Fortran 90/95

Dieter Britz

Kemisk Institut
Aarhus Universitet

3. Udgave, Oktober 2009

Indhold

Forord **4**

1 Basis **5**
1.1 Et simpelt Fortranprogram . . . 5
1.2 De fysiske rammer . . . 5
1.3 Software-rammer . . . 6
1.4 Grundreglerne . . . 6
1.5 Typer: variabler og konstanter . . . 7

2 Sætninger **10**
2.1 Generelt . . . 10
2.2 Erklæringer . . . 10
2.3 Tilordningssætninger . . . 11
2.4 IF, CASE . . . 13
2.5 Løkker (DO-sætning) . . . 15
2.6 Simpel I/O (print, read) . . . 18
2.7 STOP . . . 19

3 Arrays **20**
3.1 Definitioner . . . 20
3.2 Erklæringer af arrays . . . 20
3.3 Fysiske og aktuelle længder, elementorden . . . 21
3.4 Arraykonstanter . . . 21
3.5 Arrayudsnit . . . 22
3.6 Arraysætninger og –operationer . . . 22
3.7 Input/output af arrays . . . 23

4 Procedurer (underprogrammer) **25**
4.1 Statement functions . . . 25
4.2 Functions . . . 26
4.3 Subroutines – underprogrammer . . . 27
4.4 Argumentoverførsel: arrays . . . 27
4.5 Argumentoverførsel: tegnstrenge . . . 30
4.6 Et nyttigt eksempel: underprogrammet MATOUT . . . 30
4.7 Argumentoverførsel: procedurer . . . 31
4.8 Interne *vs* eksterne procedurer . . . 31
4.9 Modules . . . 31
4.10 Biblioteker . . . 32
4.11 Indbyggede (intrinsic) procedurer . . . 33

5 I/O formats og filer **35**
5.1 Formats . . . 35
5.2 Filer . . . 41
5.3 I/O sætninger . . . 46

6 KIND **49**
6.1 Hvad er KIND? . . . 49
6.2 Praktisk brug af KIND . . . 50
6.3 KIND i intrinsic procedurer . . . 52
6.4 Andre procedurer associeret med KIND . . . 52

7 Mere om arrays **53**
7.1 Indeksvektorer . . . 53
7.2 Reshape . . . 53
7.3 WHERE . . . 54
7.4 Dynamiske arrays . . . 55
7.5 Array intrinsics . . . 56

8 Mere om procedurer **59**
8.1 Nogle definitioner og klassificeringer . . . 59
8.2 SAVE . . . 59
8.3 RESULT . . . 60
8.4 INTENT . . . 60
8.5 Explicit interface . . . 61
8.6 OPTIONAL, brug af keywords . . . 62
8.7 Overførsel af arrays til procedurer . . . 63
8.8 PUBLIC og PRIVATE . . . 66
8.9 Array-valued functions . . . 66
8.10 Character-valued functions . . . 67
8.11 Dummyprocedurer . . . 67
8.12 Rekursion . . . 68

9 Generiske procedurer og definerede operatorer **71**
9.1 Generiske procedurer . . . 71
9.2 Definerede operatorer . . . 72

10 Derived types og strukturer **74**
10.1 Indledning . . . 74
10.2 Input-Output . . . 74
10.3 SEQUENCE . . . 75
10.4 Type constructors . . . 75
10.5 Strukturer og procedurer . . . 75
10.6 Den tredje overload: defined assignment . . . 77
10.7 Eksemplet udvidet . . . 78

11 Pegepinde og datastrukturer **81**
11.1 Pegepinde . . . 81
11.2 Pegepinde, arrays og strukturer . . . 82
11.3 Pegepinde og dynamiske datastrukturer . . . 86
11.4 Input/output med dynamiske strukturer . . . 92

12 Fossiler og andet **93**
12.1 Brugbare gamle faciliteter . . . 93
12.2 Ikke-anbefalede gamle og nye faciliteter . . . 93
12.3 Helt frarådede gamle faciliteter . . . 95

13 Fortran 95 **97**
13.1 Slettede sprogelementer . . . 97
13.2 Nyligt forældede sprogelementer . . . 97
13.3 Helt nye sprogelementer . . . 97
13.4 Nye tiltag i bestående sprogelementer . . . 99
13.5 Andre mindre tiltag . . . 100

14 Bitmønstre, afrunding, programdesign **102**
14.1 Bits, bytes, words . . . 102
14.2 Binære tal . . . 102
14.3 Lagring af de fire grundtyper i rigtige maskiner 103
14.4 Afrunding . . . 104
14.5 Valg af udtryk . . . 105
14.6 Valg af algoritme . . . 106

A Intrinsics **107**
A.1 Fortran 90 intrinsics . . . 107
A.2 Fortran 95 intrinsics . . . 117

B ASCII-tabel **119**
B.1 Kontrolkoderne . . . 119
B.2 Tastaturtegn . . . 119

Stikord **120**

Forord

Denne bog er en reinkarnation af bogen, som IDG udgav i 1999, men besluttede at droppe i 2002.

