binært* træ, der defineres som følger

- Et *binært træ* er enten tomt, eller det består af en gren, der efterfølges af to binære træer.

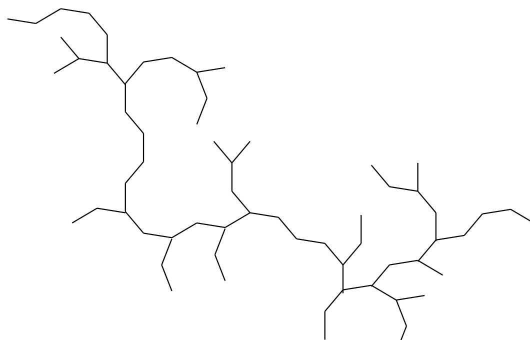
På basis af denne definition kan vi tegne et binært træ på følgende måde

```
(+ Proc Tree
      if true → skip
       | true →
           ≪tegn en gren≫
           Tree
           ≪returner til grenens endepunkt≫
           Tree
      fi
   end Tree

   Tree
+)
```

Dette er et (nondeterministisk) program, der er i nøje overensstemmelse med definitionen. Det er også klart, at programmet er rekursivt, idet proceduren Tree kaldes inde i sin egen krop. Programmet er ikke særligt konkret med hensyn til hvordan det ønskede træ skal se ud, og hvilken længde og retning dets grene skal have. Hvis vi vedtager, at en gren har længde 25 og at undertræerne forgrener sig i vinkler på ±40 grader, kan vi konkretisere programmet på følgende måde

```
(+ @"graphics.tri"

   Proc Tree(v: Int)
      if true → skip
      | true →
          turnto(v)
          move(25)
          (+ Var x, y: Int
             position[x, y]
             Tree(v-40)
             jumpto(x, y)
             Tree(v+40)
          +)
      fi
   end Tree

   jumpto(300, 0)
   Tree(90)
+)
```

Bemærk, at proceduren Tree stadig er nondeterministisk, og at den altså kan tegne mange forskellige træer. Følgende er resultatet af en udførelse af programmet



Det er ikke noget tilfælde, at træer falder så naturligt for rekursive procedurer. I en vis forstand resulterer enhver udførelse af en rekursiv procedure i en træstruktur, hvilket yderligere kan illustreres ved hjælp af følgende program til beregning af de såkaldte *Fibonacci tal*. Disse er opkaldt efter matematikeren *Leonardo Fibonacci af Pisa*, som i 1202 udgav værket *Liber Abaci* (bogen om beregninger), hvori han blandt andet stillede spørgsmålet

> Hvor mange kaninpar kan der blive af et nyfødt kaninpar på et år, hvis hvert par føder et nyt par hver måned og et nyfødt par føder første gang efter to måneder?

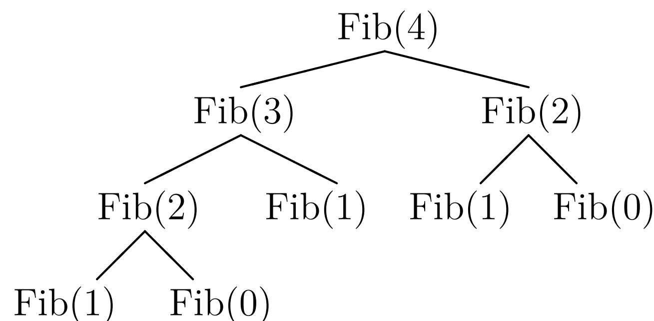
Hvis det antages, at kaniner aldrig dør, er det ikke svært at se, at der spørges efter det 12. element i følgen $\{F_n\}_{n \geq 0}$ defineret ved

$$F_0 = F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2},\ n > 1$$

Denne definition er rekursiv, da formlen for det n'te Fibonacci-tal henviser til mængden af (mindre) Fibonacci-tal. Vi kan skrive en tilsvarende simpel værdiprocedure, der udregner det n'te element

```
Proc Fib(n: Int) → (Int)
    if n = 0 → return 1
     | n = 1 → return 1
     | n > 1 → return Fib(n-1)+Fib(n-2)
    fi
end Fib
```

Et kald af Fib(5) vil give anledning til følgende træstruktur af procedurekald



12.2 Rekursive procedurer og tilstandstabeller

I dette afsnit beskrives nærmere, hvordan tilstandsdiagrammer udvikler sig under udførelsen af rekursive procedurer. Eksemplerne tjener kun til at give en tilstrækkeligt simpel beskrivelse af denne mekanisme.

Betragt følgende procedure

```
        Proc R(i: Int)
(*1*)      if i > 0 →
               write(i)
               R(i-1)
           fi
(*2*)   end R
```

Den indeholder et kald af sig selv og er dermed rekursiv. Antag nu, at vi udfører kaldet

R(3)

Lad os se hvilke tilstande vi kommer igennem

(*1*)

N	V
i: v_1	v_1: 3

(*1*)

N	N	V
i: v_1	i: v_2	v_1: 3 v_2: 2

(*1*)

N	N	N	V
i: v_1	i: v_2	i: v_3	v_1: 3 v_2: 2 v_3: 1

(*1*)

N	N	N	N	V
i: v_1	i: v_2	i: v_3	i: v_4	v_1: 3 v_2: 2 v_3: 1 v_4: 0

(*2*)

N	N	N	N	V
i: v_1	i: v_2	i: v_3	i: v_4	v_1: 3 v_2: 2 v_3: 1 v_4: 0

(*2*)

N	N	N	V
i: v_1	i: v_2	i: v_3	v_1: 3
			v_2: 2
			v_3: 1

(*2*)

N	N	V
i: v_1	i: v_2	v_1: 3
		v_2: 2

(*2*)

N	V
i: v_1	v_1: 3

Her ser vi tydeligt den rekursive mekanisme. Hvert rekursivt kald skaber en ny N-søjle og en ny variabel, en såkaldt ny *instans* af den rekursive procedure. Hver gang et kald afsluttes, fjernes den aktuelle instans. Antallet af forskellige instanser af den samme rekursive procedure kaldes for *rekursionsdybden*, der i ovenstående eksempel maksimalt er 4.

Rekursive kald er ikke forskellige fra andre kald. Vi ville få præcist den samme sekvens af tilstande, hvis vi i stedet udførte sætningen

```
(+ Proc R0 (i: Int)
       skip
   end R0

   Proc R1 (i: Int)
       if i > 0 →
             write(i)
             R0(i-1)
       fi
   end R1
```

```
Proc R₂(i: Int)
    if i > 0 →
        write(i)
        R₁(i-1)
    fi
end R₂

Proc R₃(i: Int)
    if i > 0 →
        write(i)
        R₂(i-1)
    fi
end R₃

R₃(3)
+)
```

der *ikke* involverer rekursion. Det specielle ved en *rekursiv* procedure fremfor denne tilsvarende samling af ikke-rekursive procedurer er, foruden de indlysende notationsmæssige fordele, at den tillader en *ubegrænset* rekursionsdybde. Vi skal ikke på forhånd vide, hvor mange instanser, der er brug for.

Rekursion kan også være *gensidig*, hvor fx to procedurer kalder hinanden

```
Proc Op[n: Int]
    if n > 1 →
        n := 3*n+1
        Ned[n]
    fi
end Op

Proc Ned[n: Int]
    do n mod 2 = 0 → n := n/2 od
    Op[n]
end Ned
```

Denne situation kan analyseres med samme lethed som ovenfor. Procedu-

rekald fungerer altid på den samme måde, uanset hvordan den pågældende procedure er defineret.

Ligesom **do**-sætninger kan rekursive procedurer give sætninger, hvis udførelse aldrig terminerer. Det simpleste eksempel er

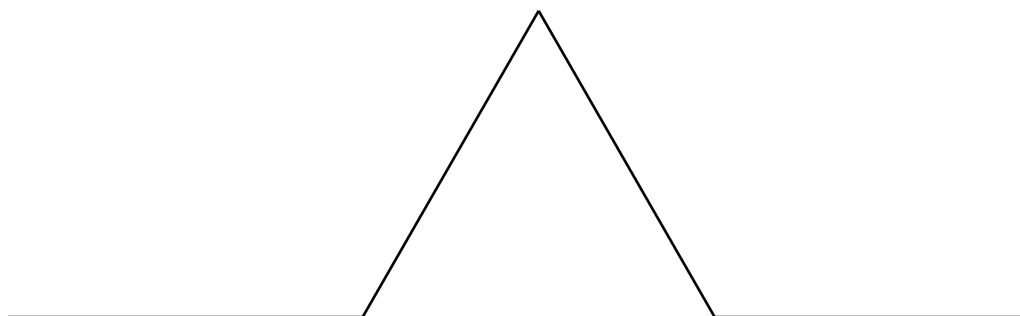
Proc P
    P
end P

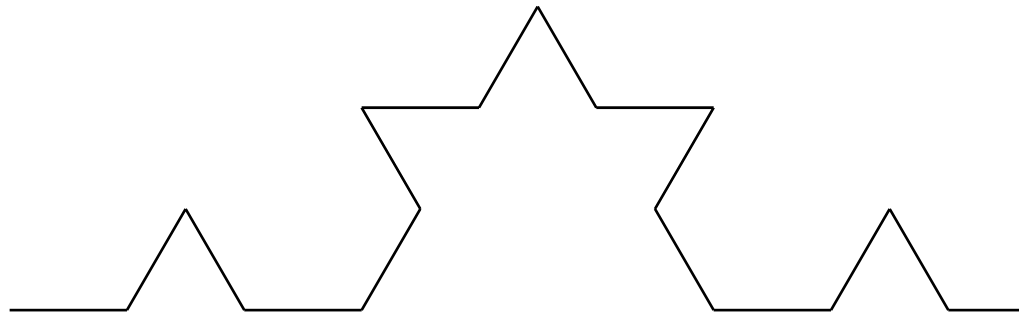
Et kald af P fører til et nyt kald af P, og så videre. Hvis enhver instans af en rekursiv procedure udfører endnu et rekursivt kald, så vil udførelsen naturligvis aldrig stoppe. Det er derfor vigtigt, at enhver rekursiv procedure indeholder en *rekursionsbetingelse*, der på et tidspunkt tillader proceduren at undgå flere rekursive kald.

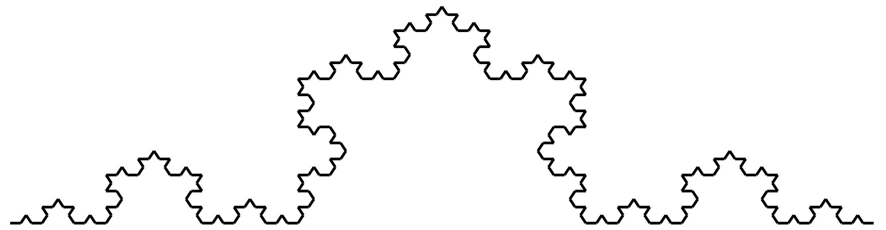
At rekursive procedurer har mange interessante anvendelser, illustreres af eksemplerne i de næste afsnit.

12.3 Eksempel: fraktaler

En *fraktal* er en kurve, som intetsteds er glat nok til at kunne approximeres med linjestykker. Et simpelt eksempel på en fraktal er den såkaldte *Koch-kurve*, der opstår som grænseværdien (for n gående mod uendelig) af *Koch-linjen* af orden n. En Koch-linje af orden n opnås ud fra en Koch-linje af orden $n-1$ ved at give alle linjestykker i den et trekantet hak. De følgende er Koch-linjer af orden 1, 2 og 4





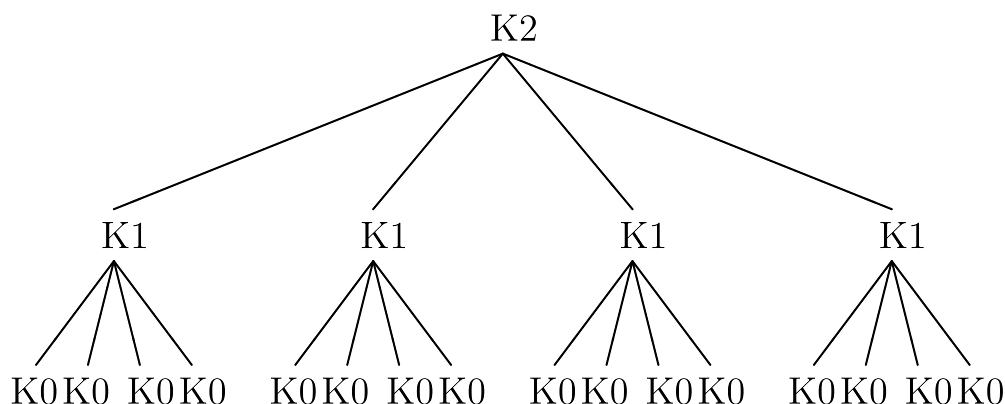


Vi kan skrive en meget simpel rekursiv procedure, der tegner en Koch-linje af en given orden.

```
Proc KochLine(n, dir, length: Int)
    if n = 0 →
        turnto(dir)
        move(length)
    | n>0 →
        KochLine(n-1, dir, length/3)
        KochLine(n-1, dir+60, length/3)
        KochLine(n-1, dir-60, length/3)
        KochLine(n-1, dir, length/3)
    fi
end KochLine
```

Bemærk, at vi foretager 4 rekursive kald, og at vi her har et eksempel på en rekursiv procedure, hvor en ikke-rekursiv formulering forekommer at være endog meget besværlig.

Udførelsestræet for et kald af Kochline(2) kan skitseres som følger, hvor vi skriver Ki for et kald af KochLine(i,. . . ,. . .)



12.4 Eksempel: syntaksanalyse

Det næste eksempel omhandler *syntaksanalyse*. Vi ser på simple udtryk med encifrede talkonstanter, plus, minus og parenteser. Eksempler på sådanne udtryk er

```
2+2
7-(2+7)-8
(2+3+4+5)
```

Vores problem er at afgøre, hvorvidt en given tekststreng er et sådant udtryk. De følgende er fx ikke udtryk

```
2+
7(2+7)-8
(2+3+4+(
```

Først angiver vi følgende *grammatik*, som er en præcis (rekursiv) beskrivelse af hvad der er udtryk

Expr	::=	**Term** |
		Term + **Expr** |
		Term − **Expr**
Term	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
		(**Expr**)

En grammatik består af et antal *regler*, som fx

Expr ::= **Term** + **Expr**

hvor ::= adskiller reglens venstre og højre side. Venstresiden består af en *nonterminal* (**Expr**) og højresiden består af to nonterminaler (**Term** og **Expr**) adskilt af en *terminal* (+). Et antal regler med samme venstreside kan angives ved at separere højresiderne med |. I ovenstående grammatik er der således 3 regler med venstreside **Expr** og 11 regler med venstreside **Term**.

En nonterminal kan *frembringe* en følge af terminaler ved *genskrivning*, det vil sige ved gentagen erstatning af en regels venstreside med dens højreside. Nonterminalen **Expr** frembringer således følgen 2 + 2 på følgende måde

Expr
Term + **Expr**
2 + **Expr**
2 + **Term**
2 + 2

Vi er interesserede i de følger af terminaler, der på denne måde frembringes af nonterminalen **Expr**. Mere præcist er vi interesserede i at skrive et program, der kan *genkende* disse tegnfølger, det vil sige, som givet en vilkårlig tegnfølge afgør om den frembringes af **Expr**.

Da grammatikken består af to gensidigt rekursive definitioner, **Expr** og **Term**, skriver vi to gensidigt rekursive procedurer med samme navne. Hver procedure har to parametre, én der indeholder (resten af) den tegnfølge der søges genkendt (og som læses fra venstre mod højre) og én der fortæller om den del af tegnfølgen, der hidtil er analyseret, er i overensstemmelse med definitionen; hvis ikke, så indeholder parameteren r en passende fejlmeddelelse.

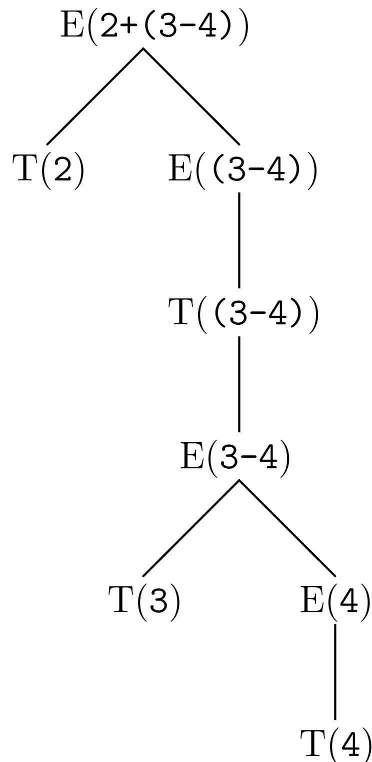
Med de polymorfe sekvenser fra afsnit 10.5, fås følgende program

```
(+ @"sequence.tri"

   Box T
      Sequence(Char)
   end T

   Type Response = Sum(ok: Unit, error: Text)

   Proc Expr[s: T'Seq, r: Response]
      Term[s, r]
      if is(r, ok) ∧ (T'Len[s] >0) →
         if T'Ltop[s] = '+' →
            T'Lpop[s]
            (+ Var q: Response
               Expr[s, q]
               r := q
            +)
         | T'Ltop[s] = '-' →
            T'Lpop[s]
            (+ Var q: Response
               Expr[s, q]
               r := q
            +)
         fi
      fi
   end Expr

   Proc Term[s: T'Seq, r: Response]
      if (T'Len[s] >0) →
         if '0' ≤ T'Ltop[s] ≤ '9' →
            r := Response(ok: #)
            T'Lpop[s]
         | T'Ltop[s] = '(' →
            T'Lpop[s]
            (+ Var q: Response
               Expr[s, q]
```

```
                if is(q, ok) →
                    if (T'Len[s]>0) ∧ (T'Ltop[s] = ')') →
                        T'Lpop[s]
                        r:=q
                    & true → r:=Response(error: ") expected")
                    fi
                & true → r:=q
                fi
              +)
        & true → r:=Response(error: "Digit or ( expected")
        fi
    & true → r:=Response(error: "Incomplete expression")
    fi
  end Term

  Var t: Text

  write(eol, "Write an expression: ")
  read[t]
  (+ Var s: T'Seq
     Var r: Response
     T'Con[s](t)
     Expr[s, r]
     if is(r, error) → write(r.error)
     & (T'Len[s]>0) →
          write("Unexpected text: ")
          T'Print[s]
     & is(r, ok) → write("Approved")
     fi
  +)
+)
```

Bemærk, hvorledes de to procedurer på meget direkte vis følger definitionen af syntaksen og reagerer (fornuftigt) på fejlsituationer.

