1
Grundbegreber.

Stilarter i programmering og sprog.

Syntaks og semantik.

Datatyper.

Kontrolstrukturer.

Udtryk.

Abstraktioner.

Parametermekanismer.

Blokke og navnebindinger.

Scope og scoperegler.

PS1 -- Grundbegreber

Noter

Parallelle kontra sekventielle programmeringssprog.

- **Parallelle sprog.**

  Understøtter samtidig afvikling af flere processer
  med hver deres kontrolforløb.

  - Inter-processkommunikation
  - Synkronisering

- **Sekventielle sprog.**

  Understøtter kun eet kontrolforløb.

Noter

På Programmeringssystemer 1 vil vi udelukkende beskæftige os med sekventielle sprog.

Vi indleder dette kursus og denne forelæsning, med et kort overblik over forskellige stilarter inden for sprog og programmering i disse sprog. Den mest overordnede skelnen er parallelle sprog kontra sekventielle sprog.

Der er mange gode grunde til at være optaget af sprog, der understøtter parallellitet. Mange aktiviteter omkring os foregår samtidigt, og en vigtig egenskab ved programmeringssprog er *på en naturlig måde* at kunne modellere verden omkring os.

Beslægtet med ovenstående er tendensen til distribution af datamaskiner og deraf følgende behov for, at forskellige, distribuerede programmer kommunikerer og synkroniserer deres indenbyrdes forløb.

Mere om parallelitet på kurset maskinel 2 på dat 2.

Stilarter: Programmering og programmeringssprog.

Imperative sprog

- Programmering med assignments og procedurer, begge med sideeffekter.
- Eks.: Fortran, Algol, Pascal, Modula, Ada

Funktionelle sprog

- Programmering med udtryk og funktioner uden sideeffekter.
- Eks: Ren Lisp, ML, Miranda.

Objekt-orienterede sprog.

- Programmering med abstrakte datatyper, 'information hiding' og nedarvning.
- Eks.: Simula, Smalltalk, Beta, CLOS, C++, Eiffel.

Logiksprog.

- Programmering med facts, if-then regler og logiske slutninger ud fra disse.
- Eks.: Prolog.

Noter

Imperative programmeringssprog er langt de mest almindeligt forekommende i praktisk brug. Et imperativt program under udførelse er karakteriseret af en *tilstand* (state), som ændres over tid i takt med, at programmet udføres. Assignmentsætningen er det vigtigste primitiv, som kan ændre tilstanden af et program. Studerende i Programmeringssystemer 1 antages at have en baggrund i imperativ programmering (i Pascal). Nærværende (og næste) forelæsning dækker hovedsageligt begreber og teknikker knyttet til imperative sprog.

Funktionel programmering er relateret til den matematiske forståelse af funktioner (og lambda calculus). Programmer skrevet i funktionelle sprog er meget lettere at analysere end imperative programmer. Man kan sige, at et funktionelt program har en tilstand, men denne tilstand ændres *ikke* i takt med programudførelsen.

Objekt-orienteret programmering tager udgangspunkt i ideen om abstrakte datatyper. Objekter er instanser af abstrakte datatyper, og objekter kan indeholde tilstande, som ændres over tid. Dette er et lighedspunkt i forhold til imperative sprog. Kontrol over synlighed af data samt udvidelse af abstrakte datatyper ved nedarvning er vigtige karakteristika. Store dele af Programmeringssystemer 1 vil handle om objekt-orienteret progammering.

Logik programmering består i at stille spørgsmål til en database af facts givet nogle inferensregler. Logikprogrammering opfattes af de fleste som meget forskellig fra de tre andre stilarter.

Syntaks og semantik.

```
if a < b
then P(a, b)
else Q(a, b)
```

	Uformel beskrivelse	*Formaliseret beskrivelse*
Syntaks • Beskrivelse af *form*.	En if-sætning består af et udtryk og to sætninger.	if_statement ::= if <expr> then <statement-1> else <statement-2>
Semantik • Beskrivelse af *mening*.	Hvis udtrykket evalueres til sand udføres statement-1 ellers udføres statement-2	

Noter

Det er vigtigt at kunne skelne mellem syntaks og semanik, altså form og mening.

