

Om Simula 67

Ekstraforelesning for IN105, 26/10-2000

Ole-Johan Dahl

Språksituasjonen tidlig 60-tall

Programmering for “vitenskapelige beregninger” (i motsetning til “administrativ databehandling”) var dominert av:

- FORTRAN (amerikansk)
- Algol 60 (europeisk)

Av strategiske grunner måtte Simula bygge på et av disse. Valget falt på Algol. En viktig grunn til det var at Algol, i motsetning til FORTRAN, utgjør en logisk sikker plattform for resonnement: Programmeringsfeil blir enten avslørt av kompilatoren, eller av kontroller under eksekvering, eller programmet har en maskinuavhengig adferd. Andre grunner er nevnt i det følgende.

Tilordningsimperativet

I Algol heter tilordningsoperatoren $:=$. Eksempel: $n := n + 1$.

Dette betyr: regn ut verdien av uttrykket til høyre, og gi variabelen til venstre denne verdi. Merk at en variabelforekomst i høyresiden betegner variabelens verdi, mens en variabel i venstresiden betegner selve variabelen.

I FORTRAN brukes likhetsoperatoren for å betegne tilordning. Det er et misbruk av likhetstegnet, som vi trenger som relasjonsoperator i matematiske uttrykk.

Misbruket har tradisjon i USA og er dessverre overtatt og videreført i JAVA.

Merknad: Eksekvering går stort sett fra venstre mot høyre i en programtetekst. Derfor ville det ha vært aller best med høyrevendt tilordningsoperator, $n+1 =: n$.

Blokkstruktur 1

I FORTRAN er data og program i det vesentlige atskilt. Det rimer dårlig med at data og algoritmer hører sammen. Det er vanskelig å snakke om det ene uten å måtte komme inn på det andre. De blir to aspekter av noe vi kan kalle dataprosesser.

I Algol 60 er data og operasjoner på dem knyttet sammen i begrepet *blokk*:

begin blokkhode; blokkhale **end**

Hvor blokkhodet er en liste av deklarasjoner av (lokale) variable og annet, og blokkhalen er en liste av imperativer som opererer på variablene.

En blokk er selv et imperativ, som kan forekomme i blokkhalen til en omgivende blokk, nestet vikårlig dypt. Ytterste blokk er selve hovedprogrammet.

Blokkstruktur 2

I Algol er det en vesensforskjell på en programtekst og en *eksekvering* av den. Eksekveringen av hver enkelt blokk er en egen dataprosess som krever dataplass for sine lokale variable og tid for å utføre sine handlinger.

En blokkeksekvering starter med at programkontrollen passerer forbi blokkens **begin**. Da allokeres det lagerplass for de lokalt deklarerte variable. Deretter utføres blokkhalen. Når kontrollen passer blokkens **end**, forsvinner variabel-instansene, slik at lagerplassen kan brukes til andre data. Det passer med at de lokale variablene bare eksisterer og er tilgjengelige så lenge kontrollen er innenfor blokk-instansen. Variable deklarert i omgivende blokker er da også tilgjengelige.

Som vi skal se, kan det eksistere flere instanser av samme blokk samtidig. Blokkhalen er tilpasset slik at lokale variable refererer til dem som hører til *den virkende blokkinstansen*.

Prosedyrer

Prosedyrer i Algol svarer til “metoder” i JAVA. En prosedyre er en blokk, spesialisert ved at den har et prosedyrehode som angir navnet og en parameterliste. Parameterlisten kan ses som en forlengelse av blokkhodet til “prosedyre kroppen”.

Prosedyrehodet og kroppen utgjør tilsammen en prosedyre-deklarasjon. Den kan inngå i blokkhodet til en blokk av vilkårlig slag. Dette passer med Algol,s prinsipp om “ortogonalitet”: at mekanismer kan kombineres på alle måter.

Kallet på en prosedyre er som i JAVA. Det oppstår en instans av prosedyrekroppen med formalparametrene initialisert fra aktualparametrene i kallet. Virkemåten til blokk-instanser gjør at prosedyrer uten videre kan være *rekursiv*e, dvs de kan inneholde kall på seg selv. Derved kan det oppstå en “stakk” av instanser av samme prosedyre med hver sine lokale variable.

Eksempel på skjelett av Algol-program

```

begin real procedure potens(real x, integer n);           /:n >= 1:/
begin real y, z;                                         /:variable lokale i potens:/
  if n=1 then y := x                                     /:n=1 & y=x:/
  else begin z := potens(x, n/2)                         /:z = xn/2:/
    if even(n) then y := z * z
    else y := z * z * x end                               /:y = xn:/
  end av potens;                                           /:y = xn:/
  real v, w; inint(v)                                   /:anta v = 2:/
  w := potens(v, 5); outreal(w, 10)                     /:gir w = 32:/
end av hovedprogrammet