Et kald af Expr på teksten `2+(3-4)` giver anledning til følgende udførelsestræ, hvor vi skriver E(...) for kald af Expr og T(...) for kald af Term



Det er klart, at en stor del af implementationerne af fx Rasmus og Trine må omhandle syntaksanalyse. I samme stil kunne de her viste procedurer udvides til at beregne udtrykkets værdi eller til at tillade mere avancerede udtryk. Her løber man imidlertid hurtigt ind i problemer, hvis løsning ligger uden for rammerne af nærværende noter. Et simpelt eksempel på et sådant problem er, at grammatikker ikke altid har den egenskab, at man direkte ved hjælp af næste symbol (+,−, 7, og så videre) kan se hvilken procedure, der skal kaldes.

12.5 Eksempel: rekursive planter

Dette sidste eksempel viser, at tegninger af rekursive procedurers udførelsestræer i visse tilfælde kan have en æstetisk værdi. Vi kan generalisere eksemplet fra afsnit 12.1 og give de tegnede træer et mere naturtro præg. Til det formål udstyrer vi proceduren med følgende parametre

r: træets retning
v: vinkel mellem grene
g: største grenlængde

h: højden af træet
f: største antal forgreninger
gs: sandsynligheden for en gren (i procent)
bs: sandsynligheden for et blad (i procent)

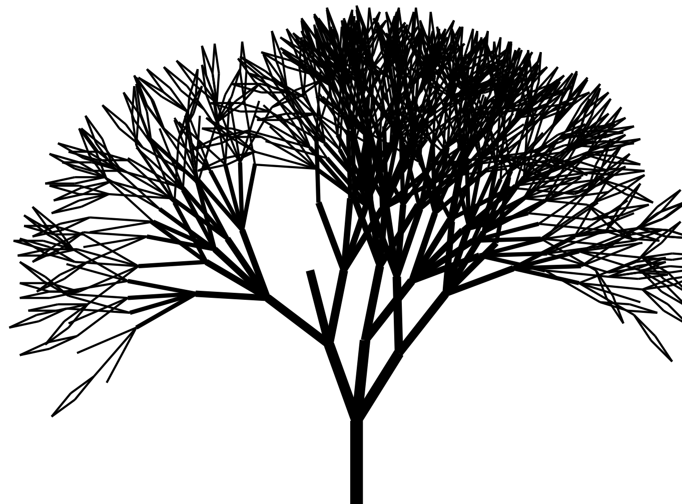
Proceduren får følgende udseende

```
Proc Tree(r, g, h, f, v, gs, bs: Int)
    if h = 0 → Leaf(v, r, g/2, bs)
    | h>0 →
        turnto(r)
        setwidth(h)
        move(g)
        (+ Var x, y, i: Int
            position[x, y]
            r:=r-(v*(f-1))/2
            i:=f
            do i>0 →
                if random(0, 100)<gs →
                    jumpto(x, y)
                    Tree(r, g-2, h-1, f, v, gs, bs)
                fi
                r, i:=r+v, i-1
            od
        +)
    fi
end Tree
```

hvor Leaf er nedenstående procedure

```
Proc Leaf(v, r, l, bs: Int)
    if random(0, 100)<bs →
        turnto(r-v/2)
        setwidth(1)
        move(l)
        turn(v)
        move(l)
        turn(180-v)
        move(l)
        turn(v)
        move(l)
    fi
end Leaf
```

Et kald af Tree(90,40,6,6,13,55,30) gav dette resultat



Ved at variere parametrene kan man efterligne et stort antal planter.

12.6 Katekismus

- △ En rekursiv procedure indeholder kald af sig selv.
- △ Hvis en rekursiv procedure skal være fornuftig, så skal den have mulighed for at undgå rekursive kald.
- △ Den må desuden kun foretage rekursive kald med mindre parametre.

- △ Til ethvert kald af en rekursiv procedure hører der et unikt udførelsestræ.
- △ Man kan også have gensidigt rekursive procedurer.
- △ Et kald af en rekursiv procedure ser ikke specielt ud i tilstandstabellen.

13 Rekursive typer

Rekursion kan også bruges på typer. Det giver sig fx udslag i definitioner af formen

$$\textbf{Type}\ T = \cdots T \cdots$$

hvor typen optræder på højresiden af sin egen definition.

13.1 Træer som rekursive typer

Med rekursion kan man definere typer, hvis værdier er *træer*. Betragt fx følgende definition af et binært træ med heltal i knuderne.

- Et *heltalstræ* er **enten** tomt, **eller** det er en *knude* med **både** et heltal **og** to heltalstræer.

Da et tomt træ bekvemt kan repræsenteres af typens standardværdi, er det oplagt at forsøge sig med følgende

$$\textbf{Type}\ \text{Tree} = \ \textbf{Prod}(\text{val: Int, left, right: Tree})$$

Man kunne få mistanke om, at værdierne af denne type ville blive uendelige, da en Tree-værdi igen indeholder to Tree-værdier. Det er imidlertid ikke tilfældet, da vi jo altid har standardværdien ?-Tree. Således vil udtrykket

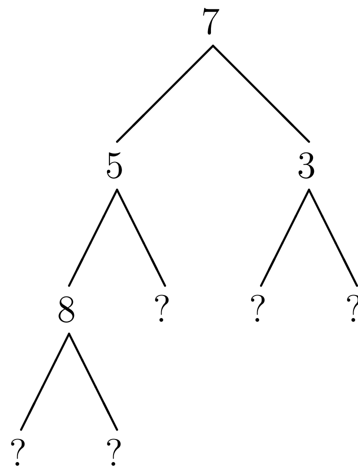
$$\text{Tree(12, ?-Tree ?-Tree)}$$

angive et binært træ med kun én knude, der indeholder heltallet 12. Hvis vi skulle angive en uendelig værdi, så måtte vi skrive et uendeligt udtryk, hvilket ikke er muligt.

Direkte manipulationer med træer bliver hurtigt uoverskuelige. I almindelighed vil man derfor benytte rekursive procedurer til både at opbygge træer og til at arbejde med dem. Næste afsnit viser større eksempler på dette, men den følgende procedure vil fx returnere summen af knuderne i et binært træ

```
Proc Add[t: Tree] → (Int)
    if  ¬is(t) → return 0
     &  true → return t.val+Add[t.left]+Add[t.right]
    fi
end Add
```

Virkemåden af en sådan procedure kan igen beskrives ved at betragte tilstandstabeller. Som eksempel ser vi på et træ, hvis “abstrakte værdi” er



Dette træ angives af følgende TRINE-udtryk U (indrykningen er foretaget for at tydeliggøre træstrukturen)

```
U : Tree(7,
        Tree(5,
            Tree(8,
                ?-Tree,
                ?-Tree
              ),
            ?-Tree
          ),
        Tree(3,
            ?-Tree,
            ?-Tree
          )
      )
```

Hvis tree nu er en variabel af type Tree, så vil sætningen

```
tree:=U
```

resultere i følgende tilstandstabel

N	V
tree: v_0	v_0: (v_1, v_2, v_3)
	v_1: 7
	v_2: (v_4, v_5, v_6)
	v_3: (v_7, v_8, v_9)
	v_4: 5
	v_5: (v_{10}, v_{11}, v_{12})
	v_6: ?
	v_7: 3
	v_8: ?
	v_9: ?
	v_{10}: 8
	v_{11}: ?
	v_{12}: ?

Vi betragter nu et kald af proceduren Add i denne tilstand. Umiddelbart efter kaldet

Add [tree]

er situationen

N	N	V
tree: v_0	t: v_0	v_0: (v_1, v_2, v_3)
		$\vdots$
		v_{12}: ?

Her er der blot sket det, at v_0 (som er resultatet af det aktuelle variabeludtryk tree) er blevet bundet til den formelle parameter t. Under udførelsen af procedurekroppen sker der nu et antal rekursive kald. Umiddelbart efter det første

Add [t.left]

har tilstandstabellen følgende udseende

N	N	N	V
tree: v_0	t: v_0	t: v_2	v_0: (v_1, v_2, v_3)
			v_1: 7
			v_2: (v_4, v_5, v_6)
			$\vdots$
			v_{12}: ?

idet resultatet af det aktuelle variabeludtryk t.left nu er variablen v_2, som er bundet til t.

I det næste rekursive kald er det v_5, der bindes til t, og derefter er det v_{11}. Nu bliver rekursionsdybden ikke større, da v_{11} indeholder det tomme træ ?-Tree. Læseren opfordres til omhyggeligt at gennemgå tilstandstabellens "bevægelser" i resten af denne beregning.

En "rekursiv variabel" som tree har naturligvis en værdi i lighed med alle andre variabler. Denne udregnes på samme måde som for variabeludtryk i almindelighed, det vil sige som summer, produkter eller lister af værdierne af delvariablerne. Værdien af tree i ovenstående tilstandstabel er

$$(7, (5, (8, ?, ?), ?), (3, ?, ?))$$

13.2 To eksempler på træer som rekursive typer

I dette afsnit behandler vi to af eksemplerne fra forrige kapitel igen, denne gang ved hjælp af rekursive typer.

Det første er processen Tree, der tegner et (nondeterministisk) binært træ. En alternativ måde at tegne et sådant træ er først eksplicit at konstruere træet (som værdi af en variabel af en passende rekursiv træ-type) og dernæst at tegne den figur, der svarer til træet. I følgende program konstruerer proceduren

```
Proc MakeTree[t: Tree]
```

et træ t af følgende type Tree

```
Type Tree = Prod(left, right: Tree)
```

Her er knudernes indhold uden betydning – det eneste interessante er selve træets facon. Programmet ser ud som følger

```
(+ @"graphics.tri"

   Type Tree = Prod(left, right: Tree)

   Proc MakeTree[t: Tree]
      if true → t := ?-Tree
       | true →
           (+ Var left, right: Tree
              MakeTree[left]
              MakeTree[right]
              t := Tree(left, right)
           +)
      fi
   end MakeTree
```

```
    Proc DrawTree[t: Tree] (v: Int)
        if is(t) →
            turnto(v)
            move(25)
            (+ Var x, y: Int
                position[x, y]
                DrawTree[t.left] (v-40)
                jumpto(x, y)
                DrawTree[t.right] (v+40)
            +)
        fi
    end DrawTree

    Var tree: Tree

    Maketree[tree]
    jumpto(300, 250)
    Drawtree[tree] (270)
+)
```

Her bliver træets facon fastlagt på forhånd, inden det bliver tegnet. Drawtree er derfor en deterministisk procedure. Læseren opfordres til at overbevise sig om, at dette program gør præcist det samme som programmet i afsnit 12.1.

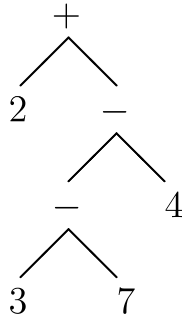
Det andet eksempel, som vi også har set tidligere, er syntaksanalysen af de simple udtryk med encifrede konstanter, plus, minus og parenteser. Vi kan definere en rekursiv type, der kan repræsentere sådanne udtryk

```
Type Expr = Sum(num: Int, plus, minus: Arg)
Type Arg = Prod(left, right: Expr)
```

Bemærk, at vi ikke eksplicit har parenteser i typen. Deres betydning modsvares af selve træets struktur. Udtrykket

$$2+(3-7)-4$$

repræsenteres således af træet



Ved hjælp af typen Expr kan vi nu modificere vores syntaksanalyse, så den i stedet for blot at analysere et udtryk, faktisk opbygger det tilsvarende Expr-træ. Ideen er igen at skrive et program, der afspejler definitionens struktur. De to nye procedurer (BuildExpr og BuildTerm) skal nu *konstruere* de træer, der hører til de analyserede tegnfølger. Dette gøres ved at modificere resultattypen Response så ok-varianten i stedet for typen Unit bliver af typen Expr og bruges til at indeholde det konstruerede træ.

```
(+ @"sequence.tri"

   Box T
      Sequence(Char)
   end T

   Type Expr = Sum(num: Int, plus, minus: Arg)
   Type Arg = Prod(left, right: Expr)
   Type Response = Sum(ok: Expr, error: Text)

   Proc BuildExpr[s: T'Seq, r: Response]
      BuildTerm[s, r]
      if is(r, ok) ∧ (T'Len[s] > 0) →
          if T'Ltop[s] = '+' →
              T'Lpop[s]
              (+ Var q: Response
                 BuildExpr[s, q]
                 if is(q, ok) → r.ok := Expr(plus: Arg(r.ok, q.ok))
                 & true → r := q
                 fi
              +)
```

```
        | T'Ltop[s] = '-' →
            T'Lpop[s]
            (+ Var q: Response
               BuildExpr[s, q]
               if is(q, ok) → r.ok := Expr(minus: Arg(r.ok, q.ok))
               & true → r := q
               fi
            +)
        fi
    fi
end BuildExpr

Proc BuildTerm[s: T'Seq, r: Response]
    if (T'Len[s]>0) →
        if '0' ≤ T'Ltop[s] ≤ '9' →
            r := Response(ok: Expr(num: ci(T'Ltop[s])-ci('0')))
            T'Lpop[s]
        | T'Ltop[s] = '(' →
            T'Lpop[s]
            (+ Var q: Response
               BuildExpr[s, q]
               if is(q, ok) →
                   if (T'Len[s]>0) ∧ (T'Ltop[s] = ')') →
                       T'Lpop[s]
                       r := q
                   & true → r := Response(error: ") expected")
                   fi
               & true → r := q
               fi
            +)
        & true → r := Response(error: "Digit or ( expected")
        fi
    & true → r := Response(error: "Incomplete expression")
    fi
end BuildTerm

Var t: Text
```

```
    write(eol, "Write an expression: ")
    read[t]
    (+ Var s: T'Seq
       Var r: Response
       T'Con[s](t)
       BuildExpr[s, r]
       if is(r, error) → write(r.error)
       & (T'Len[s]>0) →
           write("Unexpected text: ")
           T'Print[s]
       & is(r, ok) → write("Approved: ", r.ok)
       fi
    +)
+)
```

Vores ok-respons indeholder nu en trærepræsentation af udtrykket. Bemærk, at vi indlæser udtryk som *højreassociative*. Et udtryk som

$$1+2-3+4-5+6-7+8-9$$

læses som

$$1+(2-(3+(4-(5+(6-(7+(8-9)))))))$$

Til slut kan vi skrive en procedure, der *udregner* værdien af et Expr-træ

```
Proc Eval[e: Expr] → (Int)
    if is(e, num) → return e.num
    | is(e, plus) → return Eval[e.plus.left]+
                           Eval[e.plus.right]
    | is(e, minus) → return Eval[e.minus.left]-
                            Eval[e.minus.right]
    fi
end Eval
```

Proceduren Eval minder meget om proceduren Add, der beregnede summen af knuderne i et Int-træ.

Alle rekursive typer definerer en klasse af træer; deres værdier og variabler manipuleres gennem rekursive procedurer. Ovenstående eksempler kan tjene som skabeloner for generelle anvendelser af rekursive typer.

13.3 Rekursive værdier og variabler

Hvilke værdi- og variabeludtryk har en generel rekursiv type? Vi har faktisk allerede svaret på dette spørgsmål. Vi har nemlig ikke indført nye typekonstruktører, så de sædvanlige regler for **Sum**, **Prod** og **List** er stadig gældende. Lad os betragte typen af udtrykstræer igen

Type Udtryk = **Sum**(tal: Int, plus, minus: Arg)
Type Arg = **Prod**(venstre, højre: Udtryk)

Den har blandt andre følgende konstante værdiudtryk

Udtryk(tal: 17)
Udtryk(plus: Arg(Udtryk(tal: 17), Udtryk(tal: 18))

Hvis x er en Udtryk-variabel, så er følgende et udvalg af typens variabeludtryk

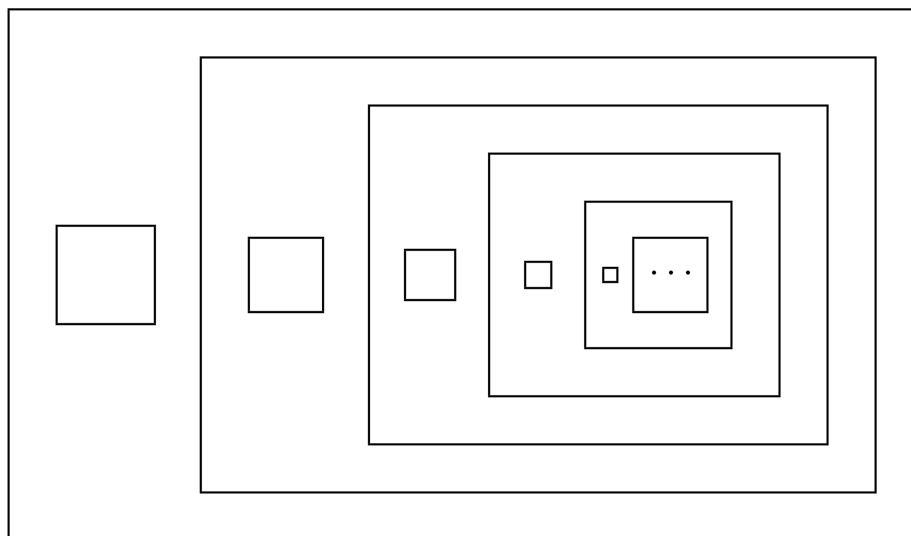
t
x.tal
x.plus.venstre.tal
x.plus.venstre.minus
x.plus.venstre.minus.højre.
x.plus.venstre.minus.højre.tal
x.plus.venstre.minus.højre.minus.højre.tal

Her er der ligeledes et klart system. Et variabeludtryk angiver en vej i variablens træstruktur af delvariabler. En variabel af en rekursiv type har således *potentielt* uendeligt mange delvariabler, selv om kun endeligt mange af dem vil kunne angive eksisterende variabler i en given tilstand; antallet afhænger af, hvor store træer variablerne indeholder. Det svarer fuldstændigt til rekursive procedurer, der tillader ubegrænset rekursionsdybde, selv om enhver tilstand kun kan indeholde endeligt mange instanser.