I bogen henvises der et antal gange til Standarden. Programmeringssproget Fortran 90 er nemlig fuldt ud specificeret i et officielt dokument, ISO/IEC 1539, "Information technology – Programming languages – Fortran". Det er dette dokument, der menes, samt Standarden for Fortran 95, ISO/IEC 1539-1:1997.

Fortran har, siden sin begyndelse i '50'erne, oplevet et antal ændringer, som endte med Fortran 90 i ca. 1990 og Fortran 95 i 1997. Den sidste er udgivet i dokumentet ISO/IEC 1539-1:1997. Fortran 95 er en mindre ændring af Fortran 90, og jeg har givet et kort resumé af ændringerne i kap. 13. Det er ikke alle steder endnu, hvor man har en oversætter til Fortran 95.

Bogen er konstrueret sådan, at man ret hurtigt kan lære sig de mest vigtige grundbegreber ud fra de første 5-6 kapitler. Når alt dette er på plads, kan det være, at resten bliver interessant, hvor der beskrives de mere avancerede muligheder Fortran 90 tilbyder.

Der er ingen opgaver. Min erfaring er, at opgaver i sådanne bøger ikke bliver brugt meget; de plejer at være hentet fra anvendelsesområder, der er forskellige fra det, man selv interesserer sig for. Det kan givetvis også siges om de eksempler, der er i bogen – men eksempler skal der jo være.

Datalogiens sprog er domineret af engelsk, og selvom der er gode danske udtryk for mange begreber, kan det i visse tilfælde virke lidt kunstigt, og ligefrem forvirrende, at prøve at finde et dansk ord. Således har jeg valgt at bruge "array", eftersom "række" ikke rækker. Ordet "module" bliver ligeledes brugt; her kunne man tænke sig at bruge det danske "modul"; men det engelske "module" betegner den nye facilitet i Fortran 90, hvorimod "modul" leder tankerne i retningen af, at et Fortran-program er (eller kan være) modulært, dvs. består af uafhængige enheder, hvilket intet har med "modules" at gøre. Der er også det lidt grimme udtryk "den intrinsic function", som er bedre end f.eks. "den indbyggede function"; og selve "function" kan næppe erstattes med "funktion" i denne sammenhæng. Det samme gælder "kind", som også er en ny Fortran-konstruktion, der heller ikke bliver oversat. Jeg bruger engelsk flertal (f.eks. "arrays") og undgår mest danske former med disse ord ("functionen"). Alt dette resulterer sommetider i lidt grimme, men mere præcise, formuleringer.

Alle de komplette eksempelprogrammer i bogen, og de mere vigtige programstumper, kan ses og downloades, ved `http://www.chem.au.dk/db~/f9095`. Eksemplerne afviger fra dem i bogen kun ved at de har de sædvanlige lange linjer i stedet for de ret korte, bogens layout tvang mig til. Alle færdige programmer er afprøvet; hvilket giver en vis garanti (men ikke 100%!) for, at de er korrekte. Jeg vil gerne vide om eventuelle fejl, bedst per email til mig, `britz@chem.au.dk`. Yderligere oplysninger om Fortran kan fås fra `http://www.fortran.com/fortran/market.html`, hvor man kan finde alt om litteratur, software, en FAQ, m.m. Nyhedsgruppen `comp.lang.fortran` er god. Har man et specifikt problem og spørger i den gruppe, får man som regel svar samme dag. Det er også her, man får nyheder at vide om sproget, nye lærebøger, oversættere, osv.

Dieter Britz, Århus, august 2006.

Editering Oktober 2009: Der blev tilføjet en bemærkning til beskrivelsen af funktionen `CMPLX`.

Kapitel 1
Basis

1.1 Et simpelt Fortranprogram

Her er et simpelt Fortran-program som viser en hel del, selv hvis man ikke ved noget om sproget endnu:

```
program SIMPEL_DEMO
! Et simpelt demoprogram.

  implicit none       ! Forklares senere
  integer :: a, b

  print *, " A og B?"
  read *, a, b
  print *, " a + b =", a+b

end program SIMPEL_DEMO
```

Det er ikke svært at se, at dette program indlæser to heltal (på engelsk: integers), og printer deres sum ud. Vi gennemgår her de enkelte linjer: Den første linje er det såkaldte **programhoved**. Symbolet **program** er et Fortran-ord, mens 'SIMPLE_DEMO' er det **programnavn**, programmøren selv har valgt for at mærke eller identificere dette program. Denne linje er obligatorisk i hvert program. Den skal parres med den sidste programlinje, med symbolet **end**; i dette tilfælde er resten af linjen valgfri, dvs., linjen

```
end
```

ville også være i orden. Men for overskuelighedens skyld er det en god ide at gentage programmets navn her. I så fald skal navnet være præcis det samme begge steder (hoved- og end-linjen).