Syntaks kan beskrives på flere niveauer:

Det laveste niveau handler om, hvordan man kan beskrive reglerne for dannelse af nøgleord, navne, tal, kommentarer og andre *leksikalske enheder*.

Det mellemste niveau kaldes ofte beskrivelsen af *konkret syntaks*. Beskrivelsen et sprogs konkrete syntaks munder ud i en hierarkisk dekomponering af et programs leksikalske enheder.

Abstrakt syntaks, det højeste syntaktiske niveau, giver ligeledes en hierarkisk dekomponering af et program, som er mere velegnet som udgangspunkt for beskrivelsen af af semantikken af programmet.

Syntaksregler beskrives som regel af en *grammatik*. Grammatikker kan udtrykkes på mange forskellige måder. Grafiske *syntaksdiagrammer* er kendte for læserne. Tekstuelle grammatikker, udtryk i den såkaldte BNF (Backus Naur Form eller Backus Normal Form) er også meget udbredte. I bogen *Object Oriented Software Construction* er der er et kort afsnit af syntaks notation (efter forordet), som afspejler den måde, syntaktiske elementer vil bliver beskrevet på i bogen.

Semantiske regler er vanskeligere at beskrive præcist og matematisk formelt. Ovenfor har vi ikke nævnt nogen specifik teknik til formaliseret beskrivelse af semantik. I praksis beskrives semantikken for det meste uformelt, eller i bedste fald som en blanding af uformel prosa og formelle definitioner. *Formel semantik* er, på trods af dette, en vigtig del af faget. Mere om dette på dat2.

En grammatik for Eiffel er vist i appendix C af OOSC; I appendix F af samme bog er vist syntaksdiagrammer, for de læsere, der foretrækker sådanne. Tag et kig på disse, og sammenlign de to formalismer.

Datatyper.

Simple, sprogdefinerede typer.

- Real, integer, boolean

Sprogdefinerede type-konstruktorer.

- Array, record, set, file

Bruger-definerede typer.

- **Type** my-type = *typeudtryk.*

Statisk typing: Typen af ethvert udtryk i et program kan bestemmes inden programmet udføres.

Noter

Alle programmeringssprog understøtter data af primitive såvel som sammensatte typer. Med indførelsen af Pascal (først i 70erne) blev det også almindeligt, at brugeren selv kunne definere navngivne typer.

Der er altid naturlige operationer tilknyttet en type. F.eks. er de aritmetiske operationer definerede på integer (og andre er tilsvarende defineret på real); og selektion i arrays a[i] er en operation på en array type. På brugerdefinerede typer er disse operationer tilknyttet på en mere løs facon til typerne, og dermed til variablene og udtrykkene af disse typer.

Ovenstående observationer er motivationen for en tættere tilknytning mellem en type og naturlige operationer på værdier af typen. Dette førte til udviklingen af begrebet **abstrakte datatyper**, som vi vender kraftigt tilbage til lidt senere i kurset.

Statisk typing er et grundelement i Pascal og mange andre, lignende sprog.

En hel anden problematik om typer er, hvilke typer ligner hinanden så meget, at variable af disse typer kan erstatte hinanden i de udtryk, hvor de indgår. Dette er typeækvivalens eller typesammenlignelighedsproblemet, som vi ikke vil dyrke eksplicit på dette kursus. Vi vender dog konkret tilbage til problematikken i forbindelse med Eiffel.

Arrays og records.

Arrays

a: **array**[index_type] **of** element_type

- Homogen tabel.
- Repræsenterer en funktion: index_type -> element-type
- Element access: a[j]
- Indekset j beregnes under programudførelsen.