Det er nu også klart, hvorfor fx en produktvariabels delvariabler ikke alle bliver skabt, når den erklæres. Hvis det var tilfældet, så ville en variabel af typen

Type Uha = **Prod**(x: Int, y: Uha)

skulle beskrives ved følgende uendelige kassogram



som vi ikke kan klare.

Der er således egentligt ikke indført noget nyt med hensyn til værdi- og variabeludtryk. Situationen omkring typernes værdimængder er dog nu lidt mere avanceret. Vi ved hvordan **List**, **Sum** og **Prod** skal fortolkes. Hvis vi direkte nedskriver oversættelsen fra typeudtryk til mængdeoperationer, så ender vi med et system af *ligninger* på mængder. Betragt fx følgende typer

Type A = B
Type B = A
Type C = **Prod**(x, y: C)
Type D = **List**(D)

De giver anledning til nedenstående ligninger på værdimængder

Val(A) = Val(B)

$$\begin{aligned}\text{Val(B)} &= \text{Val(A)}\\ \text{Val(C)} &= \text{Val(C)}\times\text{Val(C)} \cup \{?\}\\ \text{Val(D)} &= \text{Val(D)}^* \cup \{?\}\end{aligned}$$

Dette ligningssystem kan løses på flere måder, men da alle typer skal have en standardværdi, så er vi kun interesserede i løsninger der indeholder ?. Af årsager, som vi ikke skal komme nærmere ind på her, er vi desuden interesserede i den *mindste* løsning, med hensyn til $\subseteq$. Det vides fra matematikken, at en entydig sådan løsning altid findes. For ovenstående ligningssystem får vi følgende løsninger

$$\begin{aligned}\text{Val(A)} &= \{?\}\\ \text{Val(B)} &= \{?\}\\ \text{Val(C)} &= \{?,\ (?,?),\ (?,(?,?)),\ ((?,?),(?,?)),\ \ldots\}\\ \text{Val(D)} &= \{?,\ (),\ (?),\ (()),\ (?,()),\ (?,?),\ ((),(),?),\ \ldots\}\end{aligned}$$

Metoden med at opstille ligninger, der jo også gælder for ikke-rekursive typer, er helt generel. Ligningen for de binære træer bliver

$$\text{Val(Tree)} = \text{Val(Int)}\times\text{Val(Tree)}\times\text{Val(Tree)} \cup \{?\}$$

og man kan overbevise sig om, at mængden af binære Int-træer svarer til løsningen til denne ligning.

Vi kan nu også forstå hvad typen Unit er. Den er ganske enkelt defineret som

Type Unit = Unit

hvis tilhørende ligning

$$\text{Val(Unit)} = \text{Val(Unit)}$$

har løsningen Val(Unit) = {?} (husk, at # blot er et konstant udtryk for standardværdien ?).

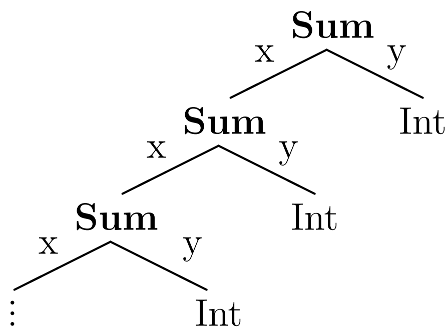
13.4 Rekursiv typeækvivalens

Slutteligt mangler vi at bestemme, hvilke rekursive typeudtryk der er ækvivalente. Vi har faktisk en ganske enkel måde at gøre dette på, idet vi blot generaliserer teknikken med normalformer fra kapitel 4.

Betragt som eksempel typen

Type E = **Sum**(x: E, y: Int)

Dens normalform nf(E) bliver træet



Da typen er rekursiv, bliver normalformen et uendeligt træ.

I visse tilfælde giver denne teknik dog et problem, som kan illustreres af definitionerne

Type F = G
Type G = F
Type H = H

Her kan man blive ved med at gå fra navne til højresider, uden at komme nogen vegne. Sådanne typer tildeler vi den specielle normalform Ω, som nu altså også må optræde som blad i (andre) normalformer. Betragt eksemplerne

Type I = **Sum**(x: I, y: J)
Type J = K
Type K = J
Type L = **List**(**Sum**(z: L, w: Int))

Deres normalformer er

Sum
x y
nf(I) = **Sum** Ω
x y
⋮ Ω

nf(J) = nf(K) = Ω

List
|
nf(L) = **Sum**
z w
List Int
|
⋮

Det skulle nu være klart, hvad en rekursiv types normalform er, og at den konstrueres ved hjælp af udfoldningsmetoden som beskrevet ovenfor. Ω'erne kommer ind som indholdet af de blade, hvor et navn ikke kan erstattes af andet end andre navne (for så møder man naturligvis en gentagelse). Typer med normalform Ω svarer altid til Unit-typen.

Således har enhver type en unik normalform. Det kan også indses, at typer med samme normalform har samme værdi- og variabeludtryk, og dermed samme værdimængder.

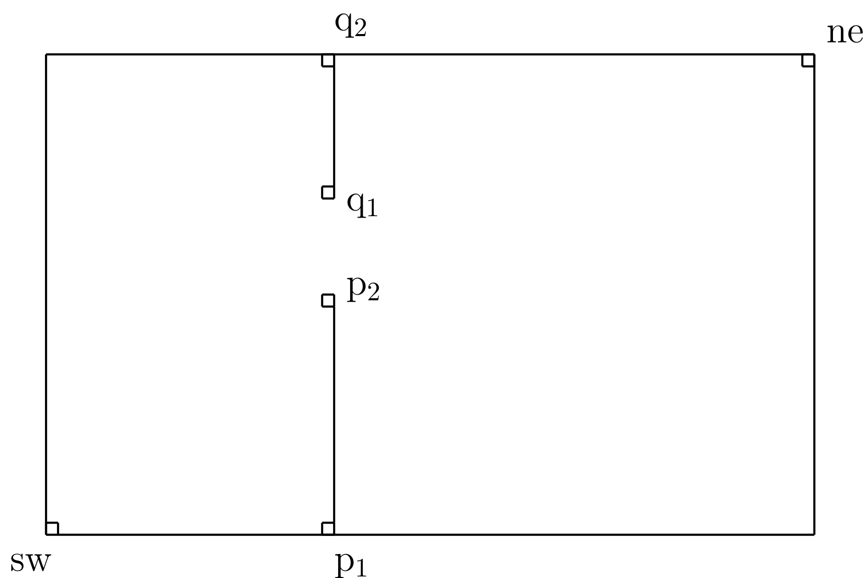
13.5 Eksempel: rekursive labyrinter

Vi fortsætter med et større eksempel, der illustrerer de fleste nye begreber.

En *labyrint* i et rektangel er et arrangement af skillevægge. Vi vil kun beskæftige os med labyrinter, der kan tegnes på et stykke kvadreret papir, hvor væggene følger markeringerne. En labyrint er *simpel*, hvis det er muligt at finde vej mellem vilkårlige to steder i rektanglet, og der ikke er løse

skillevægge. I en simpel labyrint kan man altid finde vej ud ved at gå med den ene hånd langs væggen.

Vi er naturligvis interesserede i at definere labyrinter *rekursivt*. Det kan vi gøre på følgende måde



En labyrint i et rektangel består af en skillevæg, med en døråbning, og to mindre labyrinter, en på hver sin side af væggen. Væggen kan være enten lodret eller vandret og kan placeres tilfældigt. Rektanglet angives af punkterne *sw* og *ne*, der angiver henholdsvis det sydvestlige og det nordøstlige hjørne. Punkterne p_1, p_2, q_1 og q_2 angiver placeringen af skillevæggen og døren. En labyrint kan også være så smal, at der slet ikke er plads til en skillevæg.

Vi kan definere en sådan labyrint som den rekursive type Laby

Type Point = **Prod**(x, y: Int)
Type Wall = **Prod**(p_1, p_2, q_1, q_2: Point)
Type Laby = **Prod**(sw, ne: Point, w: Wall, L_1, L_2: Laby)

I lighed med træeksemplet fra afsnit 13.2 kan vi nu generere en labyrint i et vilkårligt rektangel ved hjælp af følgende rekursive procedure MakeLaby

```
Proc MakeLaby(sw, ne: Point) → (Laby)
    if (ne.x-sw.x ≤ 1) ∨ (ne.y-sw.y ≤ 1) → return ?-Laby fi
    (+ Var w: Wall
        w := MakeWall(sw, ne)
        return Laby(sw, ne, w, MakeLaby(sw, w.q2),
                                MakeLaby(w.p1, ne))
    +)
end MakeLaby

Proc MakeWall(sw, ne: Point) → (Wall)
    (+ Var p1, p2, q1, q2: Point
        if true →
                p1 := Point(random(sw.x+1, ne.x), sw.y)
                p2 := Point(p1.x, random(sw.y, ne.y))
                q1 := Point(p1.x, p2.y+1)
                q2 := Point(p1.x, ne.y)
         | true →
                p1 := Point(sw.x, random(sw.y+1, ne.y))
                p2 := Point(random(sw.x, ne.x), p1.y)
                q1 := Point(p2.x+1, p1.y)
                q2 := Point(ne.x, p1.y)
        fi
        return Wall(p1, p2, q1, q2)
    +)
end MakeWall
```

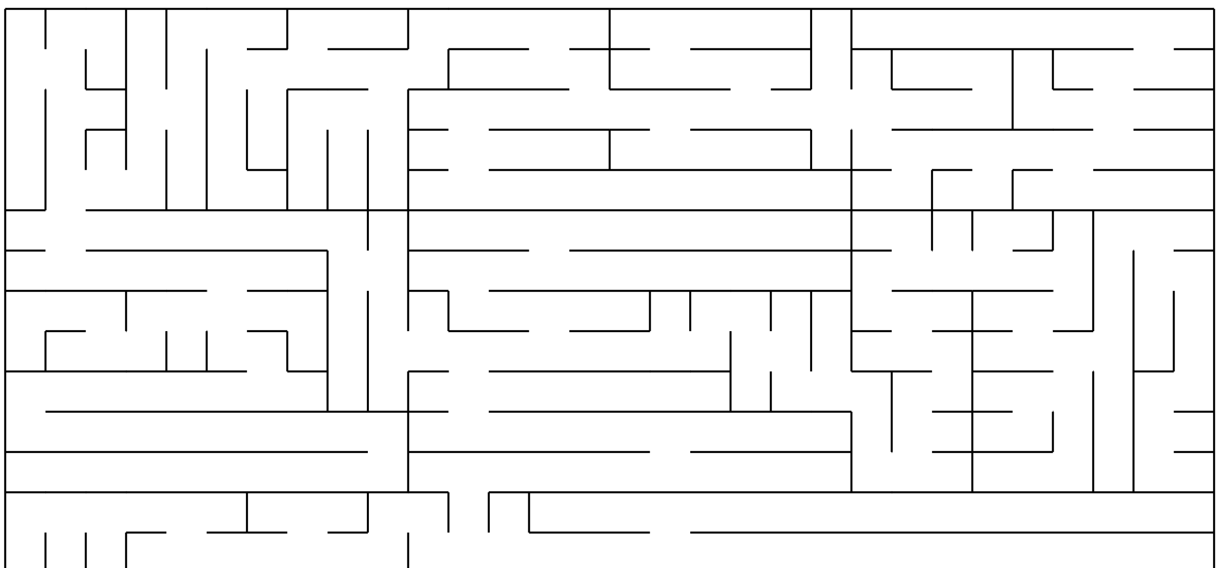
Vi vælger nondeterministisk, om skillevæggen skal være lodret eller vandret; begge valg giver jo en korrekt labyrint. Placeringen af væggen og døren vælges tilfældigt. Et lille udsnit af en værdi af type Laby ser ud som følger

((1,1),(31,15),((1,4),(9,4),(10,4),(31,4)),((1,1),(31,4),((20,1),
(20,1),(20,2),(20,4)),((1,1),(20,4),((1,3),(19,3),(20,3),(20,3)),
((1,1),(20,3),((1,2),(1,2),(2,2),(20,2)),?,?),?),((20,1),(31,4),
((20,3),(24,3),(25,3),(31,3)),((20,1),(31,3),((27,1),(27,2),(27,3),
(27,3)),((20,1),(27,3),((20,2),(24,2),(25,2),(27,2)),?,?),((27,1),
(31,3),((27,2),(27,2),(28,2),(31,2)),?,?)),?)),((1,4),(31,15),((1,6),
(17,6),(18,6),(31,6)),((1,4),(31,6),((1,5),(8,5),(9,5),(31,5)),?,?),

Vi vil meget hellere se en tegning af labyrinten på skærmen; det klares meget let af den rekursive procedure DrawLaby

```
Proc DrawLaby [L: Laby]
    if is(L) →
        DrawWall [L.w]
        DrawLaby [L.L₁]
        DrawLaby [L.L₂]
    fi
end DrawLaby

Proc DrawWall [w: Wall]
    jumptoPoint(w.p₁)
    movetoPoint(w.p₂)
    jumptoPoint(w.q₁)
    movetoPoint(w.q₂)
end DrawWall
```

Procedurerne movetoPoint og jumptoPoint skalerer labyrintkoordinater til skærmkoordinater. En typisk labyrint ser ud som følger



Rammen omkring labyrinten er tegnet med proceduren Frame

```
Proc Frame(sw, ne: Point)
    jumpto(20*sw.x, 20*sw.y)
    moveto(20*ne.x, 20*sw.y)
    moveto(20*ne.x, 20*ne.y)
    moveto(20*sw.x, 20*ne.y)
    moveto(20*sw.x, 20*sw.y)
end Frame
```

Endeligt kunne vi ønske at finde vej mellem to punkter i labyrinten. Med vores repræsentation er det hensigtsmæssigt med pile at markere de døråbninger, som vi skal igennem. Det er meget ligetil at gøre rekursivt. Lad L_1 og L_2 være de to dellabyrinter, lad d være punktet hvor døren er, og antag, at vi skal fra punktet s til punktet t. Så er der fire tilfælde

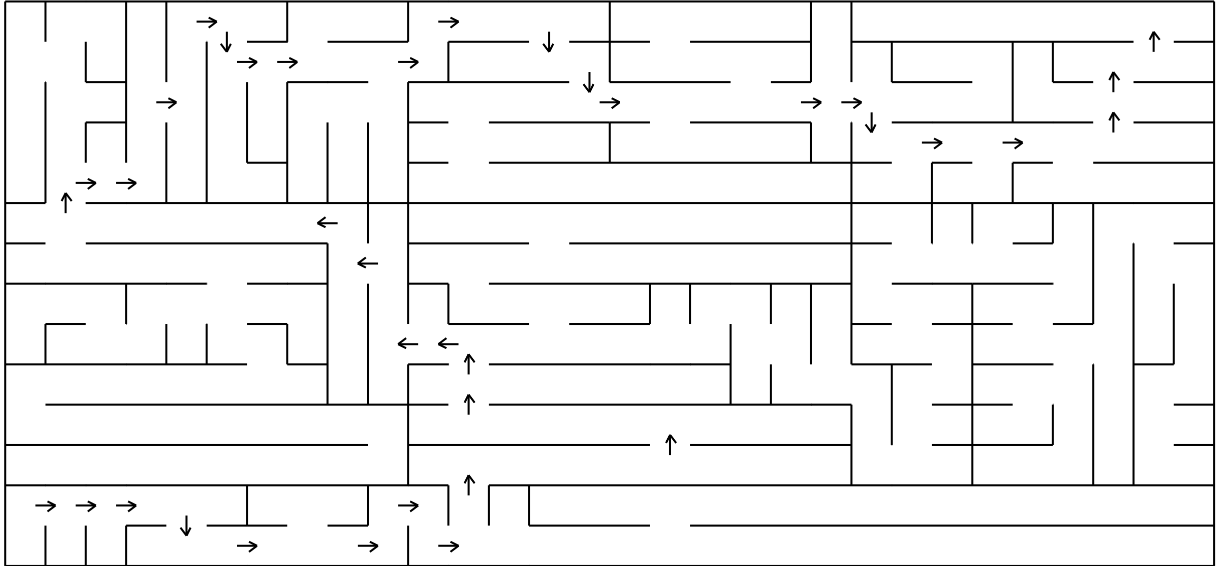
- s er i L_1 og t er i L_2. Så går vi fra s til d i L_1 og fra d til t i L_2.
- s er i L_2 og t er i L_1. Så går vi fra s til d i L_2 og fra d til t i L_1.
- s og t er begge i L_1. Så går vi blot fra s til t i L_1.
- s og t er begge i L_2. Så går vi blot fra s til t i L_2.

