

# Fra Pascal til Simula

Mads Rosendahl

25. august 1995

Pascal og Simula tilhører begge Algol-familien af programmeringssprog. De blev konstrueret med få års mellemrum og dette gør at der på langt de fleste områder er større ligheder end forskelle mellem sprogene.

Denne note er et supplement og en lsevejledning til noterne *H B Hansen: Simula, Et objektorienteret programmeringssprog*. Vejledningen henviser sig til studerende som i forvejen har kendskab til programmeringssproget Pascal og målet er at tydeliggøre forskelle og ligheder mellem de to sprog. Noter kan også lses af studerende uden det store kendskab til Pascal, som gerne vil kunne lse algoritmer udtrykt i Pascal (f.eks. fra Sedgewicks bog). Det er tanken at disse noter skal læses sideløbende med H B Hansens noter jfr. planen sidst i disse noter. Forhåbentlig kan noterne også bruges som opslagsværk hvis man skal have svar på konkrete spørgsmål vedrørende Simula.

# Indholdsfortegnelse

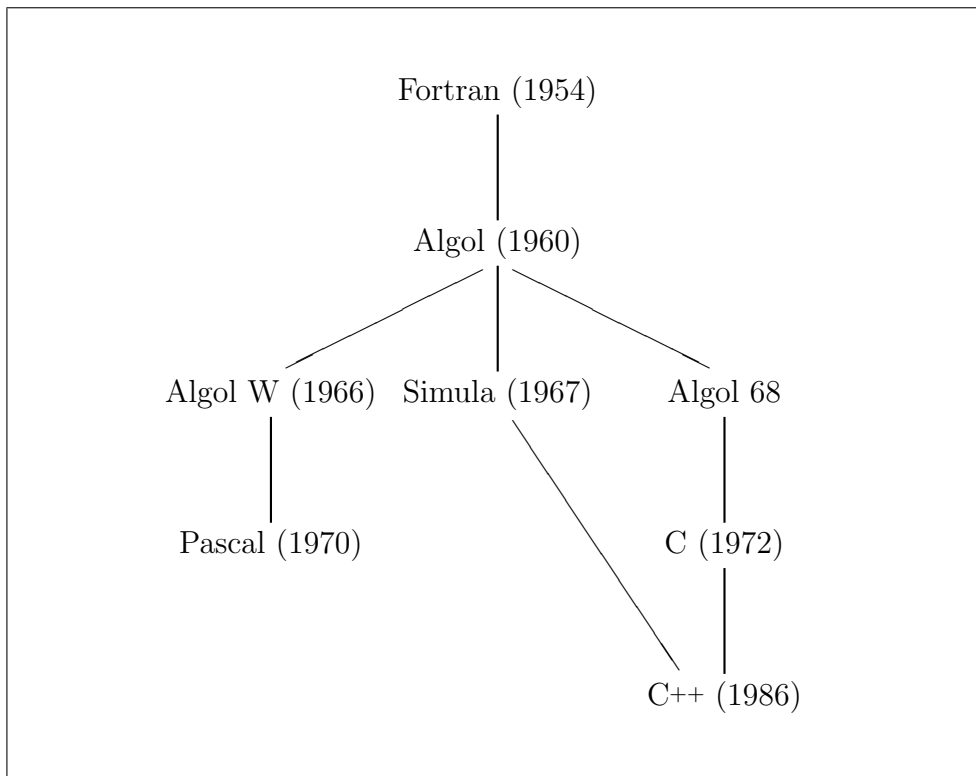
|   |           |
|---|-----------|
| <b>1 Lidt Historie</b>                    | <b>3</b>  |
| 1.1 Fra Algol til Pascal . . . . .        | 4         |
| 1.2 Fra Algol til Simula . . . . .        | 5         |
| <b>2 Pascal vs. Simula</b>                | <b>7</b>  |
| 2.1 Programmer, erkæringer, mm. . . . .   | 7         |
| 2.2 Kernelsætninger . . . . .             | 10        |
| 2.3 Datatyper . . . . .                   | 14        |
| 2.4 Indlæsning og udskrivning . . . . .   | 21        |
| 2.5 Procedurer og parametre . . . . .     | 24        |
| 2.6 Poster - records og klasser . . . . . | 34        |
| 2.7 Tekststrenger . . . . .               | 38        |
| 2.8 Konklusion . . . . .                  | 45        |
| <b>3 Litteraturliste</b>                  | <b>47</b> |

# 1 Lidt Historie

Et af de største enkelte fremskridt i programmeringssprogenes (og programmeringens) historie var sproget Algol. Sproget blev designet i årene 1958-1960 og brød afgørende med tidligere assembler og maskinkode-nære sprog. Programmer skulle ikke længere være en mystisk hemmelig kode fra den indviede programmør til den store komplekse datamaskine. Programmer skulle nu være måder at udtrykke algoritmer på så også andre programmører kunne læse og forstå dem. Sproget satte oversætterkonstruktører på en lang række nye opgaver og hovedparten af de teknikker, der nu bruges i oversættere, blev først brugt i Algol-oversættere.

Efterhånden som man fik erfaring med at bruge sproget Algol viste der sig naturligvis uhensigtsmæssigheder og småproblemer med sproget. Hvor der måske nok var enighed om problemerne var der ikke enighed om hvordan de skulle løses og slutningen af 60'erne og starten af 70'erne så da også fremkomsten af en lang række Algol-lignende sprog. Hertil hører Simula og Pascal, men også sprog som C, Algol 68, Algol W, Euler og senere C++, Ada, Modula 2, Oberon, samt mange flere. Disse sprog har forsøgt at forbedre Algol i forskellige retninger. Simula er i store træk en direkte udvidelse af Algol idet det stort set indeholder Algol som en delmængde. Derudover er det udvidet med faciliteter til objektorienteret programmering. Pascal er et forsøg på at forenkle Algol med simple sætningskonstruktioner, men til gengæld et større udvalg af datatyper. Sproget C har, ligesom Pascal, stort set samme mål som Algol, men man har forsøgt at gøre sproget mere maskinorienteret ved en tydeligere sammenhæng mellem variable og lageradresser. Nedenfor findes en lille del af det ganske komplicerede stamtræ for programmeringssprog.

Når vi skal se på forskelle og ligheder mellem Pascal og Simula er der altså to aspekter at betragte. Det ene er hvilke forenklinger og ændringer der er sket fra Algol (som delmængde af Simula) til Pascal og det andet er de objektorienterede faciliteter i Simula. I disse noter vil vi særligt se på det første aspekt. En mere historisk korrekt overskrift til disse noter vil således måske være "Fra Algol til Pascal", men vi skal dog også se et par enkelte ændringer i Simula i forhold til Algol.



## 1.1 Fra Algol til Pascal

Hvis man i få ord skal opsumere hvad der blev ændret med Pascal i forhold til Algol kan man sige at der er enklere og mere strukturerede kontrolsætninger, men flere typekonstruktioner. Omvendt betyder det så at når vi skal se på Simula er der mere komplicerede former for kontrolsætninger men til gengæld er der ikke alle de typekonstruktioner vi kender fra Pascal. Begge dele er udtryk for ændringer i programmeringsmetodikken op gennem 60'erne.

**“Goto-sætninger er skadelig”.** Ved designet af Algol opererede man med mange forslag til kontrolstrukturer. De fleste af disse blev indkorporeret i sproget og resultatet blev at algoritmer kunne udtrykkes på flere forskellige måder ved brug af sprogets kontrolsætninger. Ironisk nok viste det sig senere at mange programmører nøjedes med **if**- og **goto**-sætninger. I et berømt læserbrev fra 1968 med titlen “The goto statement considered harmful” advokerede E W Dijkstra mod **goto**-sætningen og hævdede at programmers læsbarhed er omvendt proportional med hyp-

pigheden af **goto**-sætninger. I stedet bør man bruge **if**-, **while**- og **for**-sætninger som gerne tydeligere skulle udtrykke hvad programmet skulle gøre. En sådan programmeringsstil kaldes for struktureret programmering (modsat spaghetti-programmering) og dette fik afgørende betydning i designet af Pascal.

**Typebegrebet.** Hvor typebegrebet i 50'erne mest blev opfattet som en struktur på det lager et program havde adgang til, har man siden lagt mere vægt på typer som modeller af virkelige eller tænkte objekter. De nye begreber, som blev introduceret i Simula, var med til at give mere vægt til brugen af abstrakte datatyper og at opfatte typer som modelleringsværktøjer. Med denne udvikling blev der i Pascal indføjet en række typekonstruktioner så man kunne give en mere detaljeret repræsentation af komplekse strukturer. En anden konsekvens af typebegrebet er at oversættere i højere grad kan give advarsler eller fejlmeddelelser hvis man bruger data i lageret som om det havde en anden struktur end da det blev gemt. I Pascal er dette næsten opnået, omend der dog er et par huller i sikkerhedsnettet ved variant-poster og procedurer-som-parametre.

**Om oversætterstruktur.** En af de grundlæggende ideer i definitionen af Pascal var at forenkle sproget ved at fjerne ting som en programmør let selv kunne konstruere ud fra andet i sproget. Hvor f.eks. Algol (og Simula) har sinus, cosinus og tangens funktioner har Pascal droppet tangens funktionen fordi en programmør let selv kan definere den ud fra sinus og cosinus. At gøre sproget enklere havde også det mål at gøre oversættelsen enklere. I Pascal er det således muligt at skrive en oversætter som kun behøver at læse programmet igennem én gang. En Simula eller Algol oversætter vil normalt bestå af 6-8 passager af programmet.

## 1.2 Fra Algol til Simula

Ved konstruktionen af Simula var det først og fremmest et mål at konstruere et sprog hvor man kunne simulere objekter - det der senere er blevet kaldt objekt-orienteret programmering. Man valgte så at realisere disse ideer som en udvidelse af et allerede kendt sprog - nemlig Algol. Den samme taktik er blevet brugt adskillige gange senere - f.eks. med Bjarne Stoustrups tilføjelser af objekt-orienterede faciliteter til det

allerede kendte sprog C i form af sproget C++. Fordelen ved denne tilgangsvinkel er at det er lettere at forklare nye brugere hvad der er specielt ved sproget. Ulempen er at denne baglæns-kompatibilitet gør sprogene mere klodsede end nødvendigt. Et ekstremt tilfælde i den retning er instruktionssættet til 486-chippen.

Ved konstruktionen af Simula valgte man således at basere sig på Algol, men som nævnt foretog man visse rettelser af småfejl og uhensigtsmæssigheder. Især er mulighederne for parameteroverførsel ændret og der er lidt strammere regler for brug af etiketter. En af de lidt mere pudsige ændringer er at der i Simula er tilføjet en **while**-sætning. Overraskende nok er det sket med inspiration fra Pascal. Det kan lyde underligt eftersom Simula er et ældre sprog end Pascal. Simulas fædre (Kristen Nygaard og Ole-Johan Dahl) havde imidlertid en snak med Pascal designer (Niklaus Wirth) i starten af 1968 hvor det endelige design for Simula ellers lå færdigt og man arbejdede med at skrive oversætteren. På det tidspunkt var Pascal så småt ved at tage form på tegnebrættet og **while**-sætningen blev da set som en klar forbedring over de mere komplicerede former for **for**-sætninger i Algol. På den måde kom **while**-sætningen med i Simula i allersidste øjeblik.

## 2 Pascal vs. Simula

I dette afsnit vil vi se mere detaljeret på forskellene mellem Pascal og Simula. Afsnittet vil blive opdelt i en række punkter hvor vi vil se på de enkelte dele af et programmeringssprog.

### 2.1 Programmer, erkæringer, mm.

Når man umiddelbart ser et lille Simula program er der to væsentlige forskelle fra et tilsvarende Pascal program. Det ene er at variable erklæres efter **begin**-nøgleordet i en blok, og det andet at typenavnet står før variabelnavne i variabelerklæringer.

Som eksempel skulle disse to programmer gerne svare til hinanden.