Record

- Heterogen tabel.
- Repræsenterer et Cartetisk produkt: T1 x T2 x T3.
- Element access: r.felt2
- Feltnavnet er statisk (kan *ikke* beregnes under programudførelsen).
- variant-records:
  - Variation i typen -- samme typenavn.
  - Lager-økonomi: delt lagring af varianter.

```
r:  record
      felt1:  T1;
      felt2:  T2;
      felt3:  T3
    end
```

```
v:  record
      felt1:  T1;
      felt2:  T2;
      case felt3:  T3 of
        variant-1:
          (felt-4: T4);
        variant-2:
          (felt-5: T5)
    end
```

PS1 -- Grundbegreber

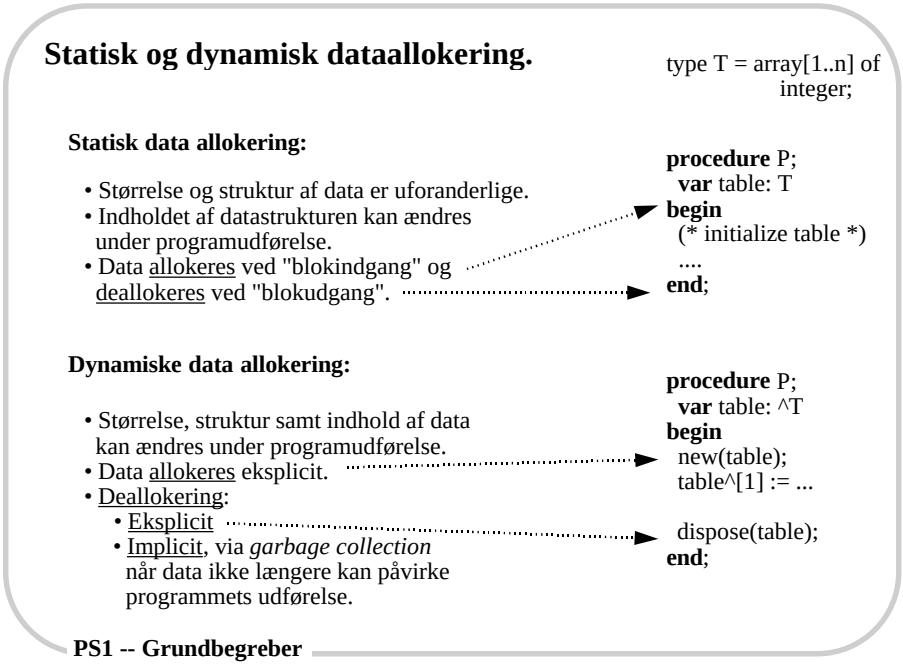
Noter

Denne slide opregner på kort form nogle vigtige egenskaber ved to vigtige typekonstruktorer fra Pascal.

Det er vigtigt at bide mærke i, at både arrays og record giver *direkte tilgang* til data i tabellerne. Underliggende betyder dette, at positionen af data i tabellen beregnes ved element access, og data slåes derved direkte op uden nogen form for tidskrævende søgning.

Variant records giver programmøren mulighed for at beskrive specialiseringer af recorden i to eller flere retninger, uden det giver typesammenlignelighedsproblemer (uanset hvilken specialisering der konkret lagres, er typen den samme). Vi vil på dette kursus bevæge os meget længere, end variant records tillader med hensyn til typespecialisering. Mere om dette under forelæsningerne om nedarvning.

Opgave: Som beskrevet ovenfor kan indekset j i a[j] være et udtryk, hvis værdi beregnes på programmets udførelsestidspunkt. Eksempelvis kan vi skrive x:= a[f(i)] hvorved array-elementet, der tilgåes, kommer til at afhænge af funktionen f og værdien af i. Diskutuer hvorfor man ikke i Pascal kan skrive x := r.f(...), hvor f er en funktion, som vi ønsker skulle returnere et feltnavn i recorden r.



Noter

I mange sammenhænge er det ikke fleksibelt nok, at alle datastrukturer allokeres af fast størrelse og struktur. Ofte ønsker vi at kunne udvide vore strukturer i takt med programudførelsen. I Pascal kan man dynamisk allokere data med proceduren new. Dynamisk allokerede data refereres via såkaldte pointere.