```

<i>h.prog</i>	<i>w</i>	<i>w</i> = 32		
<i>potens</i>	<i>x</i> = 2 <i>n</i> = 5 <i>y</i>	<i>z</i>	<i>x</i> = 2 <i>n</i> = 5 <i>y</i> = 32	<i>z</i> = 4
<i>potens</i>	<i>x</i> = 2 <i>n</i> = 2 <i>y</i>	<i>z</i>	<i>x</i> = 2 <i>n</i> = 2 <i>y</i> = 4	<i>z</i> = 2
<i>potens</i>	<i>x</i> = 2 <i>n</i> = 1 <i>y</i> = 2	<i>z</i>		

Eksempel på løkke-invariant 1

Vi skal beregne eksponentialfunksjonen etter vanlig rekkeutvikling:

$$e^x = \sum_0^\infty x^i / i! = 1 + x + x^2 / 2 + x^3 / 6 + \dots$$

Vi har behov for tre arbeidsvariable: **integer** n (teller), **real** y (sum så langt) og **real** z (nåværende ledd). Det fins mange muligheter for initialisering og oppdatering av variablene:

Initialiser: $n := 0$ eller 1, $y := 0$ eller 1 eller $1 + x$, $z := 1$ eller x .

Iterer: $n := n + 1$, $y := y + z$ eller $y + z * x / n$ eller $y + z * x / (n + 1)$,
 $z := z * x / n$ eller $z * x / (n + 1)$.

Oppdateringene kan sekvenseres på 6 måter. Dette gir $2 * 3 * 2 * 3 * 2 * 2 * 6 = 864$ mulige sammensetninger, hvorav noen er plausible, men få er korrekte.

Eksempel på løkke-invariant 2

Det er lurt å satse på en *invariant* for iterasjonsløkken, en påstand som skal holde hver gang f.eks i toppen av løkken. Vi kan velge:

$$y = \sum_{i=0}^n x^i / i! \text{ \& } z = x^n / n!$$

som sier at z er siste ledd i delsummen så langt. Velger vi nå å la n starte med verdien 1, får vi nesten uten å tenke følgende optimale resultat:

Initialiser: $n := 1$; $y := 1 + x$; $z := x$

Iterer: $n := n + 1$; $z := z * x / n$; $y := y + z$

Iterasjonen skal fortsette inntil en oppgitt relativ nøyaktighet er oppnådd. Invarianten bør være skrevet in som kommentar i programteksten. Da blir det lett å forstå programmet, og kontrollere at det er korrekt.

Klasser

En klasse i Simula 67 er ny type spesialisert blokk. Deklasasjonen er syntaktisk analog med en prosedyredeklarasjon:

class navn(parameterliste); **begin** blokkhode; blokkhale **end**

Et kall på en klasse gir opphav til en blokk-instans som kalles et *objekt* av klassen. Variable og annet deklart på ytterste bloknivå i et objekt kalles objektets “attributter”. Et objekt kan refereres av “pekere” kvalifisert med navnet på klassen, og det vil forbli i systemet så lenge det er referert av tilgjengelige pekere. Objektets attributter er (normalt) tilgjengelige utenfra, f.eks ved “dot-notasjon” som i JAVA (hvor A er attributt til klassen C):

ref(C) X ; ...; X : – **new** C (aktualparametre); ...; $X.A..$

Kvasiparallelitet

Objekter i Simula kan eksekvere “kvasiparallelt”. Dvs et objekt kan midlertidig avbryte sin virksomhet, og gjenoppta den på et senere tidspunkt etter at andre objekter har eksekvert. Dette er nyttig i flere sammenheng, men særlig i forbindelse med simulering av systemer med prosesser som går i parallell.

Primitive imperativer er tilgjengelige for kvasiparallell sekvensering, bl.a *resume*. Når et objekt X sier *resume*(Y) går kontrollen over til Y . X vil gjenoppta virksomheten etter *resume*-stedet, når et objekt sier *resume*(X).

Subklasser

Vi kan deklarere en subklasse D til en klasse C ved å bruke C som “prefiks”, C **class** D . Dette svarer til å si D **extends** C i JAVA. Vi sier at subklassen “arver” attributtene til prefiks-klassen (samt parametre og blokkhale).

La C har attributtet A og D dessuten attributtet B , og la **ref**(C) X peke på et objekt av klassen D . X kan da bare gi aksess til objektets A . For å få tilgang til attributtet B kan pekeren “omtolkes”, X **qua** $D.B$.

Simula har også en mekanisme som tester klassetilhørigheten eksplisitt.

inspect X **when** D **do** inspection blokk

evt med flere alternativer. En inspection blokk har objektets attributter som lokale.

Gjenbruk

Om en klasse ikke refererer til nonlokale størrelser, kan den kompileres separat og lagres for bruk i senere program, evt som prefiks.

Programutvikling basert på separat komplilering av klasser er et eksempel på “bottom-up” utvikling. For å gjøre Simula bedre egnet til “top-down” utvikling, ble innført begrepet “virtuell prosedyre”.

En virtuell prosedyre er et prosedyre-attributt som kan redeklarerer i en subklasse, slik at redefinisjonen gjelder på alle prefiksnivåer. Hvis et objekt har en virtuell prosedyre P , vil altså et kall på P overalt i objektet, eller utenfra, referere til den P -prosedyren som er deklartert på innerste prefiksnivå i objektet.