#### Pascal

```
var x,y : integer;  
begin  
  y := 1;  
  x := y+3  
end.
```

#### Simula

```
begin  
  integer x,y;  
  y := 1;  
  x := y+3  
end
```

Et Simula program består altså af en blok med variabelerklæringer først, og derefter nogle sætninger. Modsat Pascal kan man i Simula godt have variabelerklæringer inde i sammensatte sætninger (som derved bliver til blokke). Man kan altså godt inde i f.eks. en **if**-, **while**- eller **for**-sætning have en **begin-end** blok med lokale variable. Disse variable er så lokale inde i den **begin-end**-blok hvori de er erklæret - på samme måde som med lokale variable i en Pascal-procedure.

Ser man nøje efter i programmerne kan man se yderligere en meget lille forskel: Der er ikke noget afsluttende punktum i Simula programmet. Vi kunne have tilføjet et programhoved til Pascal-programmet, men i de fleste Pascal-versioner er det faktisk valgfrit - så vi har udeladt det her.

Når programmet kører er der den forskel mellem Pascal og Simula programmet at Simula-variable automatisk initialiseres til 0, mens Pascals variable til start har en udefineret værdi.

Alt ialt er det jo ikke just en kæmpe forskel mellem de to programmer, og det er jo blot udtryk for at de to sprog faktisk ikke er særlig forskellige.

**Kommentarer.** I Simula er der tre forskellige måder at indsætte kommentarer i et program

- end-kommentarer. Efter et **end** (altså et **begin-end**) ignoreres alt indtil noget som legalt kan efterfølge et **end**. Dvs. alt indtil et semikolon eller et **end, else, when, otherwise**. Det er en god ide at bruge denne facilitet til at angive hvad et **end** afslutter - f.eks. hvilken procedure eller hvilken **while**-sætning. Med lange programmer er det jo ikke altid let at overskue.
- kommentar-sætninger. Tekst der starter med et udråbstegn (!) og slutter med et semikolon opfattes som en kommentar og ignoreres af oversætteren. Det reserverede ord "**comment**" kan bruges i stedet for et udråbstegn til at starte denne type kommentar.
- procent-linier. Linier som starter med et procent-tegn og et blank-tegn ignoreres. Procent-linier kan også bruges til at give oversætterdirektiver.

**Tildelingssætninger og udtryk.** Som vi så ovenfor ligner tildelingssætningerne hinanden i de to sprog. I Simula er der desuden mulighed for at have en multipel tildelingssætning så hvis f.eks. variablerne "x" og "y" begge skal sættes til værdien 27 kan det gøres ved at skrive

```
x := y := 27;
```

Man kan også have endnu flere tildelinger i en sådan tildeling, og det kan læses som at tildelingssætningen har den værdi som variabelen bliver tildelt. I dette tilfælde sættes x til værdien af "y := 27" som har værdien 27. Man må dog kun bruge tildelingssætninger som udtryk i denne sammenhæng (modsat sproget C).

Udtryk (f.eks. i tildelingssætninger) i de to sprog ligner hinanden til forveksling. Der er dog en væsentlig konstruktion i Simula man ikke kender i Pascal, og det er det betingede udtryk: Et **if-then-else**-udtryk

Pascal og Simula

Simula

```
if y = 0 then x := 10 else x := 20      x := if y = 0 then 10 else 20
```

I Pascal er udregningsrækkefølgen af udtryk uspecificeret. I Simula foreskrives en venstre-til-højre udregningsrækkefølge af udtryk - med mindre dele skal overspringes pga. **if-then-else** udtryk.

**Erklæringer.** Vi har allerede set en variabelerklæring i Simula. Man kan også have konstanterklæringer ved efter navnet at skrive lig-med og en værdi.

### Pascal

```
const x = 20;
var y : integer;
begin
  y := x+10;
end.
```

### Simula

```
begin
  integer x = 20;
  integer y;
  y := x+10;
end
```

I Simula gælder en erklæring fra starten af den blok hvori den er defineret. Det gør det bl.a. unødvendigt at lave **forward**-erklæringer af procedurer som det kendes fra Pascal. Værdien i en konstanterklæring udregnes umiddelbart først der hvor den optræder tekstuel i blokken. I Pascal er der lidt mere uklarhed om dette: I Wirths standard for Pascal foreskrives det at konstanter er erklæret i hele den blok de optræder i men i de fleste implementationer gælder erklæringen kun fra den tekstuel optræder i blokken.

**Udefineret.** En af de lidt mere filosofiske forskelle mellem de to sprog er at Pascal har ladet en lang række forhold i sproget være “udefineret” eller “uspecificeret”, hvorimod mest muligt i Simula er underlagt faste regler. Således er variable i Simula initialiserede efter erklæring, hvorimod de har udefinerede værdier i Pascal. “Udefineret” betyder ikke at det er ulovligt at bruge sådanne værdier, men blot at oversætterkonstruktøren kan vælge at give sådanne variable hvilken værdi han måtte ønske. Tilsvarende foreskriver Simula at udtryk altid udregnes fra venstre til højre i den rækkefølge de forekommer, hvorimod noget tilsvarende ikke er fastlagt i Pascal. For programmøren er Simulas tilgangsvinkel lettere at gennemskue, men til gengæld giver det væsentligt mindre mulighed for at lave optimeringer under oversættelsen. Det skal dog bemærkes at implementationer af Pascal godt kan fastlægge regler for visse af disse “udefinerede” forhold uden at være i modstrid med Standard Pascal.

**Blokke og sammensatte sætninger.** Pascal og Simula er begge såkaldt blok-strukturerede programmeringssprog. Blokke er de grænser - eller virkefelter - inden for hvilke man kan have erklæringer. I Pascal giver procedurerne denne blok-strukturering. I Simula er det **begin-end**

konstruktionerne, og hertil hører også kroppe til procedurer. Det skyldes at man i Simula kan have erklæringer i alle sammensatte sætninger. I Simula kan man derfor stort set sætte lighedstegn mellem blokke (*blocks*) og sammensatte sætninger (*compound statement*). Det er stort set det samme, men der er en lille forskel som kun er interessant hvis man bruger etiketter og **goto**-sætninger.

Som hovedregel gælder i Simula at en blok er en **begin-end** konstruktion med erklæringer, hvorimod en sammensat sætning ikke har det. Den eneste grund til at skelne mellem dem er at etiketter ikke skal erklæres i Simula, og man vil gerne sikre sig mod at kunne hoppe ind i konstruktioner, hvor det vil have uheldige effekter. Man må således ikke hoppe ind i en **begin-end** hvor der er variable erklæret, for når man så når **end** vil der blive de-allokeret nogle variable som aldrig blev erklæret. At sikre sig mod det sker ved at sige at etiketter er erklæret i den blok (og altså ikke nødvendigvis den sammensatte sætning) den optræder i. Det betyder så at etiketten ikke er erklæret (og dermed kendt) uden for blokken og at man derfor ikke kan hoppe til den. Derimod kan en etikette godt bruges uden for en sammensat sætning hvori den optræder.

I Pascal er det også ulovligt at hoppe ind i en blok (dvs en procedure), men der skal etiketten erklæres i den procedure (altså blok) hvori den optræder. Det gør altså at etiketten ikke er kendt udenfor og at man derfor ikke kan hoppe ind i en procedure. I Simula er der desuden nogle ekstra regler som som siger at sætninger eller **begin-end** konstruktioner i visse sammenhænge opfattes som blokke selvom de ikke starter med erklæringer. Det drejer sig om sætningen i en **for**-sætning og kroppen til en procedure. Alt sammen fordi man ikke må hoppe ind i sådanne konstruktioner. I Pascal er det blot udefineret (og dermed bestemt ikke tilrådeligt), men ikke ulovligt at hoppe ind i en **for**-sætning.

[Ydreligere om *begin-end* og erklæringer i *HBH*: pp 55-57, 127-132]

## 2.2 Kontrolsætninger

I Simulas udvalg af kontrolsætninger spiller **goto**-sætningen en central rolle. Det skyldes nok at sproget stammer fra en periode hvor **goto**-sætningen endnu ikke blev betragtet som "skadelig". Man kan programmere fornuftigt med **goto**-sætninger, og der er programmer som bliver lidt klodsede uden brug af **goto**-sætningen - men det omvendte er så sandelig også tilfældet.

**If, while og for.** De tre vigtigste kontrolsætninger i de to sprog er **if**-, **while**- og **for**-sætningerne. Af disse er **while**-sætningen helt ens i de to sprog (og som vi nævnte før er det ikke så overraskende). For **if**-sætningen er der den lille krølle at sætningen i **then**-grenen ikke må være en betinget sætning i Simula. I Pascal er denne restriktion lempet lidt så der i **then**-grenen af en **if-then-else** sætning gerne må stå en **if-then-else**-sætning, men ikke en **if-then**-sætning.

**for**-sætningen skal have en ekstra **step** del i et Simula program, men kan ellers bruges helt som i Pascal.

### Pascal

```
var x,y : integer;
begin
  y := 1;
  for x := 1 to 10 do
    y := y*x
  end.
```

### Simula

```
begin
  integer x,y;
  y := 1;
  for x := 1 step 1 until 10 do
    y := y*x
  end
```

**for**-sætningen i Simula er dog meget mere generel end dette. Efter kolonlignemed og før **do** kan der stå en liste af såkaldte elementer. En mulighed er **step-until**, en anden mulighed er et **while**-element, og endelig kan man også have en liste af udtryk. Eksemplet ovenfra kan således også skrives som følger.

### Simula

```
begin
  integer x,y;
  y := 1;
  for x := 1,1+1,3,2*2,5,
    6,7,2*2*2,3*3,5+5
  do y := y*x
end
```

### Simula

```
begin
  integer x,y;
  y := 1;
  for x := 1, x+1 while x <= 10
  do y := y*x
end
```

Disse tre muligheder kan også blandes sammen med sikker ulæselighed som følge. Føler man sig fristet til at bruge **while**-elementer i en **for**-sætning, bør man nok overveje at bruge en **while**-sætning i stedet. En **for**-sætning med en liste af udtryk kan nok forsvares, hvis listen ikke er for lang og udtrykkene ikke er for komplicerede.

[Yderligere om **if**- og **while**-sætningen i HBH: pp 40 og 29-30]

[Yderligere om **for**-sætningen i HBH: pp 28, 31-32, 130-131]

**Case, goto og switch.** Simula har ikke en kontrol-sætning svarende til **case**-sætningen i Pascal. Til gengæld er det nemmere at bruge **goto**-sætninger, og man har mulighed for at oprette, hvad der svare til en tabel af etiketter. I Simula behøver man ikke erklære etiketter før de bruges. En **switch**-erklæring definerer en tabel af etiketter som indiceres med første etikette som nummer 1. Det skal bemærkes at man ikke kan hoppe ind i **for**-sætninger og i blokke med lokale variable.

At simulere en **case**-sætning ved hjælp af **goto**-sætninger bliver sjældent pænt og man bør nok overveje om det ikke vil være bedre blot at bruge en stribe **if**-sætninger.

### Pascal

```
var x : integer;
begin
  { indlæs x }
  case x of
    1 : { det var et 1-tal };
    2 : { det var et 2-tal };
    3 : { det var et 3-tal };
  end
end.
```

### Simula

```
begin
  switch valg := one,two,three;
  integer x;
  ! indlæs x ;
  goto valg(x);
  one: ! et 1-tal ; goto slut;
  two: ! et 2-tal ; goto slut;
  three: ! et 3-tal ; goto slut;
  slut:
end
```

Hvis indgangene i **case**-sætningen i stedet havde været 2, 3 og 4 kunne man sige “**goto** valg(x-1)” eftersom **switch**-tabellen indiceres fra 1. Havde indgangene i stedet været 7, 9 og 13 kunne man måske prøve med “**goto** valg((x-5)//2-1//(14-x))”, men det kunne også være man skulle overveje en anden sprog-konstruktion!