Denne metode implementeres af proceduren SolveLaby

```
Proc SolveLaby [L: Laby] (s, t: Point)
    if is(L) →
        (+ Var sL₁, tL₁: Bool
           sL₁ := SouthWest [L.w] (s)
           tL₁ := SouthWest [L.w] (t)
           if sL₁ ∧ ¬ tL₁ →
                SolveWall [L.w] (sL₁)
                SolveLaby [L.L₁] (s, L.w.p₂)
                SolveLaby [L.L₂] (L.w.p₂, t)
            | ¬ sL₁ ∧ tL₁ →
                SolveWall [L.w] (sL₁)
                SolveLaby [L.L₂] (s, L.w.p₂)
                SolveLaby [L.L₁] (L.w.p₂, t)
            | sL₁ ∧ tL₁ →
                SolveLaby [L.L₁] (s, t)
            | ¬ sL₁ ∧ ¬ tL₁ →
                SolveLaby [L.L₂] (s, t)
           fi
        +)
    fi
end SolveLaby

Proc SolveWall [w: Wall] (sL₁: Bool)
    (+ Var ver: Bool
        ver := w.p₁.x = w.q₂.x
        if ver ∧ sL₁ → Arrow (w.p₂, 0)
         | ¬ ver ∧ sL₁ → Arrow (w.p₂, 1)
         | ver ∧ ¬ sL₁ → Arrow (w.p₂, 2)
         | ¬ ver ∧ ¬ sL₁ → Arrow (w.p₂, 3)
        fi
    +)
end SolveWall
```

Proceduren SouthWest afgører om et punkt er sydvest for den angivne skillevæg, og dermed hvilken dellabyrint, det befinder sig i; proceduren Arrow tegner en pil i en given retning.

Vejen fra det sydvestlige til det nordøstlige hjørne ser således ud



Hvis ovenstående definitioner befinder sig på filen `labydef.tri`, så vil følgende program konstruere, tegne og finde vej i en labyrint.

```
(+ @"labydef.tri"

    Var L: Laby
    Var sw, ne: Point
    sw, ne := Point(1, 1), Point(31, 15)
    Frame(sw, ne)
    L := MakeLaby(sw, ne)
    DrawLaby[L]
    SolveLaby[L](sw, ne)
+)
```

13.6 Eksempel: UNIX filsystem

Vi slutter med et eksempel, der viser en ny anvendelse af sammenhængen mellem rekursive typer og rekursive procedurer.

Et *filsystem* i fx UNIX er organiseret som et træ, der kan beskrives med følgende typer

> **Type** FileList = **List**(File)
> **Type** File = **Prod**(name, data: Text)
> **Type** DirList = **List**(Directory)
> **Type** Directory = **Prod**(name: Text, files: FileList, dirs: DirList)

Et *directory* indeholder et antal navngivne filer og under-directories.

Kommandoerne til navigation i filsystemet i UNIX kan skitseres som følger

- *dir-navn* flytter til det angivne under-directory.
- *fil-navn* udskriver indholdet af den angivne fil.
- .. flytter til det unikke over-directory.

Sådanne kommandoer tillader, at man flytter sig rundt i træet og inspicerer de eksisterende filer. Simplifikationerne i forhold til de rigtige UNIX kommandoer er lavet for at undgå problemer med syntaksgenkendelsen.

Det er ganske ligetil at skrive en rekursiv procedure UNIX, der fortolker disse kommandoer korrekt. Rekursionen benyttes til at holde styr på vejen fra roden til det aktuelle directory.

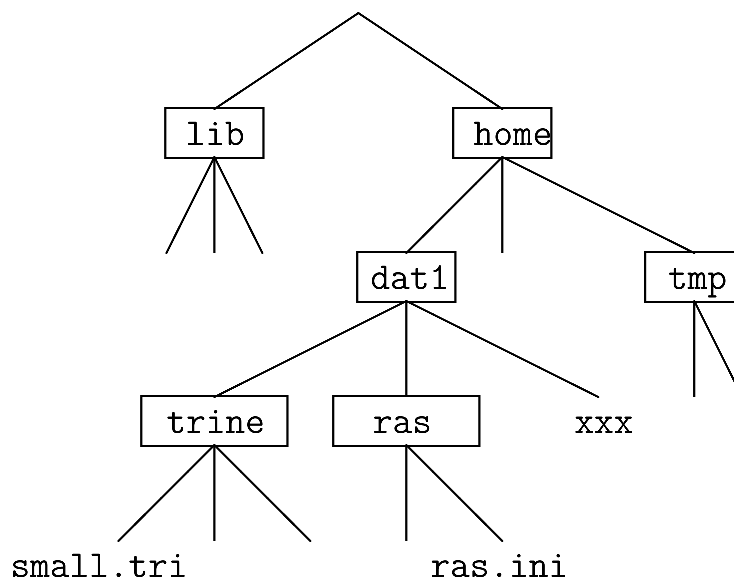
```
Proc UNIX[D: Directory]
    (+ Var comm: Text
       Var i: Int
       do true →
            write("> ")
            read[comm]
            if comm = ".." → return
            & true →
                 i := FindDir[D.dirs](comm)
                 if i ≠ ?-Int → UNIX[D.dirs.(i)]
                 & true →
                      i := FindFile[D.files](comm)
                      if i ≠ ?-Int → write(D.files.(i).data, eol)
                      & true → write("No such file or directory", eol)
                      fi
                 fi
            fi
       od
    +)
end UNIX
```

Procedurerne FindDir og FindFile benytter skabelonen for lister til at lede efter et navn i en liste.

```
Proc FindDir[DL: DirList](t: Text) → (Int)
    (+ Var i: Int
       i := 0
       do i<| DL | →
            if DL.(i).name = t → return i fi
            i := i+1
       od
       return ?-Int
    +)
end FindDir
```

```
Proc FindFile[FL: FileList] (t:Text) → (Int)
    (+ Var i: Int
        i := 0
        do i<| FL | →
            if FL.(i).name = t → return i fi
            i := i+1
        od
        return ?-Int
    +)
end FindFile
```

Bemærk, at kommandoen .. afgivet i roden af filsystemet bevirker, at proceduren UNIX terminerer. Betragt nu et filsystem med følgende udseende, hvor directory navne er indrammede



Følgende dialog med systemet kan nu opnås

```
> home
> dat1
> yyy
No such file or directory
> trine
> small.tri
Process S skip end S
```

```
> ..
> ras
> ras.ini
...
```

På dette sted vil rekursiondybden være lig med tre.

13.7 Katekismus

△ Rekursive typer har sig selv som komponenttype.

△ Værdier af en rekursiv type er altid endelige træer.

△ En værdi af en rekursiv type kan altid gennemløbes af en rekursiv procedure, således at værdien bliver procedurens udførelsestræ.

△ Værdimængder af rekursive typer findes ved at løse ligninger.

△ Normalformer af rekursive typer bliver uendelige træer (eller Ω).

14 Pointere

Mange programmeringssprog understøtter ikke rekursive typer, men har i stedet begrebet *pointere*. Rekursive typer er på mange måder blot en abstrakt overbygning på pointere. I dette afsnit præsenteres pointere som en selvstændig mekanisme, og der gives en grundig sammenligning med rekursive typer.

14.1 Pointer typer

Der er i TRINE en yderligere typekonstruktør, der skrives som følger

Type P = **Pointer**(T)

Værdimængden for typen P består, udover standardværdien, af pegepinde (pointere) til *variabler* af type T. For at følge en given tradition tillades det, at man skriver **nil** i stedet for ?-P eller ?-Pointer.

En pointertype har tre forskellige operationer tilknyttet. Hvis t er et værdiudtryk af type T, så er

$$P(t)$$

et værdiudtryk af type P. Når dette udtryk beregnes, så vil der blive oprettet en *ny* variabel af type T med indhold t; resultatet af selve udtrykket er en pegepind til denne nye variabel. Vi har altså, at beregningen af dette udtryk har en *sideeffekt* – det vil give et forskelligt resultat hver gang det beregnes. Vi tillader også den anonyme form

Pointer(t)

der kan bruges uden kendskab til navnet P. Hvis p er et udtryk af type P, så er

ref(p)

et variabeludtryk af type T, der angiver den variabel som p peger på. Hvis p angiver standardværdien, så er det en fejl at forsøge at beregne udtrykket.

Hvis x er et variabeludtryk af type P, så vil sætningen

dispose[x]

frigive den variabel, som værdien af x peger på. Efter udførelsen har x værdien ?-P. Man er således selv ansvarlig for at rydde op efter sig.

14.2 Pointere og tilstandstabeller

Med et meget simpelt eksempel kan vi illustrere, hvordan pointere påvirker tilstandstabellerne.

Betragt definitionen

Type Pint = **Pointer** (Int)

og lad p og q være variabler af type Pint. Til at begynde med, kan vi have tilstanden

N	V
p: v	v: ?
q: w	w: ?

Vi kan skabe en ny variabel med sætningen

p := Pint (87)

og opnå tilstanden

N	V
p: v	v: $\uparrow v_1$
q: w	w: ?
	v_1: 87

Vi benytter notationen $\uparrow v_1$ til at angive den pointerværdi, der peger på variablen v_1. Fra nu af angiver

ref (p)

en almindelig heltalsvariabel, nemlig v_1, så vi kan udføre

ref(p):=ref(p)+1

hvilket giver tilstanden

N	V
p: v	v: $\uparrow v_1$
q: w	w: ?
	v_1: 88

Vi kunne også skabe en variabel til q, men i stedet udfører vi tilordningen

q:=p

Nu får vi tilstanden

N	V
p: v	v: $\uparrow v_1$
q: w	w: $\uparrow v_1$
	v_1: 88

Bemærk, at der *ikke* foregik en kopiering med oprettelse af en ny delvariabel; p og q peger på den *samme* variabel. Det betyder, at optællingerne

ref(p):=ref(p)+1
ref(q):=ref(q)+1

resulterer i

N	V
p: v	v: $\uparrow v_1$
q: w	w: $\uparrow v_1$
	v_1: 90

hvor ref(p) og ref(q) begge har værdien 90. Hvis man ikke var klar over, at p og q pegede på den samme variabel, så ville dette være en særdeles overraskende situation. Hvis vi frigiver p's variabel med sætningen

dispose[p]

så bliver tilstanden denne

N	V
p: v	v: ?
q: w	w: $\uparrow v_1$

hvor p indeholder standardværdien, men hvor w stadig peger på v_1, der imidlertid ikke længere eksisterer i tilstandstabellen. Dette er ikke en veldefineret situation og er et af de problemer, som man skal være forsigtig med i forbindelse med brugen af pointere. Hvis vi nu udfører sætningerne

$$p := \text{Pint}(129)$$
$$q := \text{Pint}(129)$$

så kommer vi til tilstanden

N	V
p: v	v: $\uparrow v_1$
q: w	w: $\uparrow v_2$
	v_1: 129
	v_2: 129

Nu er det tilfældet, at sammenligningen

$$p = q$$

ikke er sand, da p og q peger på henholdsvis v_1 og v_2. Derimod er sammenligningen

$$\text{ref}(p) = \text{ref}(q)$$

sand, da begge udtryk angiver tallet 129. Hvis vi nu udfører

$$p := \text{Pint}(214)$$

så bliver tilstanden

N	V
p: v	$v : \uparrow v_3$
q: w	$w : \uparrow v_2$
	$v_1 : 129$
	$v_2 : 129$
	$v_3 : 214$

Nu er v_1 (og den plads, den optager i lageret) tabt for eftertiden, da den ikke længere peges på af noget. Man skal således huske at rydde op efter sig, hvis man er interesseret i at genbruge pladsen. Man kan heller ikke længere få fat på værdien af v_1.

Dette meget simple eksempel illustrerer faktisk samtlige aspekter af pointeres værdi- og variabelmekanismer.

14.3 Rekursive typer med pointere

Et af hovedformålene med pointertyper er at opbygge strukturer i stil med værdierne af rekursive typer.

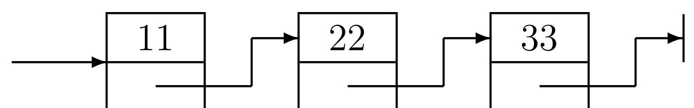
Ved hjælp af pointere kan man altid efterligne rekursive typer. Vi skal senere observere, at det modsatte ikke er tilfældet. Betragt fx den rekursive type

Type Link = **Prod**(val: Int, next: Link)

Ved hjælp af pointere kan vi efterligne den på følgende facon

Type Link = **Pointer**(Node)
Type Node = **Prod**(val: Int, next: Link)

En pointerliste kan bekvemt angives ved tegninger, hvor pointerne bliver til pile. Fx ser pointerlisten $(11, 22, 33)$ ud som



hvor **nil** tegnes som en “jordet” pointer.

Pointerlisten kan i høj grad bruges på samme måde som den rekursive type. Summen af en rekursiv listes elementer kan beregnes med følgende rekursive procedure

```
Proc Add[L: Link] → (Int)
    if ¬ is(L) → return 0
    & true → return L.val+Add[L.next]
    fi
end Add
```

Vi kan skrive en tilsvarende rekursiv procedure, der beregner summen af en pointerlistes elementer

```
Proc Add(L: Link) → (Int)
    if L = nil → return 0
    & true → return ref(L).val+Add(ref(L).next)
    fi
end Add
```

To ting skal bemærkes her: der er ingen grund til benytte en variabelparameter, da en pointerværdi er simpel; og vi er nødt til eksplicit at gå fra pointer til variabel med ref(...).

Imidlertid kan man også benytte en pointerliste *uden* at bruge rekursion. Vi kan skrive følgende *iterative* version af proceduren

```
Proc Add(L: Link) → (Int)
    (+ Var a: Int
       a := 0
       do L ≠ nil →
            a := a+ref(L).val
            L := ref(L).next
       od
       return a
    +)
end Add
```

En tilsvarende procedure kunne *ikke* skrives for den rekursive type. Vi ville komme til at *kopiere* den resterende liste i hvert gennemløb. Det specielle ved pointere er netop muligheden for at *pege* på variabler.

Vi får dog også nye problemer med pointerlisten. Hvis L_1 og L_2 er variabler af pointertypen Link, så vil tilordningen

$$L_1 := L_2$$

forårsage, at L_1 peger på den samme struktur som L_2 og *ikke* på en kopi. Ændringer i ref(L_1) vil også tage effekt i ref(L_2), hvilket sikkert ikke er tanken. Vi må istedet skrive en procedure, der foretager en korrekt kopiering

```
Proc Copy(L: Link) → (Link)
    if L = nil → return nil
    & true → return Link(Node(ref(L).val, Copy(ref(L).next)))
    fi
end Copy
```

Den ønskede tilordning bliver nu

$$L_1 := \mathrm{Copy}(L_2)$$

Hvis L_1 imidlertid havde en værdi i forvejen, så ville den ikke blive ryddet op. Til det formål må vi skrive en anden procedure

```
Proc Kill[L: Link]
    if L ≠ nil →
        Kill[ref(L).next]
        dispose[L]
    fi
end Kill
```

Bemærk rækkefølgen af kaldene af Kill og dispose. Videre er sammenligningen

$$L_1 = L_2$$

heller ikke ønskværdig, idet vi herved sammenligner pegepindene og ikke hele listerne. Vi må skrive endnu en procedure

```
Proc Equal(L1, L2: Link) → (Bool)
    if (L1 = nil) ∨ (L2 = nil) → return L1 = L2
    & true → return (ref(L1).val = ref(L2).val) ∧
                    Equal(ref(L1).next, ref(L2).next)
    fi
end Equal
```

der foretager den korrekte sammenligning.

Slutteligt kan vi overveje hvordan vi kan skrive en liste ud på skærmen. Sætningen

```
write(L)
```

vil kun udskrive pegepindens værdi, hvilket ikke fortæller os noget om listen. Vi må skrive endnu en procedure

```
Proc Print(L: Link)
    write("(")
    do L ≠ nil →
        write(ref(L).val)
        L := ref(L).next
        if L ≠ nil → write(",") fi
    od
    write(")")
end Print
```

der giver den sædvanlige udskrift.

Historisk set blev pointerne introduceret først. De rekursive typer er senere blevet udviklet for netop at undg skrivning af et stort antal procedurer af ovenstående slags.

14.4 Kanoniske tekstformater

Som med alle andre typer i TRINE kan man også udskrive og indlæse pointerværdier. Man vil dog næppe have et stort udbytte af dette, da en pointer blot er en pegepind, der ikke kan forstås uden reference til hele tilstandstabellen.

Man har derfor ikke stor gavn af at se på disse værdier. Der er heller ikke nogen mening i at gemme dem mellem programkørsler, med fx dump og load, da hvert program jo har en forskellig tilstandstabel. Hvis man har opbygget en større struktur med pointere, så er det faktisk ret besværligt at gemme den på en fil.

Udskrifter af pointerværdier kan fx se ud som følger

```
0x244c8
0x24560
0x2456c
```

og er essentielt blot det oversatte programs interne navne for variabler.

14.5 Eksempel: cyklisk stak

Ved hjælp af pointere kan man ofte lave ekstremt effektive datatyper, hvor alle operationer kun tager konstant tid. Pointere tillader nemlig, at man opretholder flere pegepinde til den samme variabel.

Et simpelt eksempel er en *cyklisk stak*, der har operationerne Init, Push, Top og Rotate og kan specificeres som følger

Box Cyclic
  Type Stack = ≪liste af heltal≫

  Proc Init [S: Stack]
    ≪S := ()≫
  end Init

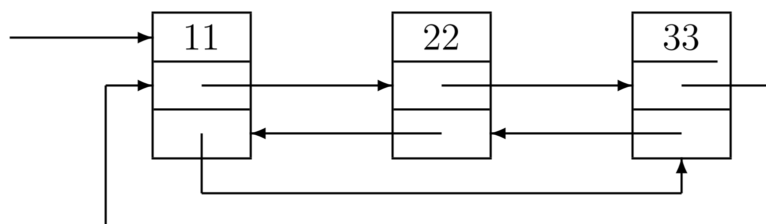
```
    Proc Empty [S: Stack] → (Bool)
        return ≪S=()≫
    end Empty

    Proc Push [S: Stack] (i: Int)
        ≪S := (i)++S≫
    end Push

    Proc Top [S: Stack, i: Int]
        ≪i := S.(0)≫
    end Top

    Proc Rotate [S: Stack]
        ≪S := S(1..|S|)++S(0..1)≫
    end Rotate
end Cyclic
```

Ved at realisere Stack som en *cyklisk dobbeltkædet* pointerliste kan vi implementere alle de ovenstående operationer i garanteret konstant tid. Ideen er, at hvert element i listen peger på både sin forgænger og sin efterfølger.

```
Box Cyclic
    Type Stack = Pointer (Node)
    Type Node = Prod (val: Int, next, last: Stack)

    Proc Init [S: Stack]
        S := nil
    end Init

    Proc Empty [S: Stack] → (Bool)
        return S=nil
    end Empty
```

```
    Proc Push[S: Stack] (i: Int)
        if S = nil →
            S := Stack(Node(i, nil, nil))
            ref(S).last := S
            ref(S).next := S
        & true →
            (+ Var T: Stack
                T := Stack(Node(i, ref(S).last, S))
                ref(ref(S).last).next := T
                ref(S).last := T
                S := T
            +)
        fi
    end Push

    Proc Top[S: Stack, i: Int]
        i := ref(S).val
    end Top

    Proc Rotate[S: Stack]
        if S ≠ nil → S := ref(S).next fi
    end Rotate
end Cyclic
```

Læseren opfordres til at eftervise, at disse operationer er implementeret korrekt. Det kan med fordel gøres ved hjælp af tegninger, som den følgende af den dobbeltkædede liste $(11, 22, 33)$



Det ses let, at alle operationerne foregår i konstant tid. Vi har dermed opnået en optimal implementation af den cyckliske stak.