Derved blir en virtuell prosedyre *utskiftbar*, slik at den aktuelle definisjonen kan utsettes til en senere subklasse. Denne form for binding er standard i mange senere objekt.orienterte språk.

Lokale klasser

Prinsippet om ortogonalitet tilsier at klassesdeklarasjoner bør kunne ligge i hodet til blokker av villkårlig type, f.eks lokalt i en ytre klasse. Denne kan få karakteren av et “applikasjonsspråk” som tilbyr spesialmekanismer i tillegg til standardbegrepene i Simula.

En predefinert klasse *SIMULATION* er et godt eksempel på et slikt applikasjonsspråk. Det har mekanismer for kvasiparallell ekskvering i en simulert tid.

Det vil sjelden være behov for mer enn én instans av et applikasjonsspråk i et program. Derfor er det praktisk å bruke en slik klasse som prefiks for en vanlig blokk, som derved arver begrepene definert i klassen. En blokk med prefikset *SIMULATION* utgjør en simuleringsmodell som selv kan innlemmes i et større program.

Simulering (fragment)

```
class SIMULATION;
begin class process;
    begin real evtime; ref(process) nextev : – none;
        ..... end of process;
    ref(process) current;
    real procedure time; time := current.evtime;
    procedure hold(real DeltaT);
    inspect current do
    begin evtime := evtime + DeltaT;
        if evtime >= nextev.evtime then
            begin ref(process) X : – current; current : – X.nextev;
                flytt X til riktig sted i tidslisten; resume(current) end
            end of hold;
        ..... end of SIMULATION
```

Fragment av simuleringsmodell

```
SIMULATION begin
  process class Car;
  begin real  $X0, T0, V$ ;
    real procedure  $X$ ;  $X := X0 + V * (time - T0)$ 
    procedure update(real  $Vnew$ );
      begin  $X0 := X$ ;  $T0 := time$ ;  $V := Vnew$  end;
      .....; update(80); hold(reisetid); .....
    end of Car;
  process class Police;
    begin ref(process)  $P$ ;
      .....;
    inspect  $P$  when Car do
      if  $X < bygrense$  &  $V > 50$  then
        begin update(50); bøtelegg  $P$  end
      .....end of Police;
    .....end of SIMULATION blokk
```

Modularisering ved prosedyrer

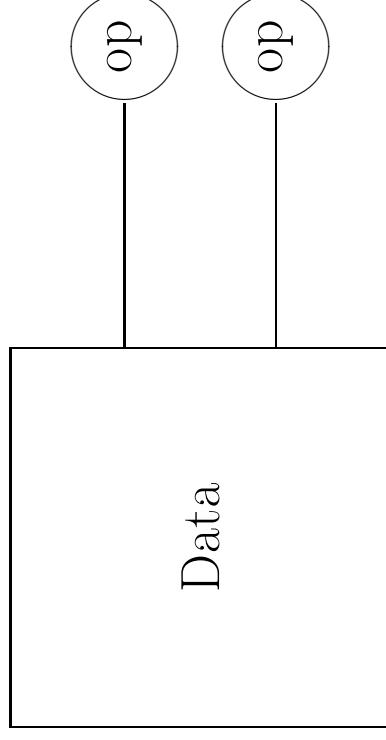
De første decenni er av datakulturen var et “program” beskrivelsen av en beregningsprosess som kommuniserte med omverdenen gjennom data inn og ut:



Tilsvarende fikk typiske programmoduler prosedyreaktig form.

Modularisering ved objektklasser

Ved konstruksjon av simuleringsmodeller oppstod det et behov for å avbilde “objekter” i et virkelig system på strukturer i en programutførelse. For å oppnå strukturlikhet var det nyttig å innføre programmoduler “duale” til prosedyrer:



altså klasser av lagrede datastrukturer som kommuniserer med omgivelsene ved operasjoner deklarert i objektet.

Programsystemer i dag

Ved utvikling av større programsystemer, f.eks interaktive, er ikke problemene svært forskjellig fra dem man møter ved simulering. Det er også her nyttig å oppnå strukturlikhet med en virkelighet som inneholder databaser, skjermbilder, interaktive brukere, osv.

Dette kan kanskje bidra til å forklare successen som Simula 67 har hatt som inspirasjon for moderne programmeringsparadigmer.

Merk at objekter i Simula 67 egentlig er potensielle korutiner. Dermed kan det med en viss rett sies at Simula 67 peker videre mot modularisering ved (parallele) prosesser.

Objektorientering

Paradigmet inneholder tradisjonelt følgende komponenter, som med ett unntak var til stede i Simula 67 som definert opprinnelig:

- Imperativ (prosedyrell) programmering
- Oppdaterbare data-objekter
- Objekter har lokale operatorer (og kan ha egne handlinger?)
- Objekter hører til deklarerte klasser
- Subklasser arver egenskapene til superklassen
- Eksplisitt bruk av (kvalifiserte) objektpekere
- Aksess til lokale størrelser i andre objekter
- Mekanismer for skjuling av lokale størrelser

Unntaket er skjuling. Mekanismer for det ble introdusert senere på initiativ av Jacob Palme (FOA, Stockholm).