Efter **goto** og i listen til en **switch**-erklæring skal der stå såkaldte etikette-udtryk (*designational expressions*), altså udtryk hvis værdi er en etikette. Et etikette-udtryk kan være en etikette, en switch-indicering eller et **if**-udtryk. Udtrykkene i en **switch**-erklæring udregnes først når de bruges, og de udregnes hver gang de bruges. En **switch**-erklæring kan derfor læses som en funktion, hvis krop er en lille **case**-sætning. Dette gør det muligt at lave rekursivt definerede **switch**-erklæringer. Hvis man dertil lægger at der i etikette-udtryk kan indgå funktionskald med side-effekter - ja så bliver det for alvor uhyggeligt at programmere.

[Yderligere om **goto**-sætningen i HBH: pp 43-44, 129-130]

**Repeat-sætningen.** Simula har ikke en kontrol-sætning svarende til **repeat**-sætningen fra Pascal. Mere generelt er der en ikke helt ualmindelig situation, hvor både Pascal og Simula kommer til kort. Det sker hvor man f.eks. har en indlæsning, med efterfølgende behandling, som skal gentages indtil en eller anden stopklods nåes under indlæsningen. Situationen kan f.eks. skitseres således i Pascal

## Pascal

```
var done : boolean;
begin
  done := false;
  while not done do begin
    { indlæs }
    if {stopklods nået} then
      done := true
    else {behandl}
  end
end.
```

Dette program skal nok gøre hvad man gerne vil have det til, men er ikke så elegant idet man ikke fra betingelsen i **while**-sætningen kan se hvad der skal til før løkken stopper. Man må i stedet se nøjere på indmaden i løkken for at se hvor variabelen *done* ændres. Situationen har den normale brug af **while**- og **repeat**-sætninger som specialtilfælde. Er der kun brug for en indlæsning kan man bruge en **repeat**-sætning, og er der kun brug for behandling kan man bruge en **while**-sætning.

Eksemplet ovenfor er nok et af de få tilfælde hvor det er “pænt” at bruge **goto**. I Simula kan man således løse det samme problem således:

## Simula

```
begin
  while true do begin
    ! indlæs ;
    if !stopklods nået; then goto stopklods;
    ! behandl ;
  end;
  stopklods:
end
```

[Yderligere om Pascals **repeat**-sætning i HBH: p 32]

## 2.3 Datatyper

Umiddelbart har Pascal et rigere udvalg af typekonstruktioner end Simula. Vi skal dog senere se at det nok snarere forholder sig lige omvendt, for med objekt-orienterede faciliteter kan man simulere eller repræsentere snart sagt hvad som helst. Her skal vi dog se på Simulas umiddelbart tilgængelige primitive datatyper og tilhørende operationer og standardfunktioner. Udover de her nævnte operationer kan man også have betingede udtryk (**if-then-else**-udtryk for alle datatyper).

**Heltal.** Heltal er hvad der gemmer sig under datatypen **integer** i de to sprog. Lad os se på forskellene i de to sprog.

Konstanter. I Pascal skrives et heltal som en stribe cifre. I Simula er der to varianter over det tema. Til at gruppere tusinder og millioner i store tal kan man bruge en understregning. En million kan således skrives som "1\_000.000". Understregninger må bruges frit i tal, men må ikke starte et tal. Tal i 2-, 4-, 8-, og 16-talssystemet kan skrives som radiks efterfulgt af et stort "R" efterfulgt af tallet. F.eks angiver "16READ" tallet EAD i 16-talssystemet, hvilket også er 3757 i 10-talssystemet.

Interval. I Standard Pascal er der kun en slags heltal. Implementationer kan ofte tilbyde et større arsenal. I Turbo Pascal kan man således vælge mellem *shortint*, *integer*, *longint*, *byte* og *word*. I Simula har man udover **integer** også **short integer**. Normalt vil **integer** være 32-bit heltal (altså mulighed for ganske store heltal) mens **short integer** vil være 16-bit heltal (tal i intervallet -32768..32767). Intervallerne for disse to slags heltal er dog implementationsafhængige og man skal ikke regne med større hastighed ved at bruge **short integer** i stedet for **integer**. Ofte vil beregninger nemlig ske med den størst mulige præcision, og derefter konverteret til den valgte resultattype. Der vil derimod ofte være en pladsbesparelse ved at vælge en type med mindre præcision.

Operationer. I Simula findes følgende operationer på heltal: plus (+) minus (−) gange (\*) heltalsdivision (//) (i Pascal: **div**) og potensopløftning (\*\*). Rest ved heltalsdivision (i Pascal: **mod**) findes som standardfunktion. Potensopløftning fungerer så man kan skrive "2\*\*3" og få resultatet 8.

Standardfunktioner. Som i Pascal vil *maxint* og *minint* give hhv. det største og mindste tal der kan repræsenteres af typen. Rest ved heltalsdivision klares af funktionerne *mod* og *rem* som opfører sig ens og som

Pascals **mod**-operation for positive tal, og med subtile forskelle for negative tal. *abs* kendes i begge sprog og *min* og *max* tager to argumenter og returnerer hhv. det mindste og største af de to tal. Funktionen *sign* returnerer hhv. -1, 0 og 1 hvis argumentet er hhv. negativt, 0 og positivt. Pascals *odd*-prædikat findes ikke. I stedet for *sqr*-funktionen har Simula den mere generelle potensopløftning som operator.

Typenavn. Der er en lidt mere spidsfindig forskel mellem ordet “integer” i de sprog. I Pascal er “integer” en standardtype, dvs. en prædefineret type, men ellers lige som alle andre typenavne i sproget. Man kan derfor omdefinere navnet til noget andet inde i procedurer. I Simula er “integer” et reserveret ord, og det må altså ikke bruges i andre sammenhænge. rsagen til dette har først og fremmest noget med indlæsningen i oversætteren at gøre. I Pascal kan oversætteren se at der kommer en variabel-erklæring ved at genkende ordet **var**, som derfor er reserveret. I Simula kan oversætteren se at der kommer en erklæring ved at der står et typenavn. I disse noter er reserverede ord forsøgt skrevet med tykke typer, og dette afsnit har derfor været en sammenligning af Pascals *integer* type og Simulas **integer** type.

[Yderligere om heltal i HBH: p 86-89, 95-96, 359-360]

**Kommatal.** Kommatal, eller mere præcist flydende kommatal (*eng: floating point numbers*), repræsenteres i begge sprog ved typen **real**. I begge sprog følges den angel-saksiske tradition med at adskille decimaldelen af et tal med et punktum, mens vi jo på dansk normalt bruger et decimal-komma. Man skal således ikke lade sig forvirre af oversættelsen af begrebet til “kommatal”.

Konstanter. Meget store eller meget små kommatal angives normalt med et kommatal efterfulgt af en eksponent. I Pascal bruges et “E” til at angive eksponenten, i Simula et ampersand (“&”). I Pascal kan man således skrive “12.5E+3” og i Simula “12.5&+3” for tallet 12500. I begge tilfælde kan plus i eksponenten udelades, og negativ eksponent kan bruges for meget små tal..

Præcision. I Simula er der to slags kommatal: **real** og **long real**. Konstanter af typen **long real** angives med et dobbelt ampersand-tegn før eksponenten: f.eks. “12.5&&+3”.

Operationer. I Simula findes følgende operationer på kommatal: plus (+) minus (−) gange (\*) division (/) og potensopløftning (\*\*).

Standardfunktioner. De to konstanter *maxreal* og *minreal* giver hhv. det største og mindste tal der kan repæresenteres af typen **real**. For typen **long real** findes tilsvarende konstanter: *maxlongreal* og *minlongreal*. Funktionerne *abs*, *sign*, *max* og *min* kan bruges som for heltal. Funktionen *entier* afrunder kommatil til nærmeste mindre heltal. Dette svarer til Pascals *trunc* funktion. Pascals *round* funktion findes ikke (se dog under “typekonvertering” nedenfor). Der findes i Simula et utal af matematiske funktioner. Se HBH appendiks C for en oversigt.

[Yderligere om kommatil i HBH: p 96-105, 360-361]

**Logiske værdier.** Både Pascal og Simula har typen **boolean** til logiske værdier. I begge sprog hedder konstanterne *true* og *false*. I Simula initialiseres variable af denne type med værdien *false*, mens variable i Pascal ikke initialiseres.

Operationer. I Pascal findes der følgende operationer for logiske værdier: **and**, **or**, **not** og **=**. Derudover kan ordningsoperationer bruges idet *false* < *true*. I Pascal er det ikke specificeret om begge argumenter til **and** og **or** udregnes hvis resultatet kan sluttet af den først udregnede værdi (som så iøvrigt ikke behøver være første argument).

I Simula kan man også bruge operationerne **and**, **or** og **not** kendt fra Pascal. Derudover findes et antal andre logiske operationer, men til gengæld kan man ikke bruge de almindelige ordningsoperationer. I Simula foreskrives en venstre-til-højre udregning, og begge argumenter udregnes til **and** og **or**. Desuden findes operationerne **and then** og **or else** som fungerer lige som **and** og **or** bortset fra at andet argument kun udregnes hvis nødvendigt. Endvidere findes operationerne **eqv** svarende til lighed (I Pascal:  $p = q$ ). og **imp** svarende til implikation (i logikken  $p \Rightarrow q$ , i Pascal lidt ulogisk:  $p \leq q$ ).

Talrelationer. Tal kan sammenlignes med de samme relationer i de to sprog. Dvs. =, <>, >, <, >=, <=. Har man problemer med sit tastatur kan man bruge følgende synonymmer for relationerne: **eq**, **ne**, **gt**, **lt**, **ge**, **le**.

Præcedensregler. Der er visse forskelle på præcedensreglerne i de to sprog. I Pascal skal man huske parenteser i udtryk som  $(x < y)$  **and**  $(y < z)$ , hvorimod dette ikke er nødvendigt i Simula. I Simula binder alle talrelationer tættere end logiske operationer og de logiske operationer binder

tættest efter følgende liste **not, and, or, imp, eqv, and then, or else, if**.

[Yderligere om logiske værdier i HBH: p 105-107, 89]

**Tegn.** Typen *char* fra Pascal hedder **character** i Simula. Der gøres i Simula et nummer ud af at forskellige maskiner kan operere med forskellige tegnsæt. Som almindelig bruger er det dog ikke noget man behøver at bekymre sig om, og hvis man har skrevet sit program så det heller ikke bekymrer sig om det, er det da også meget mere flytbart.

Konstanter. Lige som i Pascal angives en tegnværdi i Simula som et tegn omgivet af apostroffer. Et apostrof angives i Simula blot som et enkelt apostrof omgivet af apostroffer. Derudover kan man angive isotegnkoder ved et tal omgivet af udråbstegn, omgivet af apostroffer. F.eks “ ’!32!’ ”. Det er dog ikke noget man skal vide til normal brug af sproget, og hvis man føler sig foranlediget til det så brug det kun i konstanterklæringer i starten af programmet. Man bør normalt ikke basere sit program på hvilke tegn der bruges til linieafslutning, for på det område er Unix, Dos, og Macintosh vildt uenige. Hverken i Pascal eller Simula er det dog nødvendigt at vide noget om, for det vil indlæsningsprocedurerne normalt tage højde for.

Funktioner. Til konvertering mellem tegn og tegnkoder findes der i Simula funktionerne *char* og *rank* svarende til Pascals *chr* og *ord*. Desuden findes funktionerne *isochar* og *isorank* som arbejder med isotegnsystemet. Den hyppigste anvendelse af de funktioner er nok at afkode værdien af tal indlæst tegnvist, og der er den eneste antagelse at cifre har fortløbende tegnkoder. Det har de så vidt vides i alle tegnsystemer. Derudover findes to logiske funktioner *digit* og *letter* som undersøger om argumentet er hhv. et ciffer eller et bogstav.

Operationer. Som i Pascal kan man bruge de sædvanlige ordnings- og sammenligningsoperationer (=, <>, >, <, >=, <=) for tegn.