Til gengæld for øget fleksibilitet af programmet under udførelsen skaber brugen af dynamiske data og deraf følgende brug af pointere ofte programmeringsmæssige komplikationer. Det er simplethen vanskeligere at holde styr på dynamisk allokerede data, end det er tilfældet med statisk allokerede data.

Eksplicit allokering af dynamiske data er uundgåelig (på et eller andet programmeringsniveau) hvis man ønsker fleksibiliteten. Derimod kan deallokeringen af lager gøres automatisk via såkaldt *garbage collection.* Garbage collection genopsamler lagerceller, som indgår i data, der ikke længere på nogen måde kan påvirke forløbet af programmets udførelse (data kan ikke længere "nåes" fra det kørende program).

Det forventes ikke, at alle kursusdeltagere har programmeringserfaring med dynamiske data. Dynamiske data er reglen og statiske data undtagelsen i Eiffel. Derfor får man i løbet af Programmeringssystemer 1 stor erfaring i håndtering af dynamiske data. Heldigvis har Eiffel garbage collection, så vi behøver blot at bekymre os om allokering.

Kommandoer og kontrolstrukturer.

Kommandoer kan ændre programmets tilstand.

Assignment

var := expr

• Den vigtigste og mest karakteristiske primitive kommando i imperative sprog.

Procedurekald

P(x + y, z)

aktuelle parameterudtryk som evalueres til *argumenter*

• Sprogdefinerede og primitive
• Brugerdefinerede

Kontrolstrukturer

• Sekventielle
• Selektive
• Iterative

PS1 -- Grundbegreber

Noter

Der anvendes forskellig terminologi, for det vi her kalder kommadoer: sætninger (statements) og intruktioner.

Vi har allerede behandlet rollen af assignments, da vi tidligere diskuterede forskellige programmeringsstilarter.

Procedurekald kan enten referere til procedurer, som er defineret af sproget eller af brugeren. I moderne sprog er der en tendens til, at færre og færre procedurer defineres som en del af sproget. I sådanne sprog findes der et sæt af biblioteker, hvori samlinger af beslægtede procedurer defineres. Dette vil blive et grundtema på dette kursus.

Sprogdefinerede procedurer er undertiden specielle i forhold til de procedurer, som brugeren selv kan definere. F.eks. kan visse sprogdefinerede procedurer i Pascal tage et vilkårligt antal parametre.

Formålet med kontrolstrukturer i imperative programmer kan---lidt poppet---siges at være, at blande assignments på en passende, kontrollerbar og overskuelig måde. I tidernes morgen brugte man ikke kontrolstrukturer, men derimod eksplicitte og ukontrollerbare hop.

Den mest fundamentale kontrolmekanisme i imperative sprog er sekventiel sammensætning: Først udføres een kommando **;** (semikolon) dernæst en anden kommando.

Selektion.

Udvælgelse af een kommando blandt flere mulige

- If-then-else
  - Indlejrede if-then-else:
    sekventiel beregning og udvælgelse.

if *boolean-expression-1*
then *command-1*
else if *boolean-expression-2*
then *command-2*
else *command-3*

- Case
  - Tabelopslag på basis af selector-værdi.
  - Udtømmende opremsning af "cases".

case *selector-expression* **of**
  case-constant-1: *command-1*;
  case-constant-2: *command-2*;
  case-constant-3: *command-3*;
end

Noter

Pascal's if-then-else struktur udvælger i udgangspunktet blandt to muligheder på basis af et boolsk (sandt el. falsk) udtryk. Ved indlejring af en ny if-then-else i else-delen af den første (og ved passende typografisk indrykning af programteksten), kan vi dog opnå en tilsvarende effekt som ved **cond**, kendt bl.a. fra Lisp.