Disse to linjer, den første og sidste, så at sige indrammer programmet.

Den anden linje starter med '!' og det siger, at linjen er en **kommentar**; dette vil sige, at linjen kun er for programmøren eller enhver anden der læser programteksten. Computeren ignorerer al tekst der følger efter udråbstegnet på en linje. Linje fire viser en kommentar, der står ved enden af en linje, der ellers indeholder noget fortransk. Lad det her blot være sagt at den linje (altså `implicit none`) altid er god at begynde et program med.

Den tredje linje i programmet (lige efter kommentarlinjen) er en tom linje. Dette laves ved at indtaste et linjeskift i editoren. Tomme linjer tjener et programs overskuelighed, hvilket er vigtigt.

På den næste linje ser vi en **erklæring**; den siger, at variablerne a og b skal være heltal. Disse to linjer, med 'implicit none' og 'integer'... fortæller oversætteren noget. Man kalder også sådanne linjer **direktiver**, dvs. ordrer til oversætteren, og de adskiller sig fra **udførbare** linjer ved, at det man siger ikke skal udføres når programmet kører.

Den første af de tre udførbare linjer er en **print**. Den skriver teksten 'A og B?' (se nedenfor, hvorhen), dvs. en opfordring (engelsk: **prompt**) til den person, der bruger programmet, om at gøre dette. I den næste linje ser vi en **read**, hvor tallene bliver læst ind i computeren. Nedenfor forklares det, hvorfra de tal kommer. Senere i bogen bliver symbolet `'*'`, i forbindelse med **print** og **read** forklaret.

Til sidst er der atter en **print**, som printer en lille tekst, og derefter resultatet af at lægge a og b sammen. Læg mærke til forskellen mellem selve teksten, som altid vil se ud som den står der, altså 'a + b =', og den numeriske **værdi** af a+b, som kommer efter. Hvad den er, afhænger jo af, hvad værdierne af a og b var ved indtastning. Her er tale om forskellen mellem en tegnstreng og en numerisk værdi.

I den ovenstående programtekst er nogle ord de såkaldte nøgleord, dvs. ord, som hører til Fortrans ordforråd. Nogle af dem er kommandoer (**print**, **read**) og andre er direktiver (**program**, **integer**, **end**). Alle andre er valgt af programmøren selv. De omfatter bl.a. programmets navn, samt navne på variablerne. Der er ingen restriktion på brug af nøgleord som variabelnavn; for eksempel er det grammatisk korrekt at bruge 'integer' som variabelnavn, men det er ikke god praksis.

1.2 De fysiske rammer

Her kort nogle ord om de fysiske rammer. Selvom de er lidt forskellige, alt efter hvilken computer man arbejder med, er der nogle fælles træk.

Normalt sidder man foran en skærm og læser, hvad der står på den. Det er der, hvor en **print**sætning skriver. Man 'taler' med computeren ved at taste på tastaturet; det er fra dette tastatur, computeren læser input, som man ser i linjen med **read**-sætningen i. Computeren læser dog som regel først, efter man har indtastet

RET(urn). Computeren har også mulighed for at tage informationer (læse) fra filer, eller at skrive information til filer, samt oprette nye filer på stedet. Filer bliver omtalt i kapitel 5.

De fleste systemer tillader desuden, at tastaturet og skærmen kan erstattes af en fil; denne facilitet har forskellige navne, som batch mode, shells eller kommandoprocedurer, m.m. Det er afhængigt af det system man arbejder under.

1.3 Software-rammer

Ud over de fysiske rammer er der også brug for software. Den ovenstående programtekst er bare tekst, normalt skrevet ved brug af en editor. Computeren har sit eget sprog, som er meget svært at programmere i. Det er sekvenser af 0'er og 1'er. Det er derfor oversættere blev opfundet. Teksten bliver oversat af sådan en **oversætter** (eng.: **compiler**) til maskinsprog. Den oversatte tekst, nu i maskinsprog, kaldes et object-modul. Normalt skal der lægges mere til, før det hele kan udføres; dette bliver gjort af linkeren. I mange systemer er de to processer kombineret i en kommando.

1.4 Grundreglerne

Tegnsættet

Som nævnt i afsnit 1.1, må man selv vælge navne på programmet, variabler m.m.; man er også nødt til at have cifre i programteksten, samt nogle andre tegn som punktum, komma osv. Alt skal gøres med Fortrans tegnsæt. Dette indeholder alle (engelske) bogstaver, både store og små (der er ingen forskel, hvad oversættere angår), altså

```
     a b c d e f g h i j k l m n
     o p q r s t u v w x y z
og
     A B C D E F G H I J K L M N
     O P Q R S T U V W X Y Z

cifrene
     0 1 2 3 4 5 6 7 8 9

samt tegnene
     = + - * / ( ) . , ; : ! ' ''
     % & < > ? $ _
```

Bemærk understregen _ (det sidste tegn i rækken) som kan bruges i navne, og opholdet (eller mellemrum, eng. space), ikke vist, hvilket ikke er et synligt tegn. For at gøre det synligt visse steder, bliver det ofte repræsenteret som ␣. Andre tegn er ikke tilladt, for eksempel danske bogstaver. Man kan dog godt bruge disse i kommentarer, hvis systemet ellers kan reproducere dem. Af de ovenstående tegn bruges bogstaver, cifre og '_' i navne (program- og variabel-). Resten af tegnene forklares hen ad vejen.