[Yderligere om tegn i HBH: p 107-109]

**Tekststreng.** Vi skal snakke om tekststreng som datatype senere, men her blot sige at tekststreng som konstanter kan optræde i en række sammenhænge. Det drejer sig bl.a. om argumenter til udskrivningsrutiner. I Simula angives en tekststreng ved en (evt. tom) følge af tegn

omgivet af citationstegn (gåseøjne) og hvor citationstegn i strengen skal skrives dobbelt. I tekststrengen kan man også bruge isotegnkode efter notationen beskrevet ovenfor. I Pascal bruges apostroffer til det samme formål.

### Pascal

''' tegnet et apostrof  
""" tekststrengen et citationstegn

### Simula

'''  
''''

**Typekonverteringer.** I tildelingssætninger skal typen af variabelen på venstresiden og udtrykket på højresiden stemme overens for begge sprog. Pascal og Simula har dog undtagelser for taludtryk, men undtagelserne er forskellige. I Pascal må man bruge en heltalsværdi som en kommatalsværdi, men ikke omvendt. I Simula kan man tildele en heltalsvariabel værdien af et kommatal, hvorved tallet først afrundes. Tallet afrundes til nærmeste heltal efter samme regler som Pascals *round*-funktion.

### Pascal

```
var x : integer; y : real;  
begin  
  y := 3.14159;  
  x := round(y)  
end.
```

### Simula

```
begin  
  integer x ; real y;  
  y := 3.14159;  
  x := y  
end
```

Også ved parameteroverførsel til procedurer har Simula blødere regler end Pascal. I Pascal kræves at aktuel og formel parameter har samme type ved variabeloverførsel, mens noget tilsvarende ikke kræves i Simula.

Multipel tildeling. Som nævnt tidligere kan man i Simula have multiple tildelingssætninger. I sekvensen

### Simula

```
begin  
  integer x ; real y;  
  y := 3.14159;  
  y := x := y  
end
```

får både “x” og “y” værdien 3. Det skyldes at “y” tildeles værdien af “x := y” og den tildeling har den værdi som tildeles venstresiden - altså den afrundede værdi “3”.

**Tabeller.** Der er tre forskelle på tabeller (*eng. array*) i de to sprog. I Simula kan man ikke (umiddelbart) have tabeller af tabeller, men man må nøjes med flerdimensionale tabeller. I Pascal kan man til gengæld have dynamiske tabeller. Det sidste betyder at hvor tabelgrænser skal være kendt på oversættelsestidspunktet i Pascal, kan man i Simula have udtryk stående som tabelgrænser, og disse udtryk udregnes på udførelsestidspunktet når tabellen erklæres. Den sidste forskel er at Pascal bruger firkantede parenteser til indicering af tabeller, mens Simula bruger runde.

Erklæring. Erklæring af tabeller ser lidt forskellig ud i de to sprog:

### Pascal

```
var a : array [1.. 10] of integer;
begin
  a[1] := 5;
  a[2] := a[1]*2
end.
```

### Simula

```
begin
  integer array a(1:10);
  a(1) := 5;
  a(2) := a(1)*2
end
```

En tabels interval angives således som et kolon-separeret par i Simula. Flerdimensionale tabeller angives ved at angive flere komma-separerede intervaller. Man kan desuden erklære flere tabeller, med forskellige intervaller men samme grundtype i samme linie. Således vil

### Simula

```
integer array a,b(1:n),c(2:5,3:6),d(1:n,1:n),e(1:n*n);
```

erklære fem tabeller hvor "a" og "b" har "n" felter (og "n" kan altså være en variabel), "c" er to-dimensional med 16 felter og "d" og "e" har begge  $n * n$  felter, men med "e" som en en-dimensional tabel. Tabellernes størrelse afhænger her af værdien af "n" når erklæringen nåes. Hvis "n" ændres derefter så ændrer det ikke tabellernes størrelse. Man kan derimod godt have erklæringer i blokke som udføres flere gange, og hvor tabellers grænser varierer. I eksemplet nedenfor vil tabellen "tab" blive erklæret ved hvert gennemløb af **for**-løkken, men med forskellige grænser.

## Simula

```
begin
  integer i;
  for i := 1 step 1 until 10 do
    begin
      integer array tab(i:10);
      ! og her kan man så bruge tabellen tab ;
    end
  end
```

Funktioner. Der findes to funktioner for tabeller: *lowerbound* og *upperbound* som returnerer hhv. nedre og øvre grænse for en tabels interval i en given dimension. I eksemplet ovenfor vil *upperbound(c,1)* således være 5 og *upperbound(c,2)* er 6. Grænserne er heltalsværdier, men udtrykkende, der bestemmer grænserne må gerne være kommatalsudtryk, som så afrundes efter sædvanlige regler.

Typeløs tabel? Hvis man ser efter i syntaksdiagrammerne for Simula kan man se at man ikke behøver at skrive en simpel type før ordet **array** i en erklæring. Udnytter man det får man en tabel af kommatal per automatik. Det er måske en ide hvis man vil narre fjenden?

[Yderligere om tabeller i HBH: p 50-51,117-122]

**Typer og erklæringer.** Vi har nu set fire simple typer og en typekonstruktør. Derudover findes der to andre typer i Simula som vi skal omtale senere. Disse to kaldes reference-typer og det drejer sig om hægter (eller objekt-referencer) og tekststrengene. Disse typer angives i erklæringer som hhv. **ref(klassenavn)** og **text**. Nærmere om disse typer senere. Ialt er der således 6 simple typer (eller 8 hvis vi også medregner **short integer** og **long real**) og man kan have simple variable, tabeller og funktioner af disse typer.

Umiddelbart kan dette virke som et væsentligt mere begrænset udbud end Pascal, hvor man også kan have følgende typer: mængder (**set**), poster (**record**), variant-poster (**record-case**), filer (**file**), nummererede typer og intervaller. Vi skal dog senere se at det stort set alt sammen (og mere til) er tilgængeligt gennem Simulas klassebegreb.

I Simula kan erklæringsdelen af **begin-end**-blok bestå af en eller flere af følgende slags erklæringer.

- Variabel- og konstant-erklæringer. Simple variable erklæres ved et typenavn og en komma-separeret liste af variabelnavne. I denne liste kan man også have konstant-erklæringer ved at skrive et konstantnavn, et lighedstegn og et udtryk. Eksemplet her erklærer således tre variable og to konstanter.

```
integer i,j,n=10,m=n**2,cnt;
```

Modsat Standard Pascal må der i konstant-erklæringer godt stå udtryk, så længe de kan udregnes på oversættelsestidspunktet.

- Tabel-erklæringer. Vi har allerede set tabel-erklæringer. Bemærk at man ikke kan have konstant-tabel-erklæringer.
- Switch-erklæringer. Disse er beskrevet tidligere.
- Procedure-erklæringer. Disse vil blive beskrevet senere.
- Klasse-erklæringer. Disse vil blive beskrevet senere.
- Eksterne erklæringer. Disse erklæringer giver mulighed for separat oversættelse og giver altså information om hvilke erklæringer der findes i allerede oversatte dele af et samlet program. (*Se HBH side 167 for yderligere detaljer*).

Erklæringer kan komme i en vilkårlig rækkefølge. Man kan således blande variabel og procedure-erklæringer. Det er altså ligesom de fleste Pascal implementationer, men modsat Standard Pascal.

Erklæringer tager normalt effekt fra starten af den blok hvori de er erklæret. Konstanter værdi må derimod først bruges efter det sted hvor de er erklæret. I eksemplet ovenfor må man altså godt bruge konstanten "n"s værdi til at bestemme værdien af konstanten "m". Konstanterne kunne derimod ikke være erklæret i den omvendte rækkefølge.

## 2.4 Indlæsning og udskrivning

I både Simula og Pascal benyttes normalt linieorienteret indlæsning gennem særlige procedurer og udskrivning sker gennem formateringsprocedurer. Den væsentligste forskel er at disse procedurer hedder noget

forskelligt. Vi skal her se på de centrale dele af indlæsning og udskrivning, men det er kun specialtilfælde af meget mere generelle muligheder for fil-behandling i Simula. I Simula er filer objekter og den videre omtale hører retteligt hjemme under objekt-orienteret programmering.

**Indlæsning.** At indlæsning sker lineorienteret betyder at systemet indlæser en hel linie af gangen til en buffer, og den videre indlæsning sker så fra bufferen. Det er det som gør at man kan taste løs og bruge “delete”-tasten - og først når man har trykket på “retur”-tasten fanger bordet.

Indlæsning sker i Simula ved brug af følgende procedurer og funktioner:

*inimage.* Proceduren fungerer nogenlunde som *readln* i Pascal, men med den væsentlige forskel at man ikke alene skipper evt. resterende tegn på den aktuelle linie, men også læser den næste linie ind til lageret.

*endfile.* Boolsk funktion, der fungerer helt som *eof* i Pascal. På Unix maskiner betyder det at man har tastet kontrol-D og på pc'ere at man har tastet kontrol-Z.

*more.* Boolsk funktion, der fungerer som *eoln* i Pascal, men med det modsatte resultat - hvis der er mere er vi ikke nået til slutningen af linien.

*inchar.* Tegn-funktion, hvor “*c := inchar*” fungerer som “*read(c)*” i Pascal, hvis “*c*” er en tegnvariabel.

*inint.* Heltals-funktion, hvor “*i := inint*” fungerer som “*read(i)*” i Pascal, hvis “*i*” er en heltalsvariabel.

*inreal.* Kommatafs-funktion, hvor “*r := inreal*” fungerer som “*read(r)*” i Pascal, hvis “*r*” er en kommatafsvariabel.

*intext(antal).* Streng-type-funktion, som indlæser et givent antal tegn. Funktionen omtales senere i forbindelse med streng-typer.

Ud over disse har man ved indlæsning i Simula adgang til en stribe andre procedurer og funktioner, der dog ikke umiddelbart modsvarer af tilsvarende faciliteter i Pascal.

[Om indlæsning se HBH 235-269, særligt 240-242]

**Udskrivning.** I Simula sker udskrivning også linieorienteret. Dvs. at normal udskrivning sker til en buffer, som man så eksplicit skal få skrevet ud. Normalt vil det ikke være anderledes end i Standard Pascal, hvor tekst ikke behøver at blive skrevet ud før man har set et kald til *writeln*. I Turbo Pascal sker udskrivning med det samme og det gør det lettere at skrive interaktive programmer. I Simula kan det også lade sig gøre, men så skal man eksplicit bede om at få en del af en linie udskrevet ved et kald til *breakoutimage*. Lad os se lidt nærmere på udskrivningsprocedurerne i Simula.

*outchar(tegn)*. Kaldes med et tegn som argument og fungerer så som “write” i Pascal.

*outint(tal,længde)*. Kaldes med et heltal og en længde, og tallet udskrives højrestillet inden for den givne længde. Kaldet “outint(i,w)” svarer nogenlunde til Pascal-kaldet “write(i:w)”. Hvis der ikke er plads til tallet inden for “w” positioner er det en fejl i Simula, mens Pascal blot vil bruge det nødvendige antal pladser. Hvis “w” er nul bruges kun det nødvendige antal pladser og kaldet svarer altså til “write(i)”. Hvis “w” er negativ udskrives venstrestillet på “-w” pladser.

*outfix(tal,antal,længde)*. Kaldes med et kommatal og to heltal, hvor det første angiver antal decimaler og det andet antal positioner. Kaldet “outfix(r,n,w)” svarer altså nogenlunde til kaldet “write(r:w:n)” men med de samme restriktioner for “w” som for “outint”. Bemærk dog rækkefølgen af formateringsparametrene i de to sprog.

*outtext(tekst)*. Kaldes med en tekststreng (evt. fra en streng-variabel). Kaldet “outtext(“hej verden”)” i Simula svarer altså til kaldet “write(‘hej verden’)” i Pascal.

*outimage*. Fungerer som “writeln” i Pascal.