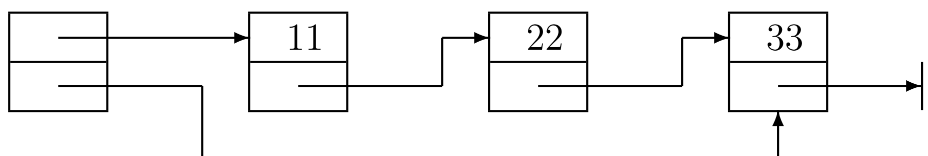
14.6 Eksempel: hurtige sekvenser

Det sidste eksempel viser en implementation af sekvenser, i hvilken alle operationer foregår i konstant tid.

Vi skal implementere følgende polymorfe datatype

```
Box IESCS(E)
    Type Seq = ≪sekvenser af E-værdier≫

    Proc Init [S: Seq]
        ≪S := ()≫
    end Init

    Proc Empty [S: Seq] → (Bool)
        return ≪S=()≫
    end Empty

    Proc Single [S: Seq] (e: E)
        ≪S := (e)≫
    end Single

    Proc Combine [S, R: Seq]
        ≪S, R := S++R, ()≫
    end Combine

    Proc Split [S: Seq, h: E, T: Seq]
        ≪h, T, S := S.(0), S(1..|S|), ()≫
    end Split
end IESCS
```

Her kan repræsentationen af listen (11,22,33) skitseres på følgende måde, hvor **nil** tegnes som en "jordet" pointer



Implementationen bliver nedenstående polybox

```
Box IESCS(E)
    Type Link = Pointer(Node)
    Type Node = Prod(val: E, next: Link)
    Type Seq = Prod(first, last: Link)

    Proc Init[S: Seq]
        S := Seq(nil, nil)
    end Init

    Proc Empty[S: Seq] → (Bool)
        return S.first=nil
    end Empty

    Proc Single[S: Seq](e: E)
        (+ Var L: Link
            L := Link(Node(e, nil))
            S := Seq(L, L)
        +)
    end Single

    Proc Combine[S, R: Seq]
        if S.first = nil → S :=: R
        & R.first ≠ nil →
              ref(S.last).next := R.first
              S.last := R.last
              R := Seq(nil, nil)
        fi
    end Combine
```

```
    Proc Split [S: Seq, h: E, T: Seq]
        if S.first ≠ nil →
            h := ref(S.first).val
            if S.first = S.last → T := Seq(nil, nil)
            & true → T := Seq(ref(S.first).next, S.last)
            fi
            S := Seq(nil, nil)
        fi
    end Split
end IESCS
```

Alle operationer tager konstant tid.

14.7 Pointere til eksisterende variabler

Indtil nu har vi kun kunnet oprette pointere til nye variabler, der blev skabt til lejligheden. Det er imidlertid ogs muligt at pege p variabler, der allerede findes i tilstanden. Hvis x er et variabeludtryk at type T, s er

pointto [x]

et vrdiudtryk af type **Pointer**(T), der peger p variablen angivet af x. Betragt flgende stning

```
(+ Var x: Int
   Var p: Pointer (Int)
   x := 87
   p := pointto [x]
+)
```

Efter tilordningen til p har vi tilstanden

N	V
x: v_1	v_1: 87
p: v_2	v_2: $\uparrow v_1$

14.7.1 Eksempel: trer med rodhftede blade

Sdanne pointere kan bruges til at skabe strukturer, der blander rekursive typer og pointere. Antag, at vi skal skrive en datatype, hvis vrdier er binre trer, i hvilke bladene indeholder en pointer til trets rod. Vi kan benytte flgende type

```
Type R = Sum(leaf: Pointer(R),
             node: Prod(left, right: R))
```

hvor det underliggende binre tr er defineret med en rekursiv type. Flgende procedure vil nu hfte bladenes pointere op til roden

```
Proc Staple[r: R]
    (+ Proc S[root, r: R]
           if is(r, leaf) → r.leaf := pointto[root]
           & is(r, node) →
                S[root, r.node.left]
                S[root, r.node.right]
           fi
       end S

       S[r, r]
    +)
end Staple
```

Hvis vi ikke havde haft pointto-operationen, s kunne vi ikke have benyttet en rekursiv type. Istedet mtte vi have brugt definitionen

```
Type R = Sum(leaf: Pointer(R),
             node: Prod(left, right: Pointer(R)))
```

med de allerede beskrevne komplikationer til flge.

14.8 Katekismus

△ En pointertypes værdier er pegepinde til variabler af elementtypen.

- △ Pointere tillader, at man har flere pegepinde til den samme variabel.
- △ En rekursiv type kan efterlignes med pointere.
- △ Man kommer dog til at skrive mange små procedurer selv til fx kopiering, oprydning, sammenligning og eksterne kommunikationer.
- △ Eksterne kommunikationer direkte med pointere er ikke interessante.
- △ Ved hjælp af pointere kan man lave meget effektive datastrukturer.
- △ Pointere og rekursive typer kan blandes ved hjlp af pointto.

15 Systemer, processer og kanaler

Indtil nu har vi kun beskæftiget os med TRINE-programmer, der består af en enkelt proces. Som nævnt i indledningen kan man imidlertid også beskrive flere simultane processer som et *system* med følgende udseende

System S
  Prc_1
  Prc_2
  $\vdots$
  Prc_k
end S

hvor Prc_i'erne er processer. Alle processerne er aktive på samme tid; systemet terminerer, når alle processerne er terminerede. Der er ikke som sådan noget vundet ved at kunne udføre flere processer; man kunne jo blot udføre dem en ad gangen efter hinanden. Det bliver først interessant når processerne kan *samarbejde*.

15.1 Kanaler og kommunikation

Forudsætningen for samarbejde er *kommunikation*; processerne må kunne udveksle information. Da det ikke er ønskværdigt, at processerne har kendskab til hinandens indre – det vil sige kan læse og skrive hinandens variabler – indfører vi endnu et begreb, nemlig *kanaler*.

En kanal har et *navn* og en *type*. Definitionen

Chan $c : T$

definerer en kanal med navn c og type T. Kanaler defineres i systemet udenfor processerne, og hører således til de statiske omgivelser for dem alle. Her kan man også definere typer, procedurer og boxe på sædvanlig vis. Når en proces udfører sætningen

$c\ !\ u$

hvor u er et værdiudtryk af type T, så udregnes u i processens aktuelle tilstand, og den tilsvarende værdi *sendes* på kanalen c. Processen *venter*, indtil værdien er blevet modtaget af en anden proces. Når en proces udfører sætningen

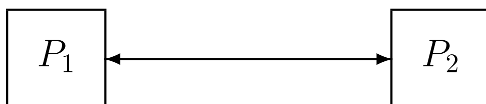
$$c \; ? \; x$$

hvor x er et variabeludtryk af type T, så udregnes x i den aktuelle tilstand, og processen *venter* på at *modtage* en værdi fra kanalen c, der så tilordnes den tilsvarende variabel. Selve kanalkommunikationen svarer således til tilordningen

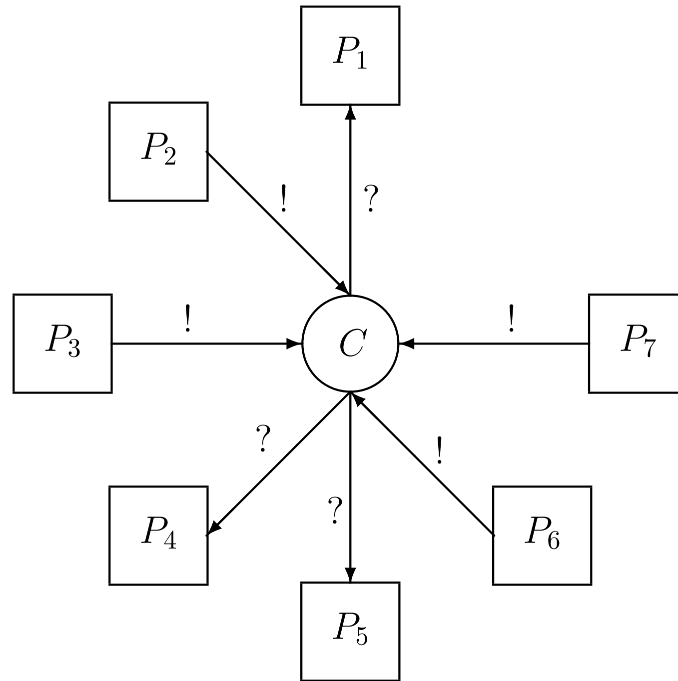
$$x := u$$

hvor højre- og venstresiden tilhører hver sin proces. Kommunikationen er *synkroniseret*, fordi begge processer venter med at udføre resten af sætningen til tilordningen er foretaget.

I det simpleste tilfælde kan vi tænke på kanalen som en ledning mellem to processer



De to operationer ? og ! svarer til at lytte og tale på ledningen. Imidlertid er kanalerne mere generelle end som så. De kan nemlig involvere vilkårligt mange processer. På et givet tidspunkt vil et antal processer have forbindelse med kanalen, som illustreret i følgende tegning, hvor processer tegnes som kvadrater og kanaler som cirkler.



Nogle processer vil sende, og andre vil modtage. Hvis der er mindst én af hver, så vil en kommunikation finde sted som beskrevet ovenfor. En kanal har en iboende nondeterminisme. Hvis der er s processer, der ønsker at sende, og m processer, der ønsker at modtage, så er der $s \times m$ *potentielle* kommunikationer, hvoraf én vil blive valgt til udførelse. I eksemplet ovenfor kan følgende par af processer potentielt kommunikere

$$(P_1, P_2), (P_1, P_3), (P_1, P_6), (P_1, P_7), (P_4, P_2), (P_4, P_3)$$
$$(P_4, P_6), (P_4, P_7), (P_5, P_2), (P_5, P_3), (P_5, P_6), (P_5, P_7)$$

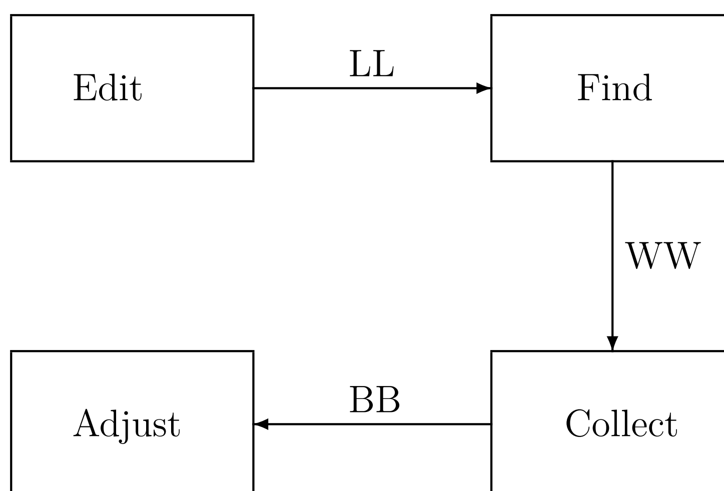
Vi kan på denne måde tænke på kanalen som en værdi-*børs*, med *sælgere* og *købere*.

15.2 Programmering med flere processer

Systemer med flere *processer* introducerer et væld af nye muligheder og problemer for programmering. De fleste af disse ligger uden for rammerne af denne bog, hvorfor vi indskrænker os til at præsentere nogle eksempler på naturligt forekommende problemer, der kan løses meget lettere ved hjælp af flere processer.

15.2.1 Eksempel: avisspalter

Et klassisk eksempel på et problem, der med fordel kan udnytte flere processer, er produktion af avisspalter. Vi er givet en tekst bestående af en række ord, der skal formatteres som en spalte af en vis bredde med lige højremargen. Man skal simultant finde ordene, finde det rigtige antal der skal ud på samme linje, justere linjen med ekstra blanke og skrive den ud. Man kan meget elegant gøre dette ved at bruge 4 processer. De organiseres som et *samlebånd*, hvor produktet af hver proces sendes videre til yderligere forarbejdning hos den næste.



Den første proces svarer til journalistens skrivemaskine og indlæser teksten en linje ad gangen. Disse *linjer* sendes videre til den anden process, der deler dem op i enkelte ord. Disse *ord* sendes til den tredje process, der bundter dem sammen. Et *bundt* indeholder så mange ord, der kan være på en enkelt linje uden at give overløb, sammen med en angivelse af hvor mange ekstra blanke, der skal sættes ind. Den sidste proces er en printer, der fordeler de ekstra blanke tilfældigt mellem ordene i et bundt og skriver den færdige linje ud. Vi får brug for følgende typer og kanaler

Type Bunch = **Prod**(blanks: Int, word: **List**(Text))

Chan LL, WW: Text
Chan BB: Bunch

Den første proces ser således ud

```
Process Edit
  do true →
     (+ Var t: Text
        read[t]
        LL ! t++" "
     +)
  od
end Edit
```

Linjerne læses én ad gangen og sendes videre afsluttet af en ekstra blank. Man kunne her forestille sig at have et helt system til tekstbehandling, hvor linjerne kan redigeres flere gange, inden de sendes videre til det egentlige spalteprogram. Den anden proces er

```
Process Find
  do true →
     (+ Var line: Text
        LL ? line
        (+ Var i, d: Int
           i, d := 0, 0
           do i+d < | line | →
                if line.(i+d) ≠ ' ' → d := d+1
                & d = 0 → i := i+1
                | d > 0 →
                     WW ! line(i .. i+d)
                     i, d := i+d+1, 0
                fi
           od
        +)
     +)
  od
end Find
```

der kan ses at virke ligesom et tidligere eksempel, der fandt blokke af cifre i en tekst.

Den tredje proces skrives som

```
Process Collect
   (+ Var cw: Int
      write("column width: ")
      read[cw]
      (+ Var used: Int
         Var word: List(Text)
         used, word := 0, List()
         do { en linje bestående af ordene i word
                fylder used+|word|-1 }
            true →
               (+ Var t: Text
                  WW ? t
                  if used+| word |+| t | ≤ cw →
                        used := used+| t |
                        word := word++List(t)
                  & true →
                        BB ! Bunch(cw-used-| word |+1, word)
                        used, word := | t |, List(t)
                  fi
               +)
         od
      +)
   +)
end Collect
```

Processen opsamler ord, indtil den indlæste spaltebredde overskrides.

Den sidste proces ser ud som følger

```
Process Adjust
    do true →
        (+ Var b: Bunch
            BB ? b
            (+ Var m: Vector
                Distribute[m](b)
                (+ Var i: Int
                    i := 0
                    do i < | b.word | →
                            write(b.word.(i), Text(' ' | m.(i)))
                            i := i+1
                    od
                    write(eol)
                +)
            +)
        +)
    od
end Adjust
```

Vi benytter her proceduren Distribute, der tilfældigt fordeler de overskydende blanke mellem ordene

```
Proc Distribute[m: Vector](b: Bunch)
    if | b.word | = 1 → m := Vector(0)
    & true →
        m := Vector(1 | | b.word |-1)++Vector(0)
        do b.blanks > 0 →
            (+ Var i: Int
                i := random(0, | b.word |-1)
                m.(i), b.blanks := m.(i)+1, b.blanks-1
            +)
        od
    fi
end Distribute
```

Hver af disse processer er ganske enkel at skrive og kan forbedres og ud-

skiftes, uden at man skal ændre de øvrige. Hvis vi vil tillade *orddeling*, så kan vi (blandt andet) tilføje en ekstra proces. Vi har fået en *faktorisering* af vores beregning. Hvis vi havde flere processorer, så ville spaltningen ovenikøbet gå hurtigere, da alle processerne kan arbejde samtidigt.

15.3 Nondeterministisk kommunikation

Når en proces udfører $c \,!\, u$ eller $c \,?\, x$, så har den bundet sig til at kommunikere på kanalen c. Hvis den potentielt er lige interesseret i to forskellige kommunikationer, så kan dette valg være begrænsende. Vi indfører derfor en mere generel kommunikationssætning

$$\begin{array}{l} \textbf{co}\ C_1 \ \rightarrow\ S_1 \\ \ \ \mid\ C_2 \rightarrow\ S_2 \\ \ \ \vdots \\ \ \ \mid\ C_n \ \rightarrow\ S_n \\ \textbf{oc} \end{array}$$

hvor C_i'erne er kommunikationer som før, og S_i'erne er sætninger. Når en proces udfører **co**-sætningen, så forsøger den *simultant* at udføre samtlige kommunikationer. Når en af disse er mulig, så fuldføres den, og det tilsvarende S_i udføres. Hvis flere kommunikationer er mulig samtidig, så vælges der *nondeterministisk* en af disse. Herefter er **co**-sætningen udført; i lighed med en **if**-sætning vil præcist ét S_i bliver udført. Den simple kommunikation C er således ækvivalent med

$$\textbf{co}\ C \rightarrow \textbf{skip oc}$$

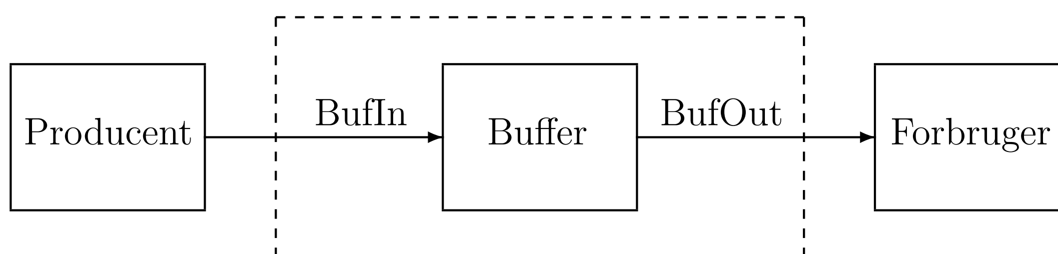
15.3.1 Eksempel: begrænset buffer

Der findes processer, hvis opførsel *kun* kan realiseres ved hjælp af nondeterministisk kommunikation. Et eksempel på dette er en (begrænset) *buffer*.