Symboler eller navne

Et program indeholder et antal **symboler**. Ordet **symbol** omfatter nøgleord som **program**, **end**, osv., samt navne på programmet eller variabler, som brugeren selv bestemmer. Symbolerne for nøgleord er givet af Fortrans syntaks, og der er regler for de symboler, programmøreren selv vælger: de kan være alfanumeriske, dvs. indeholde alfabetiske og ciffertegn, såvel som understregen. De kan have længden fra 1 til 31 tegn, og det allerførste tegn skal være et bogstav; de (eventuelt) efterfølgende må gerne være cifre eller understregen. Der må altså ikke være punktummer eller bindestreg i navne i et Fortran-program. Der må frit blandes store og små bogstaver. Det er lettere at læse og indtaste de små, eller en passende blanding af store og små, og dette anbefales derfor.

Sætninger og linjer

Et program består af en række **sætninger** eller (på engelsk), **statements**. En sætning kan være en erklæring, dvs. en besked til oversætteren, eller en ordre til computeren om at udføre noget. Som regel er der en sætning per **linje**; men det er tilladt at have flere sætninger per linje.

En linje, som altså indeholder en eller flere sætninger, kan begynde hvor som helst, for eksempel i den første position, eller med indrykning. I det simple program i afsnit 1.1 ser man, at den første og sidste linje begynder i position 1, mens alle andre er rykket ind. Det er for at gøre programmets struktur klarere; man kan let se, hvor programmet starter og ender. Indrykning bliver også anvendt til at mærke løkker og andet, se senere i bogen. Linjens længde kan være alt fra 0 til 132 tegn. Er længden 0, så er der tale om en tom linje, som kan bruges til visuelt at afgrænse programdele fra hinanden. Det er nok ikke så ofte man bruger mere end ca. 80 tegn, eftersom de fleste terminalskærme man arbejder med, ikke vil vise flere.

Som sagt, kan der være flere end en enkelt sætning per linje. Det ville for eksempel være ret passende at slå de to første udførbare linjer (print og read) sammen:

```
print *, " A og B?";  read *, a, b
```

Læg mærke til, at hvis der er flere sætninger per linje, adskilles de med ';'. Dette er nødvendigt for

at oversætteren kan finde ud af, hvor en sætning slutter og den næste begynder. Ved at lægge sætninger sammen på den måde kan man ikke bare komprimere et program, men desuden gøre det klarere at læse, når ting, der logisk hører sammen, står sammen.

En sidste mulighed er, at en sætning er så lang, at der ikke er plads på en enkelt linje til den. I dette tilfælde kan man skrive en del af den på den første linje, og fortsætte på den næste, osv. Her skal man så indikere at linjen ikke er afsluttet, ved at sætte tegnet '&' ved enden; den næste linje kan så starte med det samme tegn, men behøver ikke altid. For eksempel kunne vi skrive

```
  read *, &
           a, b
```

Normalt anvendes det kun ved lange sætninger, eller i forbindelse med visse logiske rækker, som kommer senere. Denne facilitet kaldes **continuation lines**. Der er et begrænset antal continuation lines der er tilladt, dvs. 39 – næppe en ulempe i praksis. Når det, der fortsætter på den næste linje, er en (lang) tegnstreng, så skal denne linje begynde med &, og tegnstrengen fortsætter lige efter dette tegn. Det ovenstående eksempel vil i så fald se ud som

```
  read *, &
          &a, b
```

I de fleste tilfælde kan man nøjes med kun det ene & ved enden af linjer, der skal fortsættes fra.

En kommentar er altid den tekst, der står efter et udråbstegn på en linje. Hvis tegnet står i første position, er hele linjen en kommentar. Kommentarer bruges til at forklare programmet eller dele af et program. Vi kunne for eksempel pynte vores simple program således:

```
program SIMPEL_DEMO
! Et simpelt demoprogram.
! Erklaeringerne:
  implicit none  ! Forklares senere
  integer :: a, b

! Opfordring til at indtaste A og B:
  print *, " Indtast tallene A og B:"
! Indlaesning af de indtastede tal:
  read *, a, b
! Udskrift af tallenes sum:
  print *, " a + b =", a+b
! Programmet slutter her:
end program SIMPEL_DEMO
```

Det er klart overdrevent med alle de overflødige kommentarer; det kan virke irriterende med for mange af dem. Man kan gøre dem mindre forstyrrende ved at sætte dem i slutningen af linjerne:

```
program SIMPEL_DEMO
! Et simpelt demoprogram.

  implicit none    ! Forkl. senere
  integer :: a, b  !     -''-

  print *, " A og B?"      ! a og b?
  read *, a, b             ! Indlaesning
  print *, " a + b =", a+b ! Sum udskrives

end program SIMPEL_DEMO    ! Slutter her.
```

Det er stadig for meget. Normalt burde kommentarer holdes nede til det strengt nødvendige, for at forklare noget, der måske ikke umiddelbart er gennemskueligt, ellers i indledningen af et program, hvor man burde forklare programmets formål, samt måske dato, programmørens navn og eventuelt vejledning for at bruge programmet, hvis man har skrevet det til andres brug.