*breakoutimage*. Tømmer den interne buffer, men skifter ikke linie. Den bruges normalt hvis man skal udskrive ledetekster før en indlæsning, og gerne vil have at denne ledetekst står på samme linie som det indtastede vises.

[Om udskrivning se HBH 235-269, særligt 242-244]

**Eksempel.** Indlæsning og udskrivning kan illustreres ved en lille bid kode som kopierer inddata til uddata. Det væsentligste forskel er at man i Simula med “inimage” kopierer en linie ind i den interne buffer før linien læses. I Pascal sker dette skjult bag “eoln” og “eof” kald.

### Pascal

```
var c : char;
begin
  while not eof do begin
    while not eoln do begin
      read(c);
      write(c)
    end;
    readln;
    writeln
  end
end.
```

### Simula

```
begin
  while not endfile do begin
    inimage;
    while more do
      outchar(inchar);
    outimage
  end;
end
```

I Simula-programmet har vi iøvrigt kunnet udnytte at indlæsningen sker ved hjælp af en funktion og ikke en procedure. Det gør at tegn-variablen “c” er overflødig.

**Afsluttende blanktegn, mm.** Maksimale linielængder samt linie- og fil-afslutningstegn er typisk implementationsafhængige detaljer man skal gøre sit program mest muligt uafhængig af. Lad os alligevel nævne noget om disse detaljer. I Simula antages det normalt at alle inddatalinier er 80 tegn lange, og systemet vil automatisk fylde op med blanktegn om nødvendigt. Prøver man at indtaste en linie som er længere end 80 tegn vil det give en køretidsfejl. Uddatalinier kan maksimalt være 132 tegn lange, men der vil normalt ikke blive fyldt op til denne længde med blanktegn. Vi skal senere se at man desværre er nød til at kende til grænsen på 80 tegn hvis man vil bruge funktionen “intext”.

## 2.5 Procedurer og parametre

Pascal og Simula giver begge mulighed for at definere underprogrammer og vi skal her se hvilke forskelle der er.

**Procedurer og funktionserklæringer.** Lad os først se på parameterløse underprogrammer. For procedurer er den eneste forskel at erklæringer i Simula står efter **begin** og altså ikke før som i Pascal.

### Pascal

```
procedure underprog;  
begin  
  ..  
end;  
begin  
  underprog;  
  ...  
end.
```

### Simula

```
begin  
  procedure underprog;  
  begin  
    ..  
  end;  
  underprog;  
  ...  
end
```

For funktioner skriver man resultat-typen før ordet **procedure**, men ellers fungerer funktioner som i Pascal. Det vil bl.a. sige at resultat-typen enten skal være en simpel type eller en reference-type, men altså ikke f.eks. en tabel. Resultatværdien findes ved at tildele funktionsnavnet en værdi. Bruges funktionsnavnet i andre sammenhænge er det et rekursivt kald.

### Pascal

```
function underprog : integer;  
begin  
  ..  
  underprog := ..  
end;
```

### Simula

```
integer procedure underprog;  
begin  
  ..  
  underprog := ..  
end;
```

Der er en forskel på brugen af underprogrammer i de to sprog idet Simula-funktioner må kaldes som procedurer. Resultatværdien vil da blive ignoreret. Hvis man f.eks. skal skippe nogle tegn fra inddata (altså hvis man ønsker at læse videre efter disse tegn, men ikke er interesseret i tegnene selv) så kan det gøres ved at kalde "inchar" som en procedure.

Kroppen i et Simula-underprogram behøver ikke være en **begin-end**-blok, men kan godt være en enkelt sætning. Særligt for små funktioner kan det være en betydelig besparelse, når man blot skal udregne en simpel værdi fra argumenterne. Simulas blok-struktur gør det iøvrigt muligt at erklære procedurer i alle **begin-end**-blokke - altså også inde i kontrol-sætninger.

[Ydreligere om procedurer i HBH: pp 127-146]

**Parametre.** Advarsel: Der er nogle ret besværlige forskelle mellem Simula og Pascal når det drejer sig om parametre, så hvis man vil have tjek på det hele vil det være en ide at læse resten af dette afsnit et par gange, og gøre det grundigt.

I Pascal kan parametre<sup>1</sup> til procedurer og funktioner enten overføres som værdi eller som variabel. Variabel-overførsel bruges normalt når man ønsker at få et resultat tilbage fra et underprogram. Når vi kalder proceduren "read" så sker det med en variabel som argument, og vi ønsker at få det indlæste gemt i denne variabel. Ved værdi-overførsel udregnes argumentet i kaldet, og i underprogrammet kan parameteren så bruges som en lokal - men initialiseret - variabel. Normalt bruges værdi-overførsel altså når man skal sende noget til et underprogram, og variabel-overførsel når man skal have noget retur fra et underprogram. Hvis man i et underprogram skal bruge en større struktur er det imidlertid ikke hensigtsmæssigt at bruge værdi-overførsel, hvor der jo så bliver lavet en kopi af denne struktur. Derfor vil man ofte bruge variabel-overførsel når man arbejder med større tabeller, selv om underprogrammet ikke skal ændre i sådanne strukturer.

Hvis man ikke ber om andet vil man i Simula for simple typer få værdi-overførsel og for strukturer få variabel-overførsel. Det er også det man helst vil når man skal overføre værdier til procedurer, og næsten nok når man skal have værdier tilbage. Man kommer altså ikke ved en forglemelse til at kopiere store strukturer fordi man undlod at skrive et lille **var**.

Hvad mangler så? Jo, man kan ikke variabel-overføre variable af simple typer. Hvis man ønsker at returnere en enkelt værdi af simpel type kan det selvfølgelig ske som funktionsværdi, men hvis vi skal returnere flere værdier må man bruge en helt tredje overførselsmetode i Simula: Navne-overførsel.

Navne-overførsel. Ved navneoverførsel skal programmet opføre sig som om argumentet kopieres ind alle de steder parameteren bruges i underprogrammet. Hvis der står en beregning som argument i et kald skal denne beregning udføres hver gang parameteren bruges. Argumentet skal hver gang udregnes i det virkefelt, der var på kald-stedet således at der

---

<sup>1</sup>I Pascal kaldes argumentet - altså det der står i et kald - for den aktuelle parameter, og parameteren - altså det navn man bruger i procedure-hovedet - kaldes den formelle parameter. Vi vil her blot omtale det som hhv. argument og parameter.

ikke i argumentet kan indgå variable erklæret lokalt i proceduren. Der skal nok nogle eksempler til før man kan se hvad det betyder i praksis.

Værdi-overførsel. Hvis man ønsker at der skal tages en kopi af en tabel ved overførslen kan man specificere at den skal værdi-overføres.

Erklæring. Parametre til en procedure skrives i en liste efter procedurenavnet. Typerne og overførselsmåden for parametre skrives efter procedurehovedet, og før procedurekroppen - altså hvor man i et Pascal program ville have haft lokale erklæringer. En funktion, der lægger to tal sammen, kan se således ud i de to sprog.

### Pascal

```
function sum(a,b : integer): integer;  
var res : integer;  
begin  
    res := a+b;  
    sum := res  
end;
```

### Simula

```
integer procedure sum(a,b);  
integer a,b;  
begin  
    integer res;  
    res := a+b;  
    sum := res  
end;
```

Den lokale variabel “res” er dog ikke nødvendig, og i Simula kunne man blot have skrevet.

### Simula

```
integer procedure sum(a,b); integer a,b; sum := a+b;
```

Tabeller. Skal vi beregne summen af elementerne i en tabel kan det f.eks. ske ved følgende lille funktion

### Pascal

```
type tb = array [1..20] of integer;  
function sumtab(var a:tb):integer;  
var i,s : integer;  
begin  
    s := 0;  
    for i := 1 to 20 do s := s+a[i];  
    sumtab := s;  
end;  
begin  
    ... sumtab(...)  
end.
```

### Simula

```
begin  
    integer procedure sumtab(a);  
        integer array a;  
        begin  
            integer i,s;  
            for i := lowerbound(a,1) step 1  
                until upperbound(a,1)  
                do s := s+a(i);  
            sumtab := s;  
        end;  
    ... sumtab(...)  
end
```

I dette eksempel har vi brugt variabel-overførsel af tabellen. I Simula er det standard, mens man i Pascal eksplicit må skrive **var**. I Simula angiver man ikke grænserne for tabellen, så den samme funktion kan altså bruges for heltalstabeller af forskellig størrelse. Faktisk kan man ikke engang angive tabelgrænser i procedure-hovedet. I Pascal skal grænser angives - ja man skal ovenikøbet bruge et fælles typenavn til erklæring af parameter og argument. Det skyldes at Pascal bruger en form for navne-ækvivalens når parameter- og argument-type skal sammenlignes. Pascals regler herfor er iøvrigt uhyre komplicerede og kun de færreste Pascal-programmører har helt styr over det. Man kan endvidere se at variabelen "s" i Simula ikke kræver en eksplicit initialisering - det sker automatisk.

Navneoverførsel. Hvis man ønsker en anden overførselsmetode end den man får per automatik - altså andet end værdi-overførsel for simple typer og variabel-overførsel for tabeller, så skrives det som en **name**- eller **value**-erklæringer umiddelbart efter procedurehovedet og altså før man har erklæret typen af parametre. Hvis vi skal skrive en lille procedure som ombytter værdien af to variable vil vi i Pascal bruge variabel-overførsel og i Simula navne-overførsel.

### Pascal

```
procedure ombyt(var a,b : integer);  
var c : integer;  
begin  
  c := a; a := b; b := c;  
end;
```

### Simula

```
procedure ombyt(a,b);  
name a,b; integer a,b;  
begin  
  integer c;  
  c := a; a := b; b := c;  
end;
```

Normalt vil man ikke bemærke nogen forskel mellem de to procedurer. I alle normale sammenhænge vil de fungere ens, og som tidligere Pascal-programmør er navne-overførsel den direkte parallel til variabel-overførsel. Det er imidlertid ikke det samme, og det skal vi illustrere med det følgende lille eksempel.

### Simula

```
begin  
  integer array tab(1:2);  
  integer i;  
  i := 2; tab(1):= 3; tab(2):= 1;  
  ombyt(i,tab(i))  
end
```

Med variabel-overførsel får “i” værdien 1, og tabellen “tab” værdierne 3 og 2. Med navne-overførsel får tabellen værdierne 2 og 1. Med navne-overførsel vil kaldet “ombyt(i,tab(i))” nemlig svare til følgende sekvens af sætninger

### Simula

```
c := i;           eller c := 2
i := tab(i);      eller i := 1
tab(i) := c ;     eller tab(1) := 2
```

idet vi blot har indsat “i” og “tab(i)” i stedet for “a” og “b” i kroppen på procedure “ombyt”. Årsagen til forskellen er at det indeks til tabellen “tab” som optræder i kaldet, først udregnes når den tilsvarende parameter bruges - og så udregnes det begge gange parameteren bruges. Ved variabel-overførsel udregnes indeks på kald-tidspunktet og der overføres en reference til den givne plads i tabellen.

Hvis vi havde omdøbt den lokale variabel “c” i procedure “ombyt” til “i” havde det ikke ændret noget. Det skyldes den smørre om at argumenterne skal udregnes i det virkefelt, der var på kaldstedet. Alternativt kan man sige at argumenter substitueres ind for parametrene i proceduren, dog således at lokale variable om nødvendigt omnavngives for at undgå et evt. navnesammenfald.

**Om navne-overførsel og Algol.** Hvis man synes navne-overførsel er noget underligt noget, kan man berolige sig med at det er man ikke den eneste som synes. Den væsentligste grund til at vi idag programmerer i Pascal og C, mens Algol nu stort set er et dødt sprog, er at Algol havde navne-overførsel som standard. Uden denne design-fejl (ja, det kan man vist godt kalde det) havde der næppe været så stort et spillerum for at lave forbedringer af Algol. Da man designede Simula ændrede man Algols overførselsmåde så værdi- og variabel-overførsel er standard, afhængig af om typen er simpel eller sammensat. Det er en forbedring over Algol, men gør at man så har tre forskellige overførselsmåder: Værdi-, variabel- (også kaldet reference-), og navne-overførsel. Navne-overførsel er et festligt redskab for eksperten, og det gør det muligt at skrive ganske overraskende programmer. Nu er det selvfølgelig ikke så ofte et mål i sig selv at et program skal overraske, men “Jensen’s device”<sup>2</sup> er en af datalogiens sande perler.