Mere moderne sprog udvider typisk if-then-else sætningen til denne sekventielle beregning af boolske udtryk, og efterfølgende udvælgelse af en (sekvens af) kommando(er).

Case sætningen er mere begrænset, men også mere effektiv end multivejs udvælgelse ved sekventiel beregning af boolske udtryk. Alle case-constanter skal, som navnet antyder, være bestemmelige på oversættelsestidspunkt, og derfor kan selektionen foregå ved direkte hop til den ønskede kommando på baggrund af selektorens beregnede værdi.

Bemærk at case-sætningen og "case-records" (variant records, som vi behandlede tidligere i dette kapitel) ikke har noget med hinanden at gøre.

Iteration.

Ubestemt antal gentagelser:

- **while**
  - test inden gentagelse
  - nul, en eller flere udførelser af kommando

while *expression*
do *command*

- **repeat**
  - test efter gentagelse
  - mindst een udførelse af kommandoer

repeat
command-sequence
until *expression*

Bestemt antal gentagelser:

- **for**

for i := first **to** last **do**
command

PS1 -- Grundbegreber

Noter

Man skulle måske tro, at iterative kontrolstrukturer er lige så vigtige og fundamentale som de selektive. Det er de ikke. Enhver iteration kan nemlig programmeres ved brug af rekursive procedurer.

Programmering med iterative strukturer med et (på forhånd) ubestemt antal gennemløb er svære at styre. En sådan iterativ struktur kan afstedkomme en uendelig løkke; og ofte er der randproblemer, dvs. problemer med enten det første eller det sidste gennemløb. Brug derfor for-sætninger, hvor det er muligt!

I dette kursus vil vi se på tiltag, der hjælper programmøren med at styre sådanne "svære" iterative strukturer: invariant og variant. Vi vender tilbage til dette i forelæsningen: "Udtryk og kommandoer i Eiffel".

Det er ofte mere hensigtsmæssigt at tilknytte iteration til datastrukturer end omvendt. Dette lyder måske lidt uforståeligt på nuværende tidspunkt, men vi vender tilbage til problemstillingen i forelæsningen om "Arrays og lister".

Udtryk.

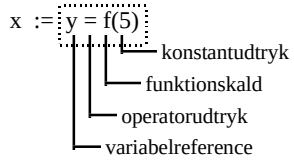
Et udtryk kan*beregnes* hvorved der fremkommer en *værdi.*

Beregning af et udtryk ændrer ikke på programmets tilstand.

Forskellige former for udtryk:

- Variabelreference.
- Konstantudtryk.
- Funktionskald og operatorudtryk.

Eksempel i Pascal:



Tilsvarende eksempel i C:

```
x = y == f(5)
```


Noter

Der er en vigtig begrebsmæssig forskel mellem kommandoer og udtryk. Som allerede omtalt kan udførelsen af en kommando ændre programmets tilstand, men kommandoen resulterer ikke i nogen værdi. Omvendt forholder det sig med et udtryk, idet udførelsen af et udtryk altid efterlader programmets tilstand uændret, men det giver en værdi.

Nogle sprog, såsom C og Lisp, forplumrer denne begrebsmæssige forskel. I mange sprog har et assignment såvel som et procedurekald en værdi, og disse kan således anvendes som udtryk. I C kan man således skrive

```
if (a = b) ...
```

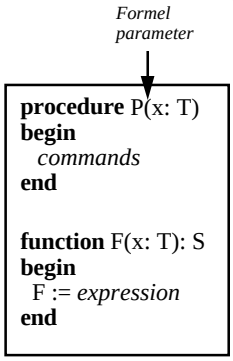
hvor a = b er et assignment hvis værdi er værdien af variablen b. Dette er dårlig programmeringsstil, som bør undgåes.

Det bør også nævnes at mange sprog tillader en funktion at have sideeffekter, således at et kald af funktionen (et udtryk) også spiller rollen som en kommando. Det er yderst vanskeligt for compilere at opdage dette i imperative sprog; det anbefales at pålægge sig selv disciplinen aldrig at lave sideeffekter i funktioner.