1.5 Typer: variabler og konstanter

Programmering består i grunden af at tage information ind, transformere den til anden information og udskrive resultatet. Sådan information kan være af forskellig art. I Fortran er de fundamentale **typer** information: numerisk, logisk og tegn. Et Fortran-program kan altså operere på alle disse typer information. Informationerne er placeret i enheder, som kan være variabler eller konstanter; de har hver sit navn. Lige som vi i algebra taler om en variabel `x` (hvilket er et navn), eller en konstant som *pi*, kan vi definere navne i Fortran. Hver information har altså to egenskaber: et navn, og en værdi (x kan være 1, 7.77 eller noget andet bestemt). Eftersom der er forskellige typer, er der forskellige slags værdier. En værdi kan være numerisk (et tal), et tegn eller en logisk værdi. For eksempel kunne vi erklære en tegnvariabel med navn `TEGN`. Spørgsmålet "Hvad er `TEGN`s værdi"? kan måske besvares med "Den er 'A'". Pointen er, at begrebet **værdi** her bliver udvidet fra kun numerisk til andre slags værdier.

Endelig er der forskellige slags numeriske værdier, og derved typer. De falder i to klasser: **integer** (heltal) og **real** (tal med decimaler efter punktummet). Inden for disse to klasser er der igen forskellige arter; en af dem har sit eget navn (typen **double precision**). Alt dette, og hvordan man definerer dem i et program, skal nu beskrives mere detaljeret.

Numeriske typer

Her skal vi nøjes med de basale typebetegnelser. Senere i bogen går vi mere i enkeltheder om disse typers egenskaber og hvordan man vælger blandt dem.

Et godt ord om *typing* (typegivning): I gamle bøger om Fortran blev det altid fremhævet, at Fortran har hvad der (på engelsk) hedder implicit typing. Dette er stadig tilfældet, men det anses nu som forkasteligt at benytte sig af det. Implicit typing betyder, at variabler, hvis navn begynder med a–h og o–z, per automatik er af typen **real**, hvorimod de, hvis navn begynder med i–n, er heltal. Dette kan undertrykkes med sætningen

```
implicit none
```

som gør, at alle variabler skal erklæres. Umiddelbart kan det virke besværligt, men erfaringen viser, at det redder programmøren fra en slags fejl, der ellers hyppigt forekommer. For at give et eksempel: hvis man benytter sig af implicit typing og bruger en heltalsvariabel med navnet `nummer`, og senere i programmet refererer til `numer`, så vil oversætteren stiltiende acceptere `numer` som en ny heltalsvariabel, der dukker op for første gang her, og give den en tilfældig værdi – næppe den, variablen `nummer` har. Bruger man derimod `implicit none`, brokker oversætteren sig over, at denne variabel ikke er erklæret, hvilket afslører fejlen. I alle programeksempler i denne bog begynder derfor hver programenhed med `implicit none`-sætningen.

Heltal (eng.: **integer**s) udgør i Fortran et endeligt sæt hele tal. Sættet af de mulige værdier er bestemt af, hvor mange cifre vi tillader heltallene at have. Dette er diskuteret nærmere i kapitel 14. Nok de fleste regnemaskiner i disse dage er såkaldte 32-bitsmaskiner, hvilket giver et standardområde for heltal fra -2^{31} til $2^{31}-1$. Heltal erklæres som **integer**, for eksempel erklærer sætningen

```
integer :: a, i, kay, m_11
```

variablerne `a`, `i`, `kay` og `m_11` som værende heltal med standardområdet. Bemærk det dobbelte kolon; det er ikke nødvendigt i dette enkelte tilfælde, men er altid anbefalet.

REAL tal: Dette er klassen af værdier der har decimaler efter punktummet (det engelske ækvivalent af kommaet; hvor man på dansk for eksempel skriver $0,5$, skriver man jo på engelsk og i Fortran 0.5). Real tal kan ligge ved heltalsværdier, men de er alligevel ikke heltal, fordi den måde de er gemt væk i computeren giver dem en helt anden struktur end heltal. Real variabler er erklæret, logisk nok, med en sætning som

```
real :: x, z, sum1, sum2, &
        summen_af_A_og_B
```

osv. Ligesom med heltal er der et begrænset sæt af disse reals; men det er lidt mere indviklet, hvordan det sæt er opbygget (se kapitel 14). Man kan sige at ved de fleste computere i dag, med deres 32-bit ordstørrelse, har typen **real** en præcision på ca. 6–7 decimaler i alt, og et område fra $\pm 10^{-38}$ til $\pm 10^{38}$. Normalt er det antallet af decimaler, der er mest interessant.