Vi betragter et system med to processer, en *producent* og en *forbruger*. Da disse to processer kører uafhængigt af hinanden og med varierende hastig-

heder, kan der opstå ventetider på to måder. Hvis producenten er hurtigere end forbrugeren, så kan det blive nødvendigt at indstille produktionen, indtil der igen er afsætning. Omvendt kan en hurtig forbruger blive nødt til at vente på leverancer fra en langsom producent.

I et system med varierende hastigheder kan man i høj grad undgå sådanne ventetider ved at indføre en *buffer*. Dette er simpelt hen et lagerområde, hvor varer kan vente i transit fra producenten til forbrugeren. Nu kan producenten fortsætte selv om forbrugeren er optaget; varen kan blot placeres i bufferen, der dog ofte vil være af begræset størrelse. Omvendt kan forbrugeren fortsætte, sålænge bufferen ikke er tom. Situationen kan illustreres som følger



Hele arrangementet med de to kanaler og bufferen kan betragtes som en helhed: en *asynkron* kanal.

Vi kan implementere en begrænset buffer som en proces, der bestyrer en (cyklisk) kø af varer. Buffer processen er altid villig til både at modtage og videresende varer, medmindre køen er enten fuld eller tom. Vi har altså her en ægte nondeterministisk kommunikation. En simpel implementation af bufferen ser ud som følger

```
Process Buffer
   (+ Var Buf: Vector
      Var Max, Size, Low, High, n: Int
      Max := 10
      Buf, Size, Low, High := Vector(0 | Max), 0, 0, 0
      do true →
         write("Buffer size: ", Size, eol)
         if Size = 0 →
               BufIn ? n
               ≪indsæt n i Buf≫
          | Size = Max →
               BufOut ! Buf.(Low)
               ≪reducer Buf≫
          | 0 < Size < Max →
               co BufIn ? n → ≪indsæt n i Buf≫
                | BufOut ! Buf.(Low) → ≪reducer Buf≫
               oc
         fi
      od
   +)
end Buffer

where ≪indsæt n i Buf≫ is
         Buf.(High), High, Size := n, (High+1) mod Max, Size+1

where ≪reducer Buf≫ is
         Low, Size := (Low+1) mod Max, Size-1
```

Bufferen udskriver løbende størrelsen af køen. For at illustrere dens anvendelse lader vi producenten være brugeren, der finder på forskellige heltal. De varierende produktionstider kommer fra almindelig ubeslutsomhed. Forbrugerens opgave er at udskrive primtalsfaktoriseringen af tallene. Vi benytter en simpel algoritme, der kan give store variationer i beregningstiden. Det samlede system ser ud som følger

```
System P

    Chan BufIn, BufOut: Int

    Process Buffer
        ...
    end Buffer

    Process User
        do true →
            write("Write a number: ")
            (+ Var n: Int
                read[n]
                BufIn ! n
            +)
        od
    end User

    Process Factor
        do true →
            (+ Var n, p: Int
                BufOut ? n
                write(n, "=")
                p := 2
                do n ≠ 1 →
                    if n mod p = 0 →
                        write(p, "*")
                        n := n/p
                    & true → p := p+1
                    fi
                od
                write("1", eol)
            +)
        od
    end Factor

end P
```

Man kan observere, at jo større bufferkøen er, jo mindre ventetid vil der være i systemet.

Den asynkrone kanal, bestående af processen og de to kanaler, kan naturligvis også anbringes i et bibliotek til senere inklusion i forskellige programmer.

15.4 Ressourcedeling

Et vigtigt problem i forbindelse med multiple processer er administrationen af delte ressourcer. En grundig analyse af dette ligger uden for rammerne af denne bog, men vi slutter med et simpelt eksempel, der illustrerer nogle af pointerne.

15.4.1 Eksempel: tegneprocesser

En måde at gemme stregtegninger på er som en liste af *streger*, hvor hver streg er en liste af *punkter*. Ideen er, at man kan reproducere tegningen ved for hver streg at starte i det første punkt og flytte sig til de øvrige punkter i rækkefølge. Et eksempel på dette er følgende system (argumenterne dx og dy til procedurerne DrawLine og DrawFigure benyttes til forskydning af tegninger)

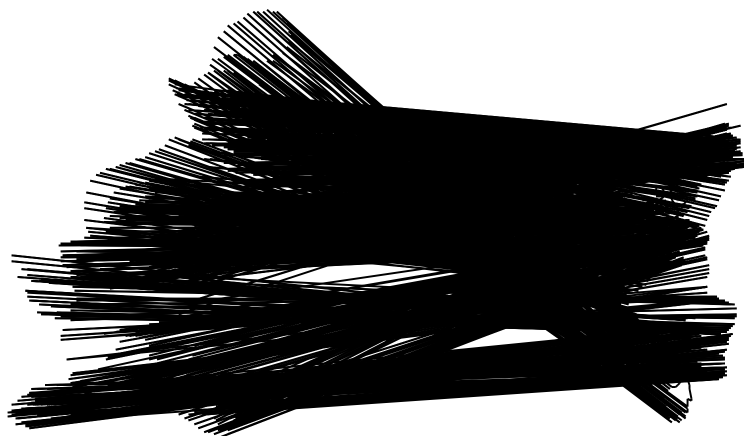
```
System F

    @"graphics.tri"

    Type Point = Prod(x, y: Int)
    Type Line = List(Point)
    Type Figure = List(Line)

    Proc DrawLine[s: Line](dx, dy: Int)
        (+ Var i: Int
           jumpto(s.(0).x+dx, s.(0).y+dy)
           i := 1
           do i < | s | →
                moveto(s.(i).x+dx, s.(i).y+dy)
                i := i+1
           od
        +)
    end DrawLine

    Proc DrawFigure[f: Figure](dx, dy: Int)
        (+ Var i: Int
           i := 0
           do i < | f | →
                DrawLine[f.(i)](dx, dy)
                i := i+1
           od
        +)
    end DrawFigure

    Process Wolf
        (+ Var f: Figure
           load[f]("wolf.fig")
           DrawFigure[f](0, 0)
        +)
    end Wolf
end F
```

Processen Wolf indlæser data fra en fil og tegner følgende figur

Det kunne nu tænkes, at vi gerne ville have tegnet to figurer *på samme tid*. Den simpleste løsning synes at være at tilføje endnu en proces

```
Process Pig
    (+ Var f: Figure
        load[f]("pig.fig")
        DrawFigure[f](300, 0)
    +)
end Pig
```

Til vores (store?) overraskelse får vi dette kaotiske resultat



Problemet er, at vi kun har én blyant, som begge processer prøver at bruge på samme tid. Vi er nødt til yderligere at *synkronisere* deres opførsel. Løsningen er at forbyde DrawLine *selv* at tegne linjestykker og i stedet forlange, at de sendes til en ny tredje proces, der "bestyrer" blyanten. Det

giver os det modificerede system

```
System F

    @"graphics.tri"

    Type Point = Prod(x, y: Int)
    Type Line = List(Point)
    Type Figure = List(Line)
    Type Segment = Prod(from, to: Point)

    Proc DrawLine[s: Line] (dx, dy: Int)
        (+ Var i: Int
           Var from, to: Point
           from := Point(s.(0).x+dx, s.(0).y+dy)
           i := 1
           do i < | s | →
                  to := Point(s.(i).x+dx, s.(i).y+dy)
                  L ! Segment(from, to)
                  from := to
                  i := i+1
           od
        +)
    end DrawLine

    Proc DrawFigure[f: Figure] (dx, dy: Int)
        (* som ovenfor *)
    end DrawFigure
```

```
    Chan L: Segment

    Process Draw
      do true →
          (+ Var l: Segment
             L ? l
             jumpto(l.from.x, l.from.y)
             moveto(l.to.x, l.to.y)
          +)
      od
    end Draw

    Process Wolf
      (* som ovenfor *)
    end Wolf

    Process Pig
      (* som ovenfor *)
    end Pig

end F
```

Nu lykkes det at få produceret den tiltænkte tegning

Eksemplet er underholdende, men illustrerer ogs udmærket de problemer og løsninger, som man involveres i, når flere samtidige processer skal dele en fælles ressource.

15.5 Katekismus

△ Et system består af et antal processer, der kommunikerer ved hjlp af kanaler.

△ Kanaler har typer, der beskriver de værdier, som de kan transmittere.

△ Man kommunikerer ved enten at sende på eller at modtage fra en kanal.

△ Kommunikationen foregår synkront.

△ Generelt kan man forsøge mange kommunikationer samtidigt.

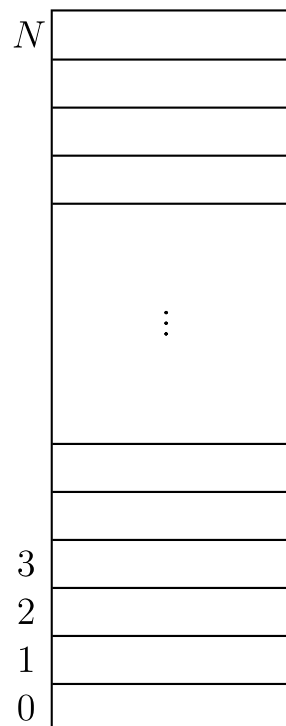
△ Indviklede kontrolstrukturer kan med fordel beskrives som multi-proces systemer.

16 Tid og plads

I de hidtidige kapitler har vi repræsenteret processernes tilstande ved hjælp af tabeller. I dette kapitel ser vi nærmere på, hvordan disse tilstandstabeller modsvares af lageret på en fysisk maskine. Denne undersøgelse leder også frem til en analyse af, hvor lang tid udførelsen af en given proces vil tage.

16.1 Et fysisk lager

En fysisk datamat har et *statisk* lager, som følgende illustration viser



Lageret består af et antal individuelle *celler*, og dets størrelse kan ikke ændre sig. Typisk vil N vre p adskillige millioner. Hver celle kan indeholde et *maskintal*, det vil sige et heltal i et fast interval, i vores tilflde $[-2147483648 \mathinner{..} 2147483647]$.

Datamaten er bygget til at kunne udføre forskellige simple manipulationer: beregninger på maskintallene, samt læsning og skrivning af cellernes indhold. En Trine-proces oversættes til en (lang) sekvens af sådanne simple

manipulationer, der udføres af datamaten. Vi vil her ikke beskæftige os nærmere med detaljerne i denne oversættelse.

16.2 Repræsentation af tilstandstabeller

Et lager er en lineær samling celler, hvorimod en tilstandstabel har mere struktur. En V-søjle indeholder variabler, der kan indeholde en samling delvariabler, der igen kan indeholde delvariabler, og så videre. Vi kan dog med lethed repræsentere en tilstand som et lagerbillede.

Hver (del)variabel i tilstanden svarer til netop én celle i lageret. Cellens indhold modsvarer variablens indhold på følgende måde.

- Hvis variablen indeholder ?, så indeholder cellen et specielt tal der angiver dette (fx -2147483648).
- Hvis variablen indeholder et heltal, så indeholder cellen dette tal.
- Hvis variablen indeholder et pseudoreelt tal, så indeholder cellen dette tal.
- Hvis variablen indeholder false eller true, så indeholder cellen enten 0 eller 1.
- Hvis variablen indeholder et tegn, så indeholder cellen det tilsvarende ASCII nummer.
- Hvis variablen indeholder en pointer, så indeholder cellen nummeret på en anden celle.
- Hvis variablen indeholder en sammensat værdi, så indeholder cellen nummeret på en *blok*.

En blok er et lille sammenhængende afsnit af lageret af følgende form

	n
1	
2	
3	
n	

Den første celle indeholder antallet af de efterfølgende celler; hver af disse svarer til en delvariabel. Med denne implementation kan vi se, at resultatet af et variabeludtryk altid er (nummeret på) en lagercelle.

Vi organiserer således det fysiske lager som et *dynamisk* lager af blokke.

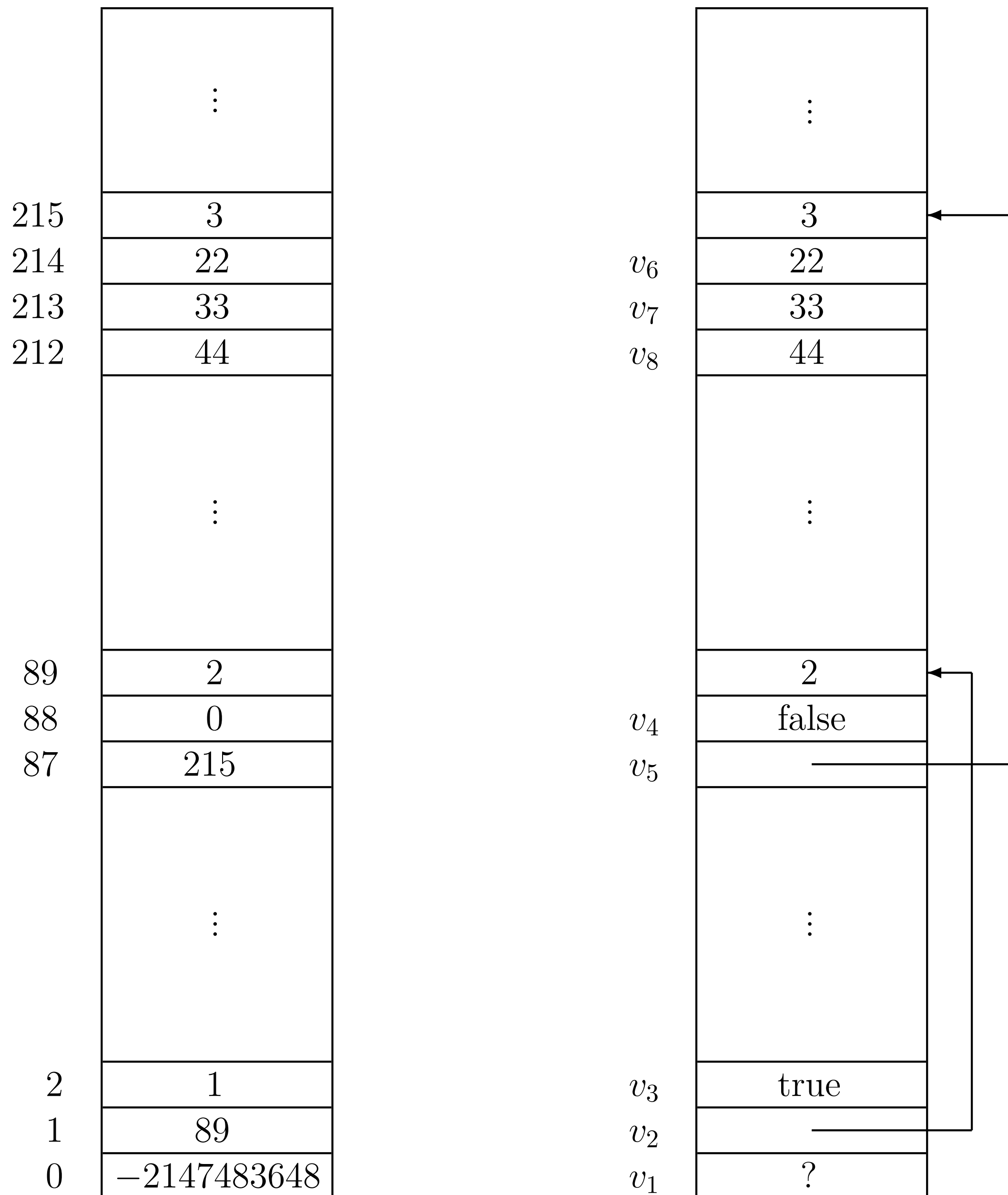
Da alle cellerne indeholder tal, er det udelukkende typesystemet, der fortæller, hvorledes de forskellige indhold skal fortolkes. I dette kapitel vil vi dog skrive de simple værdier, som vi plejer, og angive numre på lagerceller ved hjælp af pile.

Med disse beslutninger er det ganske simpelt at oversætte en tilstandstabel til et lagerbillede. De forskellige N-søjler optræder kun implicit, ved at deres variabler står nederst i lageret. Den øvrige del af lageret bruges som dynamisk lager for blokkene, der svarer til delvariablerne.