Også hvad angår terminologi og syntaks skal man være agtpågiven. I C og Lisp taler man udelukkende om “fuktioner” selv om mange af disse er procedurer (funktionskaldet er en kommand). Og i C ser et assignment (ala “a = b”) ud som et Pascal operatorudtryk. I resten af dette kursus vil vi konsekvent notere et assignment med := symbolet. Formen “a = b” vil betegne et operatorudtryk. Brug af = som assignment (eller := som lighedsoperator) vil blive opfattet som groft sløseri samt mangel på forståelse af ovennævnte vigtige skelnen mellem udtryk og kommandoer.

Abstraktioner over kommandoer og udtryk.

- En abstraktion over en kommando er en *procedure*.
- En abtraktion over et udtryk er en *funktion*.



En definition af en abstraktion består af

- Et navn for abstraktionen
- Et antal parametre, som generaliserer abstraktionen.
- Den abstraherede kommando/udtryk/...

Primær motivation:

Højne programmets abstraktionsniveau, bedre læselighed og forståelighed.

Sekundær motivation:

At identificere programfragmenter, som bruges mere end eet sted, gøre disse til en procedure, som kaldes to eller flere gange.

Noter

Abstraktioner er et af de absolut centrale elementer i programmeringssprog; tilsvarende er passende brug af abstraktioner i programmering et af de suverænt vigtigste aspekter i "god programmeringsstil".

Abstraktioner er ikke begrænsede til at virke på kommandoer og udtryk. Man kan principielt lave abstraktioner over "hvad som helst", med varierende nytteværdi, naturligvis.

Funktioner a la Pascal begrænser ikke programmøren til kun at skrive eet udtryk i kroppen af funktionen. Man kan skrive vilkårlige kommandoer, men det kræves, at der returneres et resultat (ved assignment til funktionsnavnet). Det opfattes dog som dårlig stil, hvis en funktion har sideeffekter.

Vi vil dyrke objekt-orienteret programmering på dette kursus. Den helt centrale, programmeringssproglige konstruktion er i denne sammenhæng en klasse. Det er interessant at vide, at klasser faktisk er abstraktioner over definitioner. Så vi kunne således kalde en klasse for en definitions-procedure.

Parametermekanismer.

- 'Call by value'
  - En *kopi* af a overføres til P.
  - Kopien bindes til den formelle parameter x.
  - Eventuelle assignments til x har kun lokale virkninger
- 'Call by reference'
  - Der overføres en reference (adresse) på a til P.
  - I kroppen af abstraktionen er x et alternativt navn (et *alias*) på variablen a.
  - Eventuelle assingments til x har effekt uden for proceduren.
  - Kun udtryk, hvortil der kan 'assignes', kan benyttes som var-parametre til en procedure.

```
procedure P(x: T)
begin
  commands
end

a: T;
P(a)
```

```
procedure P(var x: T)
begin
  commands
end

a: T;
P(a)
```


Noter

De to nævnte parametermekanismer er centrale, og det er meget vigtigt at forstå forskellen mellem dem.

I forskellige sprog findes der forskellige variationer, såsom 'value result' og 'name' parametre. Vi kommer ikke ind på disse i "Programmeringssystemer 1".

Forskellen mellem 'value' og 'reference' parametre blødes lidt op, hvis vi overfører pointere (referencer) som parametre. Vi vender tilbage til dette i forbindelse med parametermekanismen i Eiffel.

I Pascal kan man også overføre procedurer og funktioner som parametre.

Opgave. Dette kapitel har været inspireret af Pascal. Emnerne fra dette kapitel kan naturligvis også anskues fra andre programmeringssprog, f.eks. programmeringssproget Lisp, som jo har været berørt på Basisuddannelsen.