---

<sup>2</sup>Se *HBH side 141-144*

**Procedurer-som-parametre.** Vi har nu omtalt simple værdier og tabeller som parametre. Derudover har man en stribe andre muligheder for parametre til procedurer. I både Standard Pascal og Simula kan man have procedurer som parametre. Ikke alle Pascal implementationer gør dog det; det findes f.eks. ikke fuldt implementeret i Turbo Pascal.

I sig selv er begrebet ikke så besværligt. Man kan tage et procedure-navn og bruge det som parameter til en procedure. I proceduren kan man så kalde den overførte procedure. Man kunne f.eks. forestille sig at man har en lille beregningsfunktion som skal kunne komme med fejlmeddelelser i visse tilfælde. Afhængig af sammenhængen vil man gerne have fejlmeddelelser på dansk, engelsk eller oldbulgarsk. Så skriver man tre procedurer som udskriver fejlmeddelelser og kalder beregningsfunktionen med en af disse tre procedurer som parameter. Enkelt? Ja! Rigtig rigtig rigtig enkelt? Nå ja, måske ikke. Det er ihvertfald aldrig blevet det store hit i Pascal. Hvis der er flere forhold man gerne vil parametrisere over bliver det ganske enkelt for tungt at skrive, og for svært at vedligeholde. Til sådanne situationer har den objekt-orienterede tankegang nogle meget pænere løsninger.

Procedurer-som-parametre illustrerer tydeligt de grundlæggende ideer i blok-strukturerede programmeringssprog - og det vil sige stort set alle moderne programmeringssprog. Vi kan jo bruge lejligheden til at slå nogle begreber fast.

**Virkefelt.** En variabel har et virkefelt - nemlig den blok hvori den er erklæret. Der kan være huller i virkefeltet hvis variabelens navn erklæres til et andet formål i en lokal blok. Inde i denne lokale blok har man så ikke umiddelbart adgang til den ydre definerede variabel.

**Instanser.** En lokal variabel i en procedure kan optræde i flere instanser samtidig. Ved rekursive kald af en procedure kan der være flere aktiveringer af en procedure igang på samme tidspunkt. For hver aktivering af proceduren skal der afsættes plads i lageret til en instans af den lokale variabel. Når en aktivering af proceduren afsluttes kan den tilhørende instans inddrages. Dette kan internt organiseres ved en køretidsstak. Det skyldes at senere aktiveringer altid skal afsluttes før tidligere.

Det første af disse begreber omtaler statiske forhold - hvad man kan se ud af programteksten. Det andet drejer sig om det dynamiske - hvad

der sker under programmets udførelse. På grund af virkefeltsreglerne kan man godt erklære flere variable med samme navn (bare det er i forskellige blokke), og hver variabel kan eksistere i flere instanser.

Procedurer-som-parametre følger de samme virkefelts- og instans-regler som for alle andre parameter-overførsels-metoder. Med hensyn til disse forhold er det altså også disse regler som gælder for navne-overførsel og variabel-overførsel. En procedure som parameter vil blive udført i det virkefelt og med adgang til de instanser af variable, der var da proceduren blev brugt som parameter. Instansernes værdier kan derimod godt have ændret sig og det er de aktuelle værdier som man har adgang til.

Prøv nu at bruge disse simple regler på følgende lille program

### Pascal

```
var y : integer;
  procedure skip; begin end;
  procedure p(procedure p1);
    var x : integer;
    procedure q;
    begin writeln(x) end;
    procedure r;
    begin
      if x < 1 then p(q) else p1
    end;
  begin
    x := y; y := y+1; r
  end;
begin
  y := 0; p(skip)
end.
```

### Simula

```
begin
  integer y;
  procedure skip ; begin end;
  procedure p(p1);
    procedure p1;
    begin
      integer x;
      procedure q;
      begin outint(x,1); outimage end;
      procedure r;
      begin
        if x < 1 then p(q) else p1
      end;
      x := y; y:=y+1; r
    end;
  y := 0; p(skip)
end
```

Det er de samme regler der gælder i de to sprog. Man kan se at variabelen “y” starter med at være nul, og at den i “p” tælles op til 1, og at man i “r” derefter kalder “p” igen, hvorefter “y” tælles op til 2. Såvidt så godt, men hvad med parameteren “p1”? Hvis man mener der udskrives et 1-tal har man forstået en hel del om programmeringssprog og skal bestemt ikke skamme sig. Hvis man mener der udskrives et nul har man nok forstået alt hvad der er værd at vide om blokstrukturerede programmeringssprog. Tillykke!

**Parametertyper.** Det er nu tid til den samlede oversigt over parametertyper og overførselsmåder i Simula.

*simple typer.* For de simple datatyper - altså tal, tegn og den boolske type - overføres normalt (altså per *default*) værdi-overført, men man kan bruge navne-overførsel.

*simple tabeller.* Tabeller af simple datatyper overføres normalt variabel-overført. Det kaldes også reference-overført. Man kan også specificere at tabellen skal værdi-overføres, hvorved der overføres en kopi af tabellen.

**ref(klassenavn).** Hægter (eller objekt-referencer) skal vi omtale senere. Hægter overføres som en reference - altså som en adresse. Det kan man jo godt kalde reference-overført. I Pascal er dette også det normale, og der bliver det kaldt værdi-overførsel. Det gør man ikke her, for så ville der bare være nogen, der troede at der vil blive taget en kopi af det hægten pegede på. Derudover kan man navne-overføre sådanne hægter og det svarer nogenlunde til at variabel-overføre hægte-variable i Pascal.

**text.** Tekstvariable skal vi omtale senere. Som vi skal se er tekstvariable at opfatte som referencer til tekst-objekter, og opfører sig derfor som objekt-referencer.

**ref(..) array.** Tabeller af hægter variabel-overføres. Man kan altså ikke under overførslen få lavet en kopi af tabellen. Det kan man jo altid gøre i proceduren hvis man virkelig føler behov for det. I Pascal kan man godt værdi-overføre en tabel af hægter. Så får man taget en kopi af hægterne, men selvfølgelig ikke en kopi af det hægterne peger på.

**text array.** Tabeller af tekster variabel-overføres lige som for **ref(..) array**.

**procedure.** Procedurer overføres som beskrevet ovenfor. Det kaldes for reference-overført, men det er lidt misvisende, for det er ikke bare et spørgsmål om en reference til en procedure - der var jo også alt det der med instanser af variable.

*funktioner.* Funktioner er blot procedurer med type og opfører sig iøvrigt som procedurer.

**label.** Man kan bruge etiketter som parametre. De skal erklæres med “typen” **label** og så kan man bruge parameteren i en **goto**-sætning. Ved overførslen gælder iøvrigt de samme regler som for procedurer. Altså at mht. til instanser og aktiveringer af procedurer svarer det til at hoppe til den etikette der var adgang til da etiketten blev brugt som parameter.

**switch.** Også **switch**-erklæringer kan overføres, og de opfører sig iøvrigt som etiketter og procedurer.

Udover disse overførsels-måder kan man for alle parameter-typer specificere navne-overførsel. Det vil dog sjældent gøre andet end en reference-overførsel - det kræver nemlig at man ved at udregne referencen flere gange kan få forskellige værdier. Udover det man får per standard kan det være en god ide at begrænse sig til at bruge navne-overførsel for simple typer og objekt-referencer. Navne-overførsel bruges så når man skal have returneret noget fra et underprogram og når det ikke er mere naturligt at bruge en funktion. I Simula vil man ofte bruge funktioner til at returnere værdier hvor Pascal bruger variabel-overførte parametre. Dette ses bl.a. i udvalget af standard-procedurer/funktioner (f.eks. ved indlæsning). Hvis man i Simula skal returnere et talpar fra et underprogram vil det ofte være naturligt at skive en funktion som returnerer en reference til en post med to tal, hvorimod pascalprogrammøren nok vil vælge at variabel-overføre en post. Alt ialt er der derfor sjældent brug for andre overførselsmåder end den værdi- eller reference-overførsel man får som standard i Simula.

Hvis man føler sig foranlediget til at bruge etiketter og switch-erklæringer som parametre bør man først konsultere sin praktiserende læge for at få et udvidet sundhedstjek.

[*Ydreligere om parametre i HBH: pp 133-140*]

**Navne-overførsel igen.** Ved variabel-overførsel i Pascal skal argumentet være et variabelnavn og typen af variabelen skal være den samme som parameterens type. Ingen af delene er tilfældet for navne-overførsel i Simula. Udtryk kan navne-overføres og parameteren kan så bruges som værdi, idet argumentet vil blive genudregnet ved hver brug. Det giver derimod en køretidsfejl, hvis man forsøger at tildele parameteren en værdi i proceduren.

I Simula kan man tildele en heltalsvariabel værdien af et kommatalsudtryk. Resultatet er at der implicit sker en afrunding af kommatallet. På samme måde kan man have en navne-overført parameter af typen **real**, hvor argumentet i et kald er en heltalsvariabel. En tildeling af en værdi til parameteren vil så resultere i en implicit afrunding af værdien. I eksemplet nedenfor vil "z" således få tildelt den afrundede værdi 3. Hvis man synes det er klart nok kan man gå igang med at fundere over at programmet nedenfor udskriver værdien "3.14".

## Simula

```
begin
  procedure p(x); name x; real x;
  begin
    real y;
    y := x := 3.14159;
    outfix(y,2,4); outimage
  end;
  integer z;
  p(z)
end
```

## 2.6 Poster - records og klasser

Det kan ofte være en nyttig facilitet at kunne gruppere data sammen på en eller anden måde. Til visse formål er tabeller den naturlige måde at gøre det på, og i Pascal er poster *records* også et hyppigt brugt struktureringsværktøj. F.eks. kan et punkt repræsenteres som et koordinatpar:

### Pascal

```
var p : record x,y : integer end;
begin
  p.x := 3; p.y := 4;
  writeln(p.x)
end.
```

I Simula fungerer poststrukturen noget anderledes end i Pascal. Man har således ikke *bare* poster, men man kan have - hvad der svarer til - hægter til poster. I de to sprog kan det udtrykkes således.

## Pascal

```
type pst = record x,y : integer end; begin
```

```
var p : ^pst;
```

```
begin
```

```
  new(p);
```

```
  p^.x := 3; p^.y := 4;
```

```
  writeln(p^.x)
```

```
end.
```

## Simula

```
class pst; begin integer x,y; end;
```

```
ref(pst) p;
```

```
p :- new pst ;
```

```
p.x := 3; p.y := 4;
```

```
outint(p.x,0); outimage
```

```
end
```

I Simula erklæres posten som en klasse (**class**), med et klassenavn. Klassenavnet svarer ganske nøje til et typenavn i Pascal. Derefter følger så elementerne i blokken i en **begin-end**-blok. Der menes faktisk blok, med alt hvad det medfører, men indtil videre kan man blot tænke på det som en liste af elementer i posten. Erklæringen af variable af denne klassetype sker ved at bruge **ref**-nøgleordet. Som ved andre variabelerk-læringer kan der følge en liste af navne. I Simula initialiseres variable normalt og reference-variable har initielt værdien **none**, svarende til at variabelen ikke peger på noget. I Pascal vil det svare til at man eksplicit initialiserer variabelen til **nil**.