Er man ikke tilfreds med de (normalt) 6–7 decimaler, er der typen **double precision**, som også er real, i den forstand, at den definerer en værdi med et antal decimaler efter punktummet, men flere end med standard real. Erklæringen er, for eksempel

```
double precision :: summ, x12
```

og som oftest får man her ca. 14–15 decimaler, og et område fra $\pm 10^{-308}$ til $\pm 10^{308}$. Denne type er fossilsk i dag (se kap. 12 og 13).

I senere kapitler udvider vi de to slags reals til noget mere generelt, hvor programmøren selv bestemmer antallet af decimaler og området.

Komplekstal: De gives af typen **complex**, og de er tilgodeset i mange af de indbyggede funktioner. En `complex` variabel er en sammenpakning af to værdier af typen **real**, hvor den første repræsenterer den reelle del og den anden den imaginære del af kompleksværdien. I matematikken skriver man for eksempel $z = x + iy$, og i et Fortran-program ville man så erklære z ved

```
complex :: z
```

hvilket reserverer plads til de to reals der udgør `z`.

Konstanter: I de fleste programmer er der konstanter. For eksempel i programfragmentet

```
real :: Rente,Saldo,Rentesats,Areal,r
...
Rente = Saldo * Rentesats / 100
Areal = 3.1416 * r * r
```

ser man værdien 100 repræsenteret af tegnfølgen `100` og *pi* som `3.1416`. Disse tegn udgør faste værdier; man kan ikke ændre dem, som man kan en variabel. Konstanter har også typer, dvs. de kan være integer eller real (og andet), alt efter hvordan man skriver dem. Så er ovenstående 100 et heltal, mens 3.1416 er et real tal. Der er andre måder at skrive konstanter på. Bliver de for eksempel store, kan det være nemmere at skrive dem som potenser af 10; således kunne vi skrive 100 som `10E2`, som betyder 10^2, og 3.1416 som `31.416E-01` eller `0.31416E01`, som henholdsvis betyder $31.416 * 10^{-1}$ og $0.31416 * 10^{1}$.

Det er muligt at navngive konstanter. Et godt eksempel kunne være netop konstanten *pi*. Det kan være, at den bliver brugt flere steder i programmet, og man bliver træt af at indtaste 3.1416 hver gang. Desuden er det tænkeligt, at man begår en fejl ved en eller flere af dem, og indtaster forkert. Man kan, en gang for alle, give værdien et navn og sætte værdien samtidig, ved at benytte **parameter**-faciliteten. Det kan se sådan ud:

```
implicit none
real, parameter :: pi = 3.141593
...
Areal = pi * r * r
```

I anden linje ser vi to attributter defineret for symbolet (navnet) `pi`; at det er en real, og at det er en konstant med navn eller en **parameter**; samtidigt får den en værdi, hvor vi nu har givet den seks decimaler. Den værdi skal være i overenstemmelse, i sin type, med den type, der er angivet til venstre, altså her typen **real**. I resten af programmet kan vi så nøjes med symbolet *pi*. Bemærk at dette symbol er noget andet end et symbol for en variabel; vi må ikke prøve at ændre dets værdi med en sætning som

```
pi = 3.14   ! Forkert!
```

Oversætteren vil give en fejlmelding her. Der bliver vist andre eksempler på brugen af `parameter`.

Konstanter af typen **complex** er lidt mere indviklede. For at angive, at der er tale om en kompleks værdi, indrammes et talpar af parenteser; for eksempel repræsenterer `(1.0,2.0)` den komplekse værdi $1.0 + 2.0i$. Man må også skrive konstanten i formen `(1,2)`. Begge heltal vil i så fald blive omdannet til de passende **real** værdier.

Ikke-numeriske typer

Tegn og **tegnfølger**: Tegn er den type der hedder `character` på engelsk. Denne type består af de tegn der blev vist i afsnit 1.4, dvs. alle de (engelske) bogstaver, cifrene 0–9 og de specielle tegn i listen. *Tegnet '7'* er blot et tegn, og har intet at gøre med talværdien 7. Der er alt i alt 128 forskellige tegn eller tegnværdier; de er de 84 tegn, Fortran selv tillader, samt nogle ekstra tegn, man finder på et tastatur, som |, ^, £, ~, #, @, som ikke er en del af Fortrans syntaks. Man kan sige, at sættet er det, man kan få fra at taste på et engelsk tastatur, samt nogle usynlige tegn.