Hvilken programmeingsstilart synes I Lisp understøtter? Hvordan er den syntaktiske struktur af Lisp? Hvilke datatyper har I kendskab til i Lisp? Er der statisk typing? Er der arrays og records? Hvordan laver man assignment? Hvordan virker cond i forhold til if-then-else kommandoen. Hvilke iterative kontrolstrukturer har I kendskab til i Lisp? Kan man forestille sig andre abstraktioner end procedurer og funktioner i Lisp? Hvilke parametermekanismer understøttes af Lisp.

Blokke og navnebindinger.

En *blok* er en eller flere kommandoer tilknyttet nogle *navnebindinger*.

- I Pascal er blok-begrebet tilknyttet hovedprogrammet, procedurer og funktioner.
- I andre sprog kan enhver kommando være en blok.
- Blokke kan indlejres i vilkårlig dybde.

Eksempel på en blok i Pascal:

```
const n = 10;
type table = array[1 .. n] of integer;
var a: table
      i: 1 .. n
begin
  for i := 1 to n do a[i] := i;
  ...
end
```

En **navnebinding** er associationen mellem et navn og en værdi.

En navnebinding er *eksplicit,* hvis der findes en definition i programmet, som introducerer navnet og dets værdi.

PS1 -- Grundbegreber

Noter

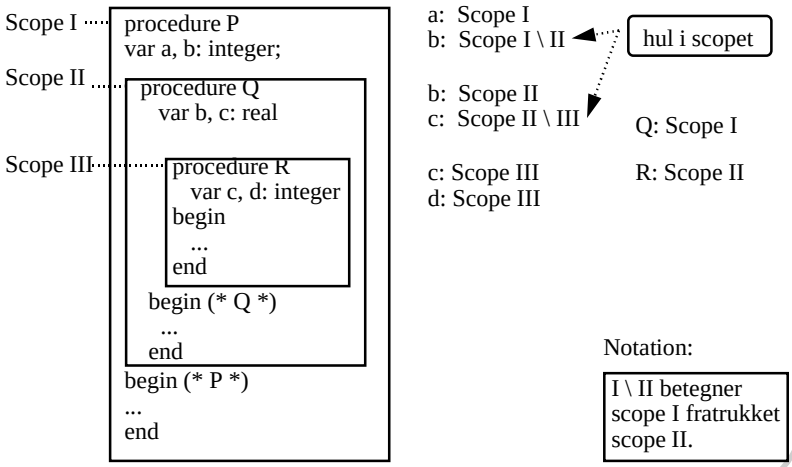
Eksplicit definition af navne og deres værdier er et meget værdifuldt element i moderne programmeringssprog. Modstykket til dette er implicit definition af navne, hvor et navn bliver introduceret første gang det optræder (første gang det bliver anvendt).

En navnebinding (eller blot en binding) knytter et navn til en værdi. Constant og type definitioner i Pascal er gode eksempler på navnebindinger. Variabel erklæringer er også eksempler herpå. Værdien er her en plads i lageret (også kaldet en *location*), ikke den værdi som er tilskrevet variablen gennem et assignment.

Navne, der forekommer (på "venstre-siden") i en definition, kaldes undertiden for *definerende navneforekomster*. Definerende navneforekomster er gode at opsøge, hvis man vil vide noget om navnets egenskaber og rolle i programmet. Navneforekomster, der ikke er definerende, er *anvendte forekomster*.

Scope og scoperegler.

Scopet af en navnebinding er det område af programmet (programteksten) hvor bindingen har effekt.



PS1 -- Grundbegreber

Noter

Scoperegler varierer naturligvis fra sprog til sprog.

Selv om scoperegler er vigtige i alle sprog, er det af største betydning, at man fæstner sig ved scopereglerne ved sprog, der lægger op til en dyb indlejring af blokke i hinanden. Dette er tilfældet i Pascal, hvor procedurer og funktioner kan indlejres i hinanden.

I Eiffel, som er det sprog vi vil benytte på Programmeringssystemer 1, er det kun i mindre omfang muligt at indlejre definitioner i hinanden, og sproget har tillige nogle ret konservative konventioner med hensyn til navnesammenfald mellem scopeniveauer.