Ved generering ser notationen lidt anderledes ud - i Pascal ved et kald til proceduren "new" - i Simula ved at skrive "p :- **new** pst". Vi skal nok vende tilbage til den notation om lidt. Når man skal bruge hægten skal man i Simula ikke skrive det der hak opad (^). I alt dette kan man så småt se noget af en holdningsforskel mellem de to sprog. I Simula snakker man om *objekt-referencer* og i Pascal om *hægter*. I Simula tænker man på det der peges på, mens det i Pascal snarere er hægten selv. I Simula-eksemplet ovenfor vil "p" være det der peges på, så man kan hive fat i førstekoordinaten som "p.x". I Pascal derimod er "p" hægten. Det der peges på er "p^", og man kan få fat i førstekoordinaten som "p^.x".

Vi så også denne forskel da vi snakkede om parametre til procedurer. I Pascal kan man godt værdi-overføre en hægte, for værdien af en hægte er blot en reference. I Simula kan man ikke værdi-overføre et reference-udtryk, for værdien er det objekt, der peges på. Man kan derimod godt reference-overføre - altså overføre en reference til objektet. Der sker altså det samme i de to sprog - man kalder det bare noget forskelligt.

Når værdien af en reference-variable nu er det objekt, der peges på, hvordan kan man så få fat i referencen selv? Hvordan kan man f.eks. sætte to reference-variable til at pege på det samme objekt? Jo, dertil

har Simula sin egen helt særlige tildelingssætning “:-”, hvor man får fat i referencerne, i stedet for det referencerne peger på. Vi har allerede set tildelingssætningen ved allokering af et objekt. Der skrev vi “p :- **new** pst” og her skaber “**new** pst” et nyt objekt af den rette type, og i tildelingen sættes “p” til at pege på dette objekt. Vi kan opsumere dette i en lille tabel.

| Pascal     | Simula                 | forklaring                                 |
|------------|------------------------|--|
| <b>nil</b> | <b>none</b>            | hægte-værdi: det tomme objekt              |
| new(p)     | p :- <b>new</b> <navn> | sæt “p” til at pege et nyt objekt          |
| p := q     | p :- q                 | sæt “p” til at pege på det “q” peger på    |
| p̂ := q̂   |                        | sæt det “p” peger på lig det “q” peger på. |

I den sidste form for tildeling skal alle elementerne i det objekt, som venstresiden peger på, have tildelt værdier hentet fra det objekt højresiden peger på. Dette kan man ikke i Simula - man må således ikke skrive “p:=q” og så håbe på at få kopieret indholdet af objektet<sup>3</sup>. I den næstsidste form er det blot en enkelt tildeling af en ny værdi (adresse) til en reference-variabel.

Lidt om ordene *klasse*, *reference-variabel* og *objekt*. I Simula har en *reference-variabel* en type, som indeholder navnet på en *klasse*. Vi skriver “**ref** (*klassenavn*)” i erklæringen. En variabel af denne type vil have en værdi som er en *objekt-reference*. Ved at bruge “**new**” operationen allokeres således et *objekt*. Initielt har reference-variablen en reference til det tomme objekt **none** som værdi. Dette er altså også et objekt og vi vil ofte snakke om reference-variable som om de har objektet som værdi.

**Inspect og with.** Af og til er punktum-notationen lidt tung når man skal have fat i elementer af en post. Begge sprog har imidlertid en sætningskonstruktion, der gør det muligt - så at sige - at åbne strukturen op så man direkte har adgang til elementerne i dem. I eksemplet ovenfor kunne tildeling til elementerne i posten derfor også ske således

### Pascal

```
with p̂ do
  begin x := 3; y := 4 end
```

### Simula

```
inspect p do
  begin x := 3; y := 4 end
```

---

<sup>3</sup>Som vi skal se senere er det dog muligt at kopiere indhold for tekst-objekter med denne notation

I begge tilfælde går dette kun godt hvis “p” peger på et “rigtigt” objekt - altså ikke til **nil**/**none**. **inspect**-sætningen i Simula hænger altså nøje sammen med referencer og der findes derfor en udvidet form af sætningen, hvor man kan have en ekstra **otherwise**-gren, som udføres hvis “p” peger på det tomme objekt **none**. F.eks.

## Simula

```
inspect p do  
  begin x := 3; y := 4 end  
otherwise  
  p := new pst
```

**Sammenligning af referencer.** For to reference-variable “p” og “q” kan man spørge om de peger på det samme objekt. I Simula skal man skrive “p == q” og i Pascal “p = q”. Man kan ikke direkte spørge om det der peges på er ens (altså indeholder samme værdier) Det skulle i Simula svare til “p = q” og i Pascal til “p^ = q^” - men det må man altså ikke. I stedet må man spørge for hver enkelt element-par om de er ens. Man kan spørge om to referencer er forskellige ved i Simula at skrive “p /= q” og i Pascal “p<>q”.

**Lidt indvendinger.** Mange Pascal-programmører får en lidt ubehagelig kløe når de ser hægter. Der er ofte en fornemmelse af at man lever på lånt tid, for allokeringerne bliver jo ikke nedlagt igen. Der kan jo hurtig oparbejde sig en mængde objekter man ikke længere har referencer til. Det er så spildplads i lageret og man kan så risikere at løbe tør for plads.

Ganske rigtigt, men i Simula har man taget højde for det, så i systemet er indarbejdet automatisk spildopsamling. Hvis systemet løber tør for plads vil det automatisk opsamle alle objekter, der ikke længere er referencer til, og så genbruge pladserne. Man kan selvfølgelig stadig løbe tør for plads, men det er altså ikke nær så farligt i Simula som i Pascal.

Simula er netop bygget til at arbejde med sådanne dynamiske strukturer. Set fra Pascal kan det godt virke lidt pudsigt med Simulas opfattelse af værdien af en reference-variabel som det objekt der peges på. Det gør det imidlertid mere naturligt at arbejde med objekter, og selve referencerne kan så ses som uinteressante implementationsdetaljer. Man skal nok lige

vænne sig til at Simulas objekt-referencer hverken helt er poster eller hæfter til poster. Det implementeres som det sidste, men som hovedregel kan man tænke på dem som det første.

[Ydreligere om klasser i HBH: pp 14-18, 163-204]

## 2.7 Tekststreng

Standard Pascal har ikke rigtig tekststreng som en datatype. Man kan bruge “**packed array .. of char**”, men det er på mange måder lidt klodset. En række implementationer har så tilføjet **string** som datatype, og vi vil her sammenligne Simulas **text** type med Turbo Pascals **string** type. Vi vil her referere til dem begge som *streng-typer*.

I Pascal bruges streng-typen som en tabel med variabel øvre grænse. Man kan således ændre i strengen og i det hele taget bruge den som en tabel af tegn. Det kan man sådan set også i Simula, men så skal man bare vide hvad man gør. I Simula er streng-variable hæfter til tekststreng og man skal i første omgang nok være lidt påpasselig med de destruktive (overskrivende) operationer. Simula- og Pascal-aktionerne nedenfor er altså ikke helt identiske, men så længe man holder sig fra de destruktive operationer vil de i praksis have samme effekt. Skal operationerne blandes med destruktive operationer må man i eksemplerne tage “copy” af højresiderne før tildelinger. Det skal vi dog nok vende tilbage til senere.

En række af de operationer vi skal bruge i Simula udtrykkes på en noget speciel måde. Man benytter en punktum notation - nærmest som om streng-variable var en poststruktur, hvor man har adgang til procedurer og funktioner i posten. Ja, det er ikke bare “som om” - sådan er det faktisk, men indtil videre kan man blot tænke på det som om det der står før punktummet er første argument til funktionen. Når man f.eks. skriver “s.length” svarer det altså til at man i Pascal skriver “length(s)”, og når man skriver “s.sub(2,4)” svarer det i Pascal til “copy(s,2,4)” idet begge giver den delstreng af “s” som starter i position 2 og er 4 tegn lang. Disse funktioner og procedurer som man angiver efter et punktum kan kun bruges på strengvariable og man kalder dem så *metoder* eller *attributter* knyttet til streng-typen. I Simula kan man godt bruge navnet “length” til andre formål og stadigvæk kan man finde længden af en tekststreng. Det sikrer de sædvanlige virkefelsesregler for poster (og klasser).

Simple tildeling. Vi vil lade “s” være en streng-variable og tildele den værdien “hej verden”. Det vil se således ud i de to sprog

## Pascal

```
var s : string(20);  
begin  
  s := 'hej verden';  
end.
```

## Simula

```
begin  
  text s;  
  s :- "hej verden";  
end
```

I Pascal afsættes 20 pladser til teksten og i den indsættes strengen "hej verden". Man skal altså på oversættelsestidspunktet vide hvor lang strengen maksimalt kan være. I Simula er streng-variablen en hægte til en tekststreng og der sker derfor ikke nogen kopiering. Bemærk at man i Simula bruger notationen for referencetildeling. Faktisk opfattes tekststrengene som tekst-objekter og den tidligere snak om objekt-referencer gælder også her. I eksemplet ovenfor er "s" i virkeligheden en reference til en tekststreng, men i praksis kan man tænke på det som om "s" har værdien "hej verden". Om strengen er gemt i "s" eller et andet sted med en reference fra "s" er så en uinteressant implementations-detajle.

Kopiering. Lad nu "s" og "t" være streng-variable. Hvis vi gerne vil kopiere indholdet af "s" over til "t" kan det ske således:

## Pascal

```
t := s;
```

## Simula

```
t :- s;
```

I Simula sættes "t" blot til at pege på det "s" peger på. Igen sker der altså ikke nogen kopiering i Simula. Til gengæld kan vi så småt ane en forskel. Hvis man kunne gå ind og ændre i det "s" peger på (altså ikke blot lade "s" pege på noget andet) ja så vil det også ændre det "t" peger på.

Delstreng. Lad igen "s" og "t" være streng-variable. Hvis "t" skal være den del af tekststrengen i "s", som starter i position "m" og er "n" tegn lang, så sker det med sætningen "t := copy(s,m,n)" i Pascal og "t :- s.sub(m,n)" i Simula.

## Pascal

```
s := 'ABCDEF';  
t := copy(s,2,3);
```

## Simula

```
s :- "ABCDEF";  
t :- s.sub(2,3);
```

Hvis variabelen "t" udskrives vil det i begge tilfælde resultere i teksten "BCD". Man kan her se at streng-variable i Simula ikke blot er hægter

til tekststreng. Det er (bl.a) også oplysning om længden af strengen. Det gør det muligt at lade “s” og “t” dele en tekststreng idet “s” er 6 tegn fra strengen “ABCDEF” startende i position 1 mens “t” er tre tegn startende i position 2. I Simula kræves der altså igen ikke nogen kopiering, men blot at man giver en ny reference.

Konkatenering. Det der har stået på højresiderne ovenfor kan bruges som værdier i mere komplicerede tekstudtryk. Som operation i sådanne udtryk har man i begge sprog adgang til konkatenering (sammensætning). I Pascal bruges et plustegn (+) og i Simula et ampersand (&) til denne operation

### Pascal

```
s := 'ABCDEF';  
t := 'GH' + s + copy(s,2,3);
```

### Simula

```
s :- "ABCDEF";  
t :- "GH" & s & s.sub(2,3);
```

Herefter vil “t” være tekststrengen “GHABCDEFBCD”. I Simula vil resultatet af en konkatenering være et nyt tekst-objekt som indeholder resultatet af konkatenering. I Pascal skal man på oversættelsestidspunktet være sikker på at erklære “t” så der er plads til den nye streng. I Simula bliver der først afsat plads til strengen på kørselstidspunktet.

Tekst til tegn. I Pascal er der en meget simpel sammenhæng mellem tekst og tegn. Et tegn kan bruges som en tekststreng med længden 1 og et tegn kan hives ud af en tekststreng ved simpel indicering. Helt så let er det ikke i Simula. I Simula er det nærmest som om man skal opfatte en tekststreng som en fil man kan læse fra og skrive til. Det gør det hele en smule mere besværligt.

Vi starter med at hive et tegn ud af en tekststreng. Lad “s” være en streng-variabel, “c” en tegn-variabel (*char/character*), og lad os sige at vi gerne vil have fat i det i’te tegn i strengen.