Fx vil tabellen

N	N	V
x: v_1	z: v_3	v_1: ?
y: v_2		v_2: (v_4, v_5)
		v_3: true
		v_4: false
		v_5: (v_6, v_7, v_8)
		v_6: 22
		v_7: 33
		v_8: 44

blive repræsenteret af følgende lagerbillede



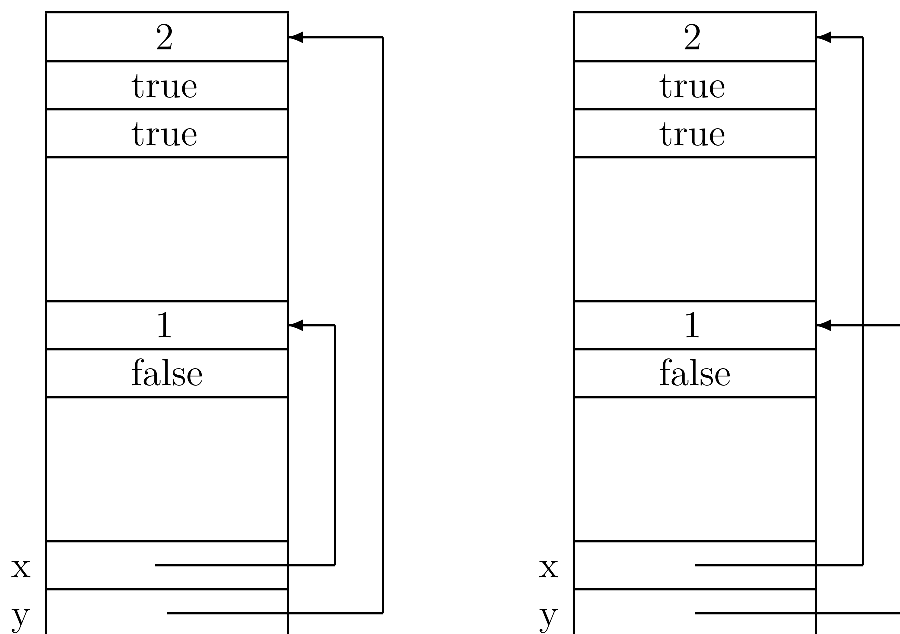
Til venstre ses det faktiske billede, og til højre ses den repræsentation som vi vil benytte fremover.

16.3 Tidsforbrug

Med denne implementation af en Trine-tilstand kan vi se, hvad tidsforbruget af de forskellige manipulationer bliver. En ombytning

```
x:=:y
```

vil kun tage konstant tid, da vi blot skal ombytte indholdet af cellerne svarende til x og y. Dette er sandt, uanset om deres typer er simple eller sammensatte. Tilstanden før og efter en sådan ombytning kan fx se ud som følger



Vi ser, at de to celler blot bytter indhold. Dette giver naturligvis essentielt samme resultat som tilordningen

x, y := y, x

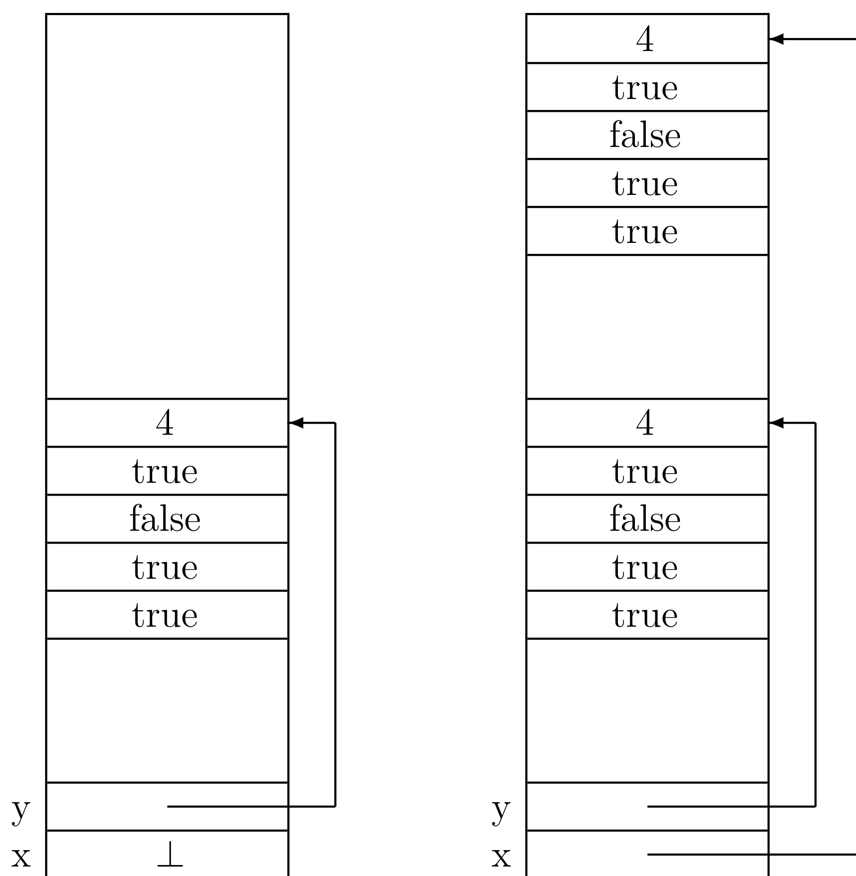
bortset fra, at vi slipper for at kopiere begge værdierne.

En overførsel af en variabelparameter vil tilsvarende kun tage konstant tid, da vi blot skal kopiere et nummer på en blok.

Derimod vil en tilordning

x := y

eller tilsvarende en overførsel af en værdiparameter, kunne tage meget længere tid, da vi skal *kopiere* værdien. Tilstanden før og efter en sådan tilordning kan fx se ud som følger



I almindelighed vil en tilordning tage tid proportional med antallet af lagerceller, som den tilordnede værdi fylder. For en værdi a kan vi beregne denne størrelse $\|a\|$ som følger

- $\|a\| = 1$, hvis a er en simpel værdi.
- $\|a\| = 2 + \|a_1\| + \|a_2\| + \cdots + \|a_k\|$, hvis $a = (a_1, a_2, \ldots, a_k)$.

Vi har fx, at

- $\|10\| = 1$
- $\|(11, 12, 13, 14)\| = 2 + 4 = 6$
- $\|((15, 16), (), (17), (18, 19, 20))\| =$
  $2 + (2 + 2) + (2 + 0) + (2 + 1) + (2 + 3) = 16$

Alle operationer på simple værdier, så som $+$, $-$ og $>$, tager konstant tid, hvilket svarer til, at de involverede værdier har størrelse 1.

Udvælgelse af delvariabler, det vil sige `x.(i)` og x.n, tager også konstant tid, da vi jo blot skal aflæse en enkelt lagercelle.

Operationen ref på pointere tager også konstant tid, da vi igen blot skal aflæse en enkelt lagercelle. Oprettelsen en ny pointervariabel tager tid proportionalt med størrelsen af den værdi, som den indeholder. Oprydning af en enkelt pointervariabel tager konstant tid.

Operationer på sammensatte værdier tager tid proportional med størrelsen af de involverede værdier. Fx er tiden for at beregne a++b proportional med $\|a\| + \|b\|$. Bemærk, at $\|a\|$ kan være meget større end $|a|$, hvis listen indeholder andet end simple værdier.

De eneste undtagelser til denne regel er beregningen af en listes længde og **is**-check både for standardværdier og sumvarianter. De kan nemlig udføres i konstant tid, da vi i hvert tilfælde blot behøver at aflæse en enkelt celle.

En overførsel af en variabelparameter tager kun konstant tid, da vi blot skal kopiere et nummer på den relevante lagercelle. En overførsel af en værdiparameter tager derimod den samme tid som en tilordning.

I almindelighed kan vi angive følgende simple princip for, hvor lang tid det tager at udføre en Trine-proces

- Tidsforbruget for en proces er proportionalt med summen af størrelserne af de værdier, der beregnes under udførelsen.

Dette princip danner basis for en teori for *tidskompleksitet* af programmer, der falder uden for rammerne af denne bog.

16.4 Katekismus

△ Et fysisk lager består af en række celler, der hver indeholder et maskintal.

△ Simple værdier repræsenteres direkte af maskintal.

△ Sammensatte værdier repræsenteres af klumper af celler.

△ Simple operationer på maskintal og lageret tager konstant tid.

- △ Størrelsen af en værdi er det antal celler, som den optager.
- △ Operationer på sammensatte værdier tager tid proportionalt med størrelsen af værdien.
- △ Udførelsen af en proces tager tid proportionalt med den samlede størrelse af de værdier, der beregnes undervejs.

17 Adgang til UNIX

Dette kapitel beskriver, hvordan man p simpel vis fra et TRINE-program kan f adgang til de fleste faciliteter i UNIX.

17.1 Systemkald

Flgende vrdiudtryk

system(u_1,u_2,...,u_n)

hvor u_i'erne er vrdiudtryk, angiver et *systemkald*. Effekten er som flger: vrdierne af u_i'erne beregnes, og deres kanoniske tekstformater konkateneres. Den resulterende tekst udfres som en UNIX-kommando, prcist som hvis den var skrevet i et UNIX-vindue. Udfrelsen vil som regel resultere i nogle linjers respons fra UNIX. Disse vil blive returneret som en vrdi af typen **List**(Text).

Flgende simple program vil sledes (nsten) simulere et UNIX-vindue

```
(+ Var t: Text
   Var r: List(Text)
   read[t]
   do t ≠ "exit" →
        r := system(t)
        (+ Var i: Int
            i := 0
            do i<|r| →
                  write(r.(i), eol)
                  i := i+1
            od
        +)
        read[t]
   od
+)
```

17.2 Menu-boxen

En *menu* prsenterer brugeren for en rkke valgmuligheder angivet p separate linjer. Med brug af musen kan man derefter trffe sit valg, som programmet s kan reagere p. Man kan f adgang til menuer under X-vinduessystemet gennem flgende box

```
Box Menu
    Type Data = ≪menu parametre≫

    Proc Init [D: Data]
        ≪initialiser≫
    end Init

    Proc Title [D: Data] (t: Text)
        ≪angiv titel≫
    end Title

    Proc Geometry [D: Data] (w, h, x, y: Int)
        ≪angiv bredde, hjde og skrm-koordinater≫
    end Geometry

    Proc Line [D: Data]
        ≪tilfj en delestreg≫
    end Line

    Proc Item [D: Data] (i: Text)
        ≪tilfj en indgang≫
    end Item

    Proc Show [D: Data] → (Text)
        ≪vis menuen og returner en valgt indgang≫
    end Show
end Menu
```

Flgende eksempel benytter en simpel menu

```
(+ @"menu.tri"

    Var m: Menu'Data
    Menu'Init[m]
    Menu'Title[m]("Vlg et dyr")
    Menu'Item[m]("hund")
    Menu'Item[m]("kat")
    Menu'Line[m]
    Menu'Item[m]("mus")
    Menu'Item[m]("hamster")
    Menu'Line[m]
    Menu'Item[m]("guldfisk")
    write(Menu'Show[m])
+)
```

17.3 Prompt-boxen

En *prompt* tillader, at brugeren udfylder felterne i et delvist fortrykt skema, som programmet s kan reagere p. Man kan f adgang til prompts under X-vinduessystemet gennem flgende box

```
Box Prompt
    Type Data = ≪prompt parametre≫

    Proc Init[D: Data]
        ≪initialiser≫
    end Init

    Proc Title[D: Data](t: Text)
        ≪angiv titel≫
    end Title

    Proc Geometry[D: Data](w, h, x, y: Int)
        ≪angiv bredde, hjde og skrm-koordinater≫
    end Geometry
```

```
    Proc Item[D: Data] (p, r: Text)
        ≪tilfj en indgang p med fortrykt svar r≫
    end Item

    Proc Show[D: Data]
        ≪vis prompten≫
    end Show

    Proc Response[D: Data] (p: Text) → (Text)
        return ≪svaret for indgang p≫
    end Response
end Prompt
```

Flgende eksempel benytter en simpel prompt

```
(+ @"prompt.tri"

    Var p: Prompt'Data
    Prompt'Init[p]
    Prompt'Title[p]("Personlige oplysninger")
    Prompt'Geometry[p](350, 150, 100, 200)
    Prompt'Item[p]("Efternavn", "")
    Prompt'Item[p]("Fornavn", "")
    Prompt'Item[p]("Kursus", "dProg1")
    Prompt'Show[p]
    write(Prompt'Response[p]("Fornavn"), " ")
    write(Prompt'Response[p]("Efternavn"), " flger ")
    write(Prompt'Response[p]("Kursus"), eol)
+)
```

17.4 Choice-boxen

Et *choice* giver brugeren mulighed for at udvlge en delmngde af et givent udvalg. Man kan f adgang til choices under X-vinduessystemet gennem flgende box

```
Box Choice
    Type Data = ≪choice parametre≫

    Proc Init [D: Data]
        ≪initialiser≫
    end Init

    Proc Title [D: Data] (t: Text)
        ≪angiv titel≫
    end Title

    Proc Geometry [D: Data] (w, h, x, y: Int)
        ≪angiv bredde, hjde og skrm-koordinater≫
    end Geometry

    Proc Item [D: Data] (c: Text)
        ≪tilfj en indgang≫
    end Item

    Proc Show [D: Data]
        ≪vis choicen≫
    end Show

    Proc Chosen [D: Data] (c: Text) → (Bool)
        return ≪indgang c er valgt≫
    end Chosen
end Choice
```

Flgende eksempel benytter en simpel choice

```
(+ @"choice.tri"

    Var c: Choice'Data
    Choice'Init[c]
    Choice'Title[c]("Kurser")
    Choice'Item[c]("dProg1")
    Choice'Item[c]("Mat10")
    Choice'Item[c]("Mat11")
    Choice'Show[c]
    if Choice'Chosen[c]("dProg1") →
        write("Du flger dProg1", eol)
    fi
    if Choice'Chosen[c]("Mat10") →
        write("Du flger Mat10", eol)
    fi
    if Choice'Chosen[c]("Mat11") →
        write("Du flger Mat11", eol)
    fi
+)
```

17.5 Katekismus

△ Fra TRINE-programmer kan man udfre UNIX-kommandoer.

△ Gennem specielle boxe kan man bruge menu-, prompt- og choice-faciliteter.

A Grafik

Fra TRINE programmer kan man tegne p skrmen ved hjælp af procedurer i biblioteket `graphics.tri`. Tegningerne vil kunne ses i et specielt grafikvindue, der automatisk oprettes når TRINE programmet udføres. Man tegner i den første kvadrant af et sædvanligt koordinatsystem. Størrelsen af grafikvinduet kan justeres som ved ethvert andet X-vindue.

Tegnevinduet har til enhver tid en tilstand, der består af

- en aktuel position (et koordinatpunkt)
- en aktuel vinkel (i grader)
- en aktuel farve
- en aktuel linjebredde (fra 1 og opefter)
- en aktuel tekstfont familie, stil og strrelse

Man kan tænke på dette som en *blyant*, der har en farve og en stiftbredde, og som er placeret et bestemt sted på papiret pegende i en bestemt retning. Der er følgende grafikkommandoer

Proc clear
  ≪farv hele vinduet hvidt≫
end clear

Proc setcolor(c: Text)
  ≪st den aktuelle farve til c≫
end setcolor

Proc move(d: Int)
  ≪tegn d enheder i den aktuelle retning≫
end move

Proc moveto(x, y: Int)
  ≪tegn hen til punktet (x,y)≫
end moveto

Proc jump(d: Int)
    ≪flyt d enheder i den aktuelle retning≫
end jump

Proc jumpto(x, y: Int)
    ≪flyt hen til punktet (x,y)≫
end jumpto

Proc turn(a: Int)
    ≪drej den aktuelle vinkel a grader≫
end turn

Proc turnto(a: Int)
    ≪st den aktuelle vinkel til a grader≫
end turnto

Proc setwidth(w: Int)
    ≪st den aktuelle linjebredde til w≫
end setwidth

Proc position[x, y: Int]
    ≪st (x,y) til den aktuelle position≫
end position

Proc status[x, y, a: Int, c: Text, w: Int]
    ≪st (x,y,a,c,w) til den aktuelle tilstand≫
end status

Proc readpixel(x, y: Int) → (Text)
    return ≪farven af punktet (x,y)≫
end readpixel

Proc windowsize[w, h: Int]
    ≪st w og h til bredde og hjde af grafikvinduet≫
end windowsize

Proc fill(x_0, y_0, x_1, y_1: Int)
≪udfyld rektanglet fra (x_0,y_0) til (x_1,y_1)≫
end fill

Proc arc(x, y, r, a, d: Int)
≪tegn cirkelbuen fra a til a+d grader
med centrum (x,y) og radius r≫
end arc

Proc arcfill(x, y, r, a, d: Int)
≪udfyld cirkelbuen fra a til a+d grader
med centrum (x,y) og radius r≫
end arcfill

Proc mouse[x, y, b: Int]
≪st (x,y,b) til position og knap for sidste museklik≫
end mouse

Proc clearmouse
≪glem alle ubehandlede museklik≫
end clearmouse

Proc print(t: Text)
≪skriv teksten t≫
end print

Proc printlength[l: Int](t: Text)
≪st l til lngden af teksten t≫
end printlength

Proc setfamily(f: Text)
≪st tekstens font-familie til f≫
end setfamily

Proc setstyle(s: Text)
≪st tekstens font-stil til s≫
end setstyle

Proc setsize(s: Int)
≪st tekstens font-strrelse til s≫
end setsize

Proc showgif(f: Text)
≪vis gif-billedet i filen f p den aktuelle position≫
end showgif

Proc gifsize[w, h: Int] (f: Text)
≪st (w,h) til bredde og hjde af gif-billedet i filen f≫
end gifsize

Proc dumpgraphics(f: Text)
≪gem grafikvinduet p filen f≫
end dumpgraphics

Proc loadgraphics(f: Text)
≪hent grafikvinduet p filen f≫
end loadgraphics

Tekstfonte findes i alle kombinationer af

- familie: courier, times, helvetica, schoolbook, symbol,
- stil: normal, bold, italic, bold-italic, og
- strrelse: 8, 10, 11, 12, 14, 16, 17, 18, 20, 24, 25, 34.

P alle skrme kan man benytte farverne “black” og “white”. P farve- og grtoneskrme er der utallige andre muligheder, der kan ses med UNIX-kommandoen `showrgb`.