Man kan erklære variabler af typen tegn i et program; de kan bestå af enkelte tegn, eller tegnfølger eller tegnstrenge (engelsk: **character strings**), dvs. sekvenser af enkelte tegn. Når man vil oprette en tegnstreng, skal man også angive, hvor lang den skal være. Her er nogle erklæringer af tegnvariabler:

```
character         :: char, tegn, a
character(LEN=10) :: string, kort*4
```

hvor den første sætning erklærer de tre variabler, `char`, `tegn` og `a` som enkelt-tegn. I den anden sætning siger (LEN=10) i grunden at alt, der erklæres i denne linje burde være en tegnfølge af 10 tegn; det er også tilfældet med variablen `string`. Men vi ser, at den næste variabel, `kort`, er efterfulgt af *4. Dette tvinger oversætteren til at se bort fra længden 10 og oprette en streng af 4 tegn i stedet. På denne måde kan man erklære flere tegnstrengvariabler af forskellige længder på en enkelt linje. I det næste kapitel beskrives, hvordan tegn og tegnstrenge kan tilskrives værdier, og hvordan man kan pille ved udvalgte dele af en tegnstreng.

Lige som et bestemt tal, som for eksempel 7, er en konstant, er der tegnkonstanter. Et eksempel er tegnet '7'; dette tegn er en tegnværdi, og den værdi kan tildeles en tegnvariabel, for eksempel med sætningen

```
char = '7'
```

Der er også følger af konstanter, som `'abcd'`. For at angive at der er tale om tegn, skal disse konstanter altid "indrammes" af enkelte eller dobbelte anførselstegn. Altså, `'abcd'` eller "`abcd`" er det samme. Det giver et problem: hvordan kan man have anførselstegn i tegnstrengen? Reglen er at hvis man indrammer en streng med to enkelte anførselstegn, så er der ingen problemer med et dobbelt et i strengen, men for at have et enkelt i strengen skal man sætte to efter hinanden; de bliver så tolket som et enkelt. Det er tilsvarende for dobbelte i en streng indrammet af dobbelte. Det sikreste og letteste er at indramme med den ene slags hvis strengen skal indeholde den anden. Det engelske ord "don't" kan f.eks. repæsenteres enten som `"don't"` eller `'don''t'`.

Logical: Den sidste af grundtyperne er typen **logical** eller den logiske type. Den består af et sæt af blot to mulige værdier, som udtrykker om noget er *falsk* eller *sandt*. De bliver brugt i forbindelse med beslutninger og forgreninger, omtalt i kapitel 2. Typen **logical** er erklæret, logisk nok, med for eksempel

```
logical :: fejl, alt_i_orden, OK
```

og også her har vi konstanter, dvs. hele to konstanter. De to værdier er skrevet i formen `.false.` og `.true.` og her er det punktummerne, der markerer typen af de to konstanter. Vi kan altså have sætninger som

```
fejl = .true.
OK   = .false.
```

osv.

Kapitel 2
Sætninger

2.1 Generelt

I kapitel 1 blev nogle generelle grundregler fremlagt; i dette kapitel gennemgår vi selve de forskellige slags sætninger (eng.: **statements**). Disse omfatter erklæringer; aritmetiske, tegn- og logiske tilordningssætninger; **if**- og **case-**sætninger, samt løkker. Simpel, "list-directed" input/output er også beskrevet her, samt `STOP`. Disse dækker ikke alle. `CALL`-sætningen kommer i kapitlet om procedurer, og endnu nogle sætninger bliver nævnt i senere kapitler med mere avancerede såvel som frarådede sætninger.

Sætningsrækkefølge

En vigtig regel skal nævnes her: I hver enhed (program, procedure eller module) skal alle erklæringssætninger komme først, efterfulgt af de eksekverbare (se definitionerne nedenfor). Bryder man den regel, giver oversætteren typisk fejlmeldingen "Statement out of order".

2.2 Erklæringer

Lige efter programhovedet (eller, mere generelt, hovedet) følger et antal erklæringer, hvor variabler og eventuelt nogle konstanter (parameters) defineres. Disse sætninger tager effekt, når programteksten bliver oversat, og de kaldes **direktiver** til oversætteren, i modsætning til de såkaldte eksekverbare (eng.: executable)sætninger, som har effekt (bliver udført) når programmet endelig kører. Ordet "direktiv" betyder, at sådanne sætninger giver besked til oversætteren om, for eksempel, at nogle variabler skal have visse egenskaber.

Implicit none

I kap. 1 blev der vist nogle erklæringssætninger. Der blev også vist den allerførste man bør starte ethvert program (og underprogram) med:

```
implicit none
```

Man undgår en hel del fejl ved at tvinge sig selv til at erklære alle variabler eksplicit ved at benytte denne sætning. Fejl er faktisk det, der tager det meste af en programmørs tid.