### Pascal

```
c := s[i];
```

### Simula

```
s.setpos(i);  
c := s.getchar;
```

Tegn til tekst. Hvis vi gerne vil gemme et tegn i en streng er det ganske let i Pascal eftersom tegn-variable kan bruges som streng-udtryk. I Simula er det noget mere indviklet.

## Pascal

```
s := c;
```

## Simula

```
s := blanks(1);  
s.putchar(c);
```

Her er vi så for første gang udsat for en destruktiv operation. Operationen “putchar” vil indsætte og overskrive tegnet i det “s” peger på. Nu har vi gjort det ret harmløst ved først at sætte “s” til at pege på et nyt tekstobjekt bestående af et blanktegn. Set under et sker der altså ikke noget destruktivt her.

I Simula ser det jo nærmest ud som om man læser fra, og skriver til strengen, som om det er en fil. Faktisk er det stort set også hvad der sker. Indlæsning og udskrivning sker gennem buffere og disse buffere er faktisk streng-variable, så når vi læser et tegn ind fra en fil, så læser vi det fra en buffer som blev læst ind med “inimage”.

Initialisering. I Simula har streng-variable initielt den tomme streng som værdi. Dette svarer iøvrigt til konstanten “**notext**”. I Pascal har variable undefinerede værdier ved starten af udførelsen.

Sammenligning. I både Simula og Pascal kan streng-variable sammenlignes ved de sædvanlige relationer. Skriver man således “s = t” er det i Pascal indholdet af “s” og “t” der sammenlignes og i Simula indholdet af det “s” og “t” peger på. Iøvrigt bruges den leksikografiske ordning således at “aca” > “abb”.

Længden af en streng. I Pascal afsættes der statisk en fast plads til tekststrengene, men inden for dette kan længden variere. I Simula er det lige omvendt. Der bliver først afsat plads til tekst-objekter under udførelsen, men når først man har gjort det kan dens længde ikke ændres.

Når en strengvariabel “s” indeholder en streng kan man få oplyst længden af den på følgende måde

## Pascal

```
i := length(s);
```

## Simula

```
i := s.length;
```

Parametre. Streng-variable og tekstudtryk kan bruges som parametre til underprogrammer. Normalt reference-overføres tekster: Der overføres altså en reference til tekst-objektet. Desuden kan man specificere værdioverførsel hvorved der først bliver taget en kopi af tekstobjektet. Holder man sig pænt fra destruktive operationer er det dog overflødigt. Endelig

kan man navneoverføre streng-variable hvis man gerne vil returnere tekster fra et underprogram.

Indlæsning. I Pascal kan man indlæse en linie ved at kalde “readln” med en streng-variabel. Dette kan man også gøre hvis man allerede har læst noget af linien idet operationen så vil indlæse resten af linien. I Simula kan man bruge funktionen “intext” til at læse en linie, men den fungerer kun fornuftigt hvis man ved hvormeget der er tilbage af linien. Simula-systemet vil normalt sørge for at alle inddatalinier fyldes op med blanktegn til en længde på 80 tegn. Hvis man i indlæsningen er ved starten af en linie kan man få fat på en hel linie ved funktionskaldet “intext(80)”. Kaldet returnerer en reference til et text-objekt, og man kan ved “:-” tildele det til en streng-variabel. Vi kan også bruge funktionen til at lave et ganske kort program til at kopiere inddata til uddata:

## Simula

```
begin
  while not endfile do begin
    outtext(intext(80));
    outimage
  end
end
```

Man kan også få fat i den del af en linie hvor de afsluttende blanktegn er fjernet. Det sker ved kaldet “intext(80).strip”, hvor man altså først læser 80 tegn ind og derefter danner et tekst-objekt, hvor afsluttende blanktegn er fjernet. Det skal bemærkes at man herved både får fjernet de blanktegn Simula-systemet tilføjer og evt. blanktegn man selv har tastet sidst på en linie.

**For-sætningen.** Og til slut en lille detalje som nok skal overraske en og anden Simula-programmør. I Simula kan **for**-sætningen også bruges med en kontrol-variabel af reference-type. Det giver så ikke nogen mening at bruge et såkaldt **step-until** element, men man kan have **while**- og liste-elementer med reference-tildeling. Det lyder nok som det rene volapyk, men hvis vi skal skrive et enslydende brev til fire gode venner så kan vi bruge følgende lille program stump.

## Simula

```
begin
  text s;
  for s:- "Petersen", "Poulsen", "Pallesen", "Pil"
  do begin
    outtext("Kære ");
    outtext(s);
    outimage
    ! og nu noget mere tekst ;
  end;
end
```

Det skal altså læses som at variabelen "s" på skift bliver tildelt (en reference til) et af de fire navne og hver gang bliver kroppen af **for**-sætningen udført. Denne form for **for**-sætning kan iøvrigt også bruges for andre reference-typer (**ref**(*klasse*)). Sætningsformen er endnu et udtryk for at man i Simula har gjort et nummer ud af at gøre det naturligt at arbejde med objekter og referencer - bl.a. ved at tillade brugen af objekter og referencer i flest mulige sammenhænge.

**Plads.** Simula og Pascals måder at arbejde med tekststrengene på er ret forskellige, men det er ikke så besværligt at gøre de samme ting. I grundtanken bag ligger også en forskellig brug af lageret. I Simula er det mere naturligt at arbejde med dynamiske objekter og ja, det kan give anledning til spild - altså tekst-objekter man ikke længere har referencer til. Til gengæld har Simula indbygget en automatisk spildopsamling, som gør det muligt at genbruge dette lager om nødvendigt. At flere streng-variable kan deles om tekststrengene v.h.a. referencer kan også betyde at lageret bliver udnyttet mere effektivt.

**Og lidt om resten.** Hermed skulle vi gerne have beskrevet hvad der skal til for at kunne bruge tekststrengene og streng-variable i praksis. Der er dog noget mere til historien - ihvertfald i Simula. I Simula kan man nemlig også gå ind og ændre i tekst-objekter (de tekststrengene som streng-variable peger på). Gør man det vil det så også ændre værdien af andre streng-variable som peger på samme streng. At ændre en streng er let nok; det sker ved at bruge en simpel tildelingsætning. Der er imidlertid den restriktion at man ikke må ændre tekstkonstanter, som optræder i programmet. Man må først tage en kopi af sådanne.

## ulovligt

```
s :- "hej verden";  
s := "god";
```

## Simula

```
s :- copy("hej verden");  
s := "god";
```

Herefter skulle man så tro at "s" pegede på strengen "god", men her skal vi så huske at tekst-objekters længde ikke kan ændres. "s" peger på et objekt af længden 10, og det skal det blive ved med. Nå så kan det jo være at "s" har værdien "god verden". Nej heller ikke. Ved tildeling indsættes fra venstre og resten udfyldes med blanktegn, så værdien er "god ". Man kan så få fat i det delobjekt hvor afsluttende blanktegn er fjernet. Det sker som "s.strip", og det giver altså en hægte til de første tre tegn af det "s" peger på. Men før man bruger det skal man dog have helt check på det. Lad os nemlig sige at vi laver tildelingen "t :- s.strip". Så har "t" rigtig nok værdien "god". Hvis vi herefter laver tildelingen "s := "hejsa alle", ja så har "t" ikke længere værdien "god". Den peger nemlig på de tre første tegn af det "s" peger på, altså "hej". Alt ialt - det kan være nyttigt - men man skal vide hvad man gør før man kaster sig ud i at bruge destruktive opdateringer af tekst-objekter.

**Opsummering** . Vi slutter lige af med en kort opsummering af nogle af de operatører og funktioner man har adgang til,

Konstanter. Konstanter af typen tekst skrives med citationsstegn. Desuden er **notext** den tomme streng "".

Tekst-generatorer. Nye tekst-objekter kan dannes på fire måder

*blanks(antal)*. Funktionen "blanks" tager et heltal som argument og danner et nyt tekst-objekt med det givne antal blanktegn.

*copy(tekst)*. Funktionen "copy" tager et tekstudtryk som argument og danner et nyt tekst-objekt, med det samme indhold.

*intext(antal)*. Funktionen "intext" indlæser et givent antal tegn fra ind-data og returnerer det som et tekstobjekt.

**&**. Operatoren "&" tager to tekstudtryk og danner et nyt tekst-objekt med sammensætningen som indhold.

Metoder. Ved hjælp af punktum-notationen har man adgang til en række funktioner og procedurer som virker på streng-variable. Udover de nedenfor nævnte findes der adskillige andre med mere specialiserede formål.

*sub(position, antal)*. Returner en reference til en delstreng af tekstobjektet.

*length*. Returner længden af tekstobjektet.

*setpos(position)*. Sæt den position i tekstobjektet hvorfra senere indsættelse eller hentning af tegn skal ske.

*getchar*. Hent et tegn fra den aktuelle position i strengen.

*putchar(tegn)*. Indsæt et tegn på den aktuelle position i strengen

*strip*. Returner en reference til den delstreng som ikke indholder afsluttende blanktegn.

[Ydreligere om tekststrengene i *HBH*: pp 53-55, 205-234]

## 2.8 Konklusion

Nu er vi nået igennem de dele af Pascal og Simula som kan sammenlignes direkte. Tilbage er de objekt-orienterede faciliteter i Simula. En diskussion af dette ligger imidlertid uden for denne notes ramme så vi vil stoppe behandlingen af sprogene her.

Det vil måske her være naturligt at sige noget om hvilket sprog der er bedst, pænest, mest effektivt, el.lign. Det vil jeg undlade. Hvis man skal se på hvad der gør et sprog til en succes er det nemlig sjældent sådanne forhold som spiller ind. Pascal er en succes - ikke nogen tvivl om det - men her 25 år senere kan det ikke forklares med at det er det bedste sprog til sit formål. Sprogets skaber Niklaus Wirth har senere designet nogle sprog hvor forhold han fandt u hensigtsmæssige i Pascal er blevet forbedret. Disse sprog er imidlertid ikke blevet en succes i noget nær samme omfang som Pascal. Pascals succes idag skyldes nok snarere Turbo Pascal oversætteren og udbredelsen af pc'ere.

Simula var en succes. Det var et sprog med stor indflydelse og nogen udbredelse i slutningen af 60'erne og starten af 70'erne. Et sprog som C++ har siden nået en væsentlig større udbredelse inden for objekt-orienteret programmering. Det er dog ikke klart at det skulle sige noget om hvorvidt det ene sprog er bedre end det andet.

Skal jeg forsøge en konklusion vil det nok være at med nært beslægtede sprog spiller det konkrete valg af sprog til en given programmeringsopgave en forbløffende lille rolle. Ja, der er forskelle mellem Simula og Pascal, men et problem kan løses med næsten identiske programmer i de to sprog. Hvis man programmerer en del er det let at blive grebet af sit daglige programmeringssprog. Man kan blive "fan" af det - man kan nå dertil, hvor man hellere helt vil holde op med at programmere end at skulle programmere i et andet sprog. Sådanne holdninger kan måske være naturlige nok, men den faktiske grund skyldes sjældent sprogenes egentlige kvaliteter.

### 3 Litteraturliste

Borland, *Turbo Pascal Reference guide v 5.5*, Borland International, 1989.

H B Hansen, *Simula, Et objektorienteret programmeringssprog*, Roskilde Universitetscenter, 1982.

Hewlett Packard, *HP Pascal Language Reference*, Hewlett Packard, 1991.

P Holm & M Taube, *SIMDEB User's Guide for Unix*, Lund Software House, 1987.

P Holm, *SIMULA User's Guide for Unix*, Lund Software House, 1987.

K Jensen & N Wirth, *Pascal User manual and report*, Springer Verlag, 1978.

K Nygaard & O-J Dahl, *The Development of the Simula Language*, History of Programming Languages, pp. 439-493, Academic Press, 1981.

Simula a.s., *SIMULA(R) Programmers Reference Manual*, Simula a.s, Oslo, Norway 1989.

N Wirth, *Recollections about the Development of Pascal*, ACM Sigplan Notices vol 28 no 3, March 1993.