B Grammatik

Dette appendix beskriver syntaksen af TRINE programmer. I det følgende angiver

- **Kategori** * en følge af **Kategori**
- **Kategori** $^+$ en ikke-tom følge af **Kategori**
- **Kategori** $^{*\lambda}$ en følge af **Kategori** adskilt af kommaer
- **Kategori** $^{+\lambda}$ en ikke-tom følge af **Kategori** adskilt af kommaer
- **Kategori** $^\circ$ enten 0 eller 1 forekomster af **Kategori**

Syntaksen for et lovligt TRINE program skal genereres af følgende grammatik.

```
Trine        ::=  Program Where*
Program      ::=  System | Process | Statement | Def+

System       ::=  system Name
                      SysDef* Process+
                  end Name

Process      ::=  process Name StackSize°
                      Statement
                  end Name

StackSize    ::=  ( IntConst )

Def          ::=  TypeDef | ProcDef | BoxDef | BoxInst
SysDef       ::=  ChanDef | Def
ChanDef      ::=  chan NamesType
TypeDef      ::=  type Name = TypeExp

TypeExp      ::=  int | bool | char | real |
                  prod ( NamesType*λ ) |
```

sum (**NamesType** $^{*\lambda}$) |
list (**TypeExp**) |
pointer (**TypeExp**) |
DefName |
« **Ascii** $^{+}$ »

DefName ::= **Name** | **Name**'**Name**

NamesType ::= **Name** $^{+\lambda}$: **TypeExp**

ProcDef ::= proc **Trace** $^{\circ}$ **Name** **Formals**
    Statement
end **Name**

Trace ::= trace

Formals ::= **Var-Formals** $^{\circ}$ **Val-Formals** $^{\circ}$ **Res-Formal** $^{\circ}$
Var-Formals ::= [**NamesType** $^{*\lambda}$]
Val-Formals ::= (**NamesType** $^{*\lambda}$)
Res-Formal ::= -> (**TypeExp**) | -> [**TypeExp**]

BoxDef ::= box **Name** **Box-Formals** $^{\circ}$
    Def $^{+}$
end **Name**

Box-Formals ::= (**Name** $^{*\lambda}$)

BoxInst ::= box **Name**
    Name (**TypeExp** $^{*\lambda}$)
end **Name**

Statement ::= skip | wait | stop |
VarExp $^{+\lambda}$:= **ValExp** $^{+\lambda}$ |
VarExp :=: **VarExp** |
ProcCall |
if **Guards** fi |

```
                    do Predicate° Guards od |
                    co Co-Guards oc |
                    return Res-Actual° |
                    (+ StmDef* Statement +) |
                    Comm |
                    « Ascii+ » |
                    Statement Statement |
                    abort ( ValExp ) | abort |
                    read [ VarExp+λ ] |
                    write ( ValExp+λ ) |
                    system ( ValExp+λ ) |
                    dump [ VarExp ] ( ValExp ) |
                    load [ VarExp ] ( ValExp ) |
                    dumppage [ VarExp ] ( ValExp ) |
                    loadpage [ VarExp ] ( ValExp ) |
                    append ( ValExp , ValExp ) |
                    dispose [ VarExp ] |
                    trace TraceVars° TraceText° |
                    graphics GraphicsVars° GraphicsVals° |
                    Predicate

TraceVars       ::= [ Name+λ ]
TraceText       ::= ( ValExp )

GraphicsVars    ::= [ VarExp*λ ]
GraphicsVals    ::= ( ValExp*λ )

ProcCall        ::= DefName Actuals

Actuals         ::= Var-Actuals° Val-Actuals°
Var-Actuals     ::= [ VarExp*λ ]
Val-Actuals     ::= ( ValExp*λ )
Res-Actual      ::= VarExp | ValExp

Comm            ::= Send | Receive
Send            ::= Name !ValExp
```

```
Receive      ::=  Name ?VarExp

Guards       ::=  ValExp -> Statement Choice°
Choice       ::=  | Guards | & Guards

Co-Guards    ::=  Comm -> Statement Co-Choice°
Co-Choice    ::=  | Co-Guards

StmDef       ::=  VarDef | Def

VarDef       ::=  var NamesType

VarExp       ::=  BasicVar Dot*
BasicVar     ::=  Name | ProcCall | ref ( ValExp )
Dot          ::=  . Name | . ( ValExp+λ )

Predicate    ::=  { Ascii* }

ValExp       ::=  IntConst | BoolConst |
                  CharConst | RealConst |
                  TextConst | ProdConst |
                  SumConst | ListConst |
                  PointerConst | UndefConst |
                  VarExp |
                  value [ VarExp ] |
                  pointto [ VarExp ] |
                  readchar
                  trychar
                  ProcCall |
                  MonOp ValExp |
                  ValExp BinOp ValExp |
                  ValExp ( ValExp .. ValExp ) |
                  is ( ValExp ) |
                  is ( ValExp , Name ) |
                  Name ( ValExp | ValExp ) |
                  list ( ValExp | ValExp ) |
                  | ValExp | |
```

```
                 « Ascii+ » |
                 maxint | clock | eol |
                 SysProc1 ( ValExp ) |
                 SysProc2 ( ValExp , ValExp ) |
                 SysProc3 ( ValExp , ValExp , ValExp ) |
                 ( ValExp )

SysProc1     ::= abs | it | ti | ci | ic | bt | tb | tr | floor |
                 ceil | round | sqrt | exp | ln | sin | cos | arctan
SysProc2     ::= random | rrandom
SysProc3     ::= rt

IntConst     ::= Digit+

BoolConst    ::= false | true

CharConst    ::= ' Ascii '

RealConst    ::= Mantissa Exponent
Mantissa     ::= Digit+ Decimals°
Decimals     ::= . Digit+
Exponent     ::= E Sign° Digit+
Sign         ::= + | -

TextConst    ::= " Ascii* "

ProdConst    ::= Name ( ValExp*λ ) |
                 prod ( ValExp*λ )

SumConst     ::= Name ( Name : ValExp ) |
                 sum ( IntConst : ValExp )

ListConst    ::= Name ( ValExp*λ ) |
                 list ( ValExp*λ )

PointerConst ::= Name ( ValExp )
                 pointer ( ValExp )
```

UndefConst ::= ?-int | ?-bool | ?-char | ?-real |
?-prod | ?-sum | ?-list | ?-pointer |
?-**DefName** | # | nil

MonOp ::= - | not
BinOp ::= + | - | * | / | mod | % | ++ | or | and | **RelOp**
RelOp ::= = | <> | < | > | <= | >=

Name ::= **Letter** **AlphaNum** *
AlphaNum ::= **Letter** | **Digit**

Where ::= where « **Ascii** $^+$ » is **WhereIs**

WhereIs ::= **TypeExp** | **Statement** | **ValExp**

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Letter ::= a | b | ··· | | A | B | ··· |

Kategorien **Ascii** svarer til alle ASCII tegnene.

Et **Name** kan ikke være et af de følgende *reserverede* navne

abort	abs	and	append	arctan	bool
box	bt	ceil	chan	char	ci
clock	co	cos	dispose	do	dump
dumppage	end	eol	exp	false	fi
floor	graphics	ic	if	int	is
it	list	ln	load	loadpage	maxint
mod	nil	not	oc	od	or
pointer	pointto	proc	process	prod	random
read	readchar	real	ref	round	return
rrandom	rt	sin	skip	sqrt	stop
sum	system	tb	ti	tr	trace
true	trychar	type	value	var	wait
where	write				

For de reserverede navne skelnes der ikke mellem store og små bogstaver.

C Typografi

Programmerne i denne note er skrevet med forskellige specialtegn, der ikke findes p tastaturet. Den flgende tabel angiver en oversttelse fra disse tegn til ASCII, der skal benyttes i de TRINE programmer, man selv skriver.

Noten	ASCII
$\rightarrow$	`->`
$\leq$	`<=`
$\geq$	`>=`
$\ll$	`«`
$\gg$	`»`
$\wedge$	`and`
$\vee$	`or`
$\neg$	`not`
$\neq$	`<>`

Programmet

(+ **Var** x, y: Int
read [x, y]
if $\neg$ (x $\leq$ y) $\vee$ (y $\neq$ x) $\rightarrow$ **skip**
& true $\rightarrow$ $\ll$noget$\gg$
fi
+)

skal sledes skrives som

```
(+ var x,y: Int
   read[x,y]
   if not (x<=y) or (y<>x) -> skip
    & true -> «noget»
   fi
+)
```

D Programeksempler

Dette appendix indeholder en liste over de vigtigste programeksempler. For hvert eksempel gives dets sidenummer, dets filnavn i UNIX-systemet og en kort beskrivelse.

Side 13; `februar.tri`; beregner givet et årstal antallet af dage i februar måned.

Side 14; `potenssum.tri`; beregner en sum af potenser.

Side 15; `reciprok.tri`; udskriver den uendelige decimalrepræsentation af reciprokværdien af et indlæst heltal.

Side 15; `primfaktor.tri`; udskriver primfaktoropløsningen af et indlæst heltal.

Side 20; `veksler.tri`; udbetaler et indlæst beløb som byttepenge.

Side 23; `pi.tri`; beregner en approximation af pi.

Side 34; `listemax.tri`; finder det største element i en liste af heltal.

Side 40; `cifferblokke.tri`; finder alle cifferblokke i en tekst.

Side 41; `gennemsnit.tri`; beregner gennemsnittet af karakterer i en relation over elever. Benytter filen `folkeskole`.

Side 65; `sikker.tri`; sikker indlæsning af et heltal.

Side 68; `orddeling.tri`; indsætter delestreger i danske ord.

Side 79; `polynomie.tri`; beregner værdien af et polynomie givet dets koefficienter og et punkt.

Side 100; `euklid.tri`; udfører Euklids udvidede algoritme.

Side 109; `petri.tri`; tegner en figur. Benytter filen `figures.tri`.

Side 110; `figures.tri`; bibliotek med procedurer til tegning af figurer.

Side 123; `database.tri`; opdaterer en database af studenter. Benytter filen `rgang95`.

Side 135; `dubletter.tri`; finder dubletter blandt indlæste tal. Benytter filen `set.tri`.

Side 138; `papkasser.tri`; en box med operationer på papkasser.

Side 145; `sequence.tri`; en box med polymorfe sekvenser.

Side 147; `catalog.tri`; en box med polymorfe kataloger.

Side 160; `tree.tri`; rekursiv tegning af træer.

Side 161; `fibonacci.tri`; beregning af Fibonaccital.

Side 166; `fraktal.tri`; tegning af en fraktal kurve.

Side 169; `genkend.tri`; genkendelse af simple udtryk.

Side 172; `plante.tri`; rekursiv tegning af pæne træer.

Side 179; `typetree.tri`; rekursiv opbygning og tegning af træer.

Side 181; `opbyg.tri`; opbygning og genkendelse af simple udtryk.

Side 194; `labyrint.tri`; tegning og løsning af labyrinter. Inkluderer filen `labydef.tri`.

Side 196; `unixfile.tri`; skitse af UNIX-lignende filsystem. Benytter filen `unixdir`.

Side 208; `cyclic.tri`; en box med cykliske pointerlister.

Side 211; `iscs.tri`; en polymorf box med effektive pointersekvenser.

Side 218; `spalte.tri`; et multiproces system til produktion af avisspalter.

Side 224; `buffer.tri`; brug af en buffer til asynkron kommunikation i et multiproces system.

Side 229; `wolfpig.tri`; administration af en delt ressource i et multiproces system. Benytter filerne `wolf.fig` og `pig.fig`.

Side 241; `menu.tri`; en box der giver adgang til en X-menu.

Side 242; `prompt.tri`; en box der giver adgang til en X-prompt.

Side 243; `choice.tri`; en box der giver adgang til en X-choice.

Indeks

abort, 7
abs, 25, 26
addition, 24, 26
afprøvning, 77
    ekstern, 78
    intern, 79
akkumulatorvariabel, 51
anonym konstant, 53
append, 58
arc, 248
arcfill, 248
arctan, 27

beregningsregler, 27
bevis
    gyldighed, 87
    terminering, 97
Bool, 3, 5, 25
    disjunktion, 25
    konjunktion, 25
    negation, 25
    uligheder, 25
box, 130
    polymorf, 140
bt, 57

ceil, 26
Char, 3, 5
    uligheder, 25
choice, 243
ci, 57
clear, 246
clearmouse, 248
clock, 25
co-oc, 222
cos, 27

dekoreret program, 85
delvariabel
    af liste, 33
    af produkt, 41
    af rekursiv type, 184
    kassografi, 47
disjunktion, 25
division, 24, 26
do-od, 13, 20
dump, 56, 58
dumpgraphics, 249
dumppage, 58

eol, 57
exp, 27

filer, 58
    append, 58
    dump, 58
    dumppage, 58
    inklusion, 107
    load, 58
    loadpage, 58
fill, 248
floor, 26
fortegnsskift, 25

gensidig rekursion, 164
gifsize, 249
grafik, 246
gyldighed
    betinget, 87
    iteration, 91
    sekvens, 89
    selektion, 90
gyldigt udsagn, 85

ic, 57
if, 12
if-fi, 12, 19
indeks, 33
indhold, 4
indlæsning, 10
indskudt sætning, 10, 15
inklusion, 107
Int, 3, 5, 24
    abs, 25
    addition, 24
    clock, 25
    division, 24
    fortegnsskift, 25
    maxint, 25
    mod, 24
    multiplikation, 24
    random, 25
    subtraktion, 24
    uligheder, 25
invariant, 92
    af datatype, 135
is, 6, 44
it, 57
iteration, 13
    gyldighed, 91
    invariant, 92

jump, 247
jumpto, 247

kanal, 215
    kommunikation, 215
    nondeterminisme, 217
    synkronisering, 217
kanonisk tekstformat, 55
    grammatik, 56
konjunktion, 25
konstant, 6
konverteringer, 57
    bt, 57
    ci, 57
    ic, 57
    it, 57
    rt, 57
    tb, 57
    ti, 57
    tr, 57
korrekthed, 99

liste, 32
    af mængder, 31
    anonym, 54
    del, 35
    delvariabel, 33
    indeks, 33
    kanonisk tekstformat, 56
    konstant, 34
    længde, 33
    oversigt, 38, 47
    sammensætning, 35
    skabelon, 50
    standardværdi, 38
    Text, 39
ln, 27
load, 56, 58
loadgraphics, 249
loadpage, 58

maxint, 25
menu, 241
mod, 24
mouse, 248
move, 246
moveto, 246
multiplikation, 24, 26

navne, 255

negation, 25
nondeterminisme, 19, 21, 222
normalform, 61, 130, 188

ombytning, 8, 235
operatorassociering, 27
operatorhierarki, 27
overløb, 24, 26
overskygning, 19

parameter, 113
  værdi, 113
  variabel, 113, 124
pointer, 199
  tilstandstabel, 200
  type, 199
polybox, 140
  instantiering, 140
polymorfi, 140
position, 247
print, 248
printlength, 248
procedure, 107
  kald, 110, 115
  krop, 110, 115
  parameter, 110, 115
  rekursiv, 158
  tilstandstabel, 113
  værdi, 119
  variabel, 120
proces, 215
produkt, 40
  af mængder, 31
  anonym, 54
  delvariabel, 41
  komponent, 41
  konstant, 41
  oversigt, 43
  skabelon, 50
  standardværdi, 43
  tilstandstabel, 42
program
  dekoreret, 85
prompt, 242
pseudoreelle tal, 21, 26

random, 25
read, 10, 56
readchar, 10
readpixel, 247
Real, 21, 26
  abs, 26
  addition, 26
  arctan, 27
  ceil, 26
  cos, 27
  division, 26
  exp, 27
  floor, 26
  ln, 27
  multiplikation, 26
  round, 26
  rrandom, 27
  sin, 27
  sqrt, 27
  subtraktion, 26
ref, 199
rekursionsdybde, 163
rekursiv
  gensidigt, 164
  procedure, 158
  type, 175, 203
relationer, 72
reserverede navne, 255
return, 118–120
round, 26
rrandom, 27

rt, 57

sætning, 7
- abort, 7
- indskudt, 10, 15
- iteration, 13
- kommunikation, 222
- ombytning, 8, 235
- read, 10
- return, 118–120
- sekvens, 8
- selektion, 12
- simultan tilordning, 9
- skip, 7
- stop, 7
- tilordning, 7, 236
- ubestemt, 64
- wait, 7
- write, 10

sammenligning, 25
sekvens, 8
- gyldighed, 89

selektion, 12
- gyldighed, 90

setcolor, 246
setfamily, 248
setsize, 249
setstyle, 248
setwidth, 247
showgif, 249
sin, 27
skabelon, 50
skip, 7
sqrt, 27
standardværdi, 3, 5, 6, 23
- anonym, 54

statisk omgivelse, 152
- overskygning, 154

status, 247
stop, 7
subtraktion, 24, 26
sum, 44
- af mængder, 31
- anonym, 55
- delvariabel, 44
- is, 44
- kanonisk tekstformat, 56
- konstant, 44
- skabelon, 51
- tilstandstabel, 45

syntaks, 250
system, 215, 240

tb, 57
termineringsfunktion, 98
Text, 39
ti, 57
tilordning, 7, 236
- simultan, 9

tilstand, 3
tilstandstabel, 4
- af sum, 45
- liste, 36
- pointer, 200
- procedure, 113
- produkt, 42
- rekursiv procedure, 161
- repræsentation af, 233

tr, 57
trinvis forfinelse, 64
trychar, 10
turn, 247
turnto, 247
type, 3
- ækvivalens, 59, 130, 187
- aktuel, 141

atomar, 3
formel, 140
konstruktør, 30
list, 32
navngiven, 30
normalform, 61, 130, 188
parameter, 140
pointer, 199
prod, 40
regulær, 30
rekursiv, 175, 203
sum, 44
ubestemt, 64

udsagn, 83
gyldigt, 85
udskrivning, 10
udtryk
konstant, 6
sammensat, 6
ubestemt, 64
værdi, 5
variabel, 5
uligheder, 25
UNIX, 240

værdi, 3
sammenligning, 25
standard, 3
værdimængde, 3
af liste, 33
af produkt, 40
af sum, 44
værdiparameter, 113
værdiprocedure, 119
værdiudtryk, 5, 184
Val, 3
value, 5
implicit, 11
variabel, 3
variabelparameter, 113, 124
deling, 127
sideeffekt, 126
variabelprocedure, 120
variabeludtryk, 5, 184, 199

wait, 7
where, 64
windowsize, 247
write, 10, 56