Simple erklæringer

Her skal der kun nævnes enkle former af erklæringer; de mere avancerede kommer i senere kapitler. Generelt sagt består en erklæringssætning af

typebetegnelse [, *andet*] [::]
symbol [, *symbol* ,...]

hvor *typebetegnelse* kan være `integer`, `real`, `complex`, `character(len=...)` eller `logical`. De firkantede parenteser betyder, at den del er valgfri, eller kan være det. Således er de to koloner :: valgfri i disse simple erklæringer. Da de imidlertid er påkrævet i visse sammenhænge, er det anbefalelsesværdigt at vænne sig til altid at bruge dem. Delen *andet* – som kan mangle – kan være, for eksempel, `parameter`specifikation, som beskrives nedenfor. På den højre side af :: er der et eller flere symboler. Hvis de er variabler, så er det altså dem, der skal have deres type nævnt på venstre side. Nogle eksempler er:

```
integer           :: i,j,n,antal
real              :: x,sum,squares
logical           :: alt_i_orden
complex           :: tal
character(len=10) :: c,str*8,a*1
```

Den sidste erklæring er interessant. `len=10` siger, at der er tale om en tegnstreng med en længde på 10 tegn. Men på den højre side ser vi, at mens variablen `c` skal have dette attribut, skal `str` være kun 8 tegn lang, og `a` kun et enkelt tegn. På denne måde, ved at bruge `*` og længden, kan man erklære flere tegnstrenge, af forskellige længder, på samme linje.

Startværdier (initialisering)

Nogle variabler får en værdi, første gang de bliver nævnt, mens andre bruges, som om de har en værdi i forvejen. Et eksempel er en løbende sum, eller et minimum- eller maksimumstal. Man kan i så fald bruge en sætning der giver dem en værdi, men en mere bekvem måde er at initialisere dem i selve erklæringssætningen. I det ovenstående kan variablen `sum` være sådan en variabel, og den sætning kunne være

```
real  :: x, sum=0.0, squares
```

som sætter `sum` lig med nul fra starten. Nogle regnemaskiner er så "venlige" at nulstille alle variabler i starten, og man kan derfor være fristet til at antage dette. Problemet er, at andre maskiner ikke er så venlige, og ved disse er der helt tilfældige værdier i variabler i starten. Når man så flytter et kørende program fra den ene slags maskiner til den anden, kan det gå galt. Det er altså en god ide altid at initialisere alle de variabler, hvis startværdier har betydning.

Denne måde at initialisere variabler på kan dog ikke bruges i procedurer; mere om det i kap. 4 og 8.

2.3 Tilordningssætninger

En tilordningssætning (eng.: **assignment statement**) er en sætning, i hvilken en variabel får tilordnet (eller tilskrevet) en værdi; generelt:

```
var = udtryk
```

Man skal være klar over, at lighedstegnet = her ikke betyder "er lig med", men snarere betyder "skal få værdien". Det siges her, fordi det ellers kan være forvirrende at se den variabel, der står på venstre side, også stå på højre side som en del af udtrykket. Det er helt i orden, for eksempel, at bruge sætningen

```
i = i + 1
```

som i matematikken jo ville være vrøvl. I et Fortran-program betyder denne sætning at variablen `i` skal blive lig med dens gamle værdi forøget med 1.

Udtryk generelt

Et udtryk er en samling af en eller flere variabler og/eller konstanter, samt muligvis en eller flere operatorer. Der står altid et udtryk på den højre side af en tilordningssætning. Der er forskellige slags udtryk og tilhørende operatorer og konstanter, alt efter hvilken type værdi det bliver. I det følgende gennemgås de basale typer udtryk.

Aritmetiske udtryk

Et eksempel på et aritmetisk udtryk ses i den ovenstående sætning; udtrykket `i+1` er en aritmetisk operation. Her er der en variabel (`i`), en konstant (`1`) samt en operator (`+`) der forbinder dem. Der er de følgende operatorer:

`+`	addition
`-`	subtraktion
`*`	multiplikation
`/`	division
`**`	eksponentiering

Desuden kan der være parenteser. De kommer ind i billedet på grund af operatorprioritet. Hvis man har et udtryk som `a + b *c`, så kunne der i princippet være tvivl om, om $a + bc$ er ment, eller $(a + b)c$. Derfor måtte man indføre operatorprioriteter. Således har operationerne `+` og `-` samme, laveste prioritet; `*` og `/` samme og højere prioritet, og eksponentiering `**` den højeste. I udtrykket `a + b * c` bliver multiplikationen altså udført først (højeste prioritet), efterfulgt af additionen. Hvis vi lægger `**` til vores eksempel: `a + b * c**2`, så betyder det, logisk nok, $a + bc^2$, og ikke, for eksempel, $a + (bc)^2$ osv. Vil man tvinge en anden prioritet, kan man gøre det med parenteser. Hvis vi altså mener $a + (bc)^2$, skriver vi i programmet

```
y = a + (b*c)**2
```

En bemærkning om eksponentiering. Hvis potensen er et heltal, er der sjældent problemer, og oversætteren udfører muligvis operationen ved multiplikation; dvs., `x**2` bliver muligvis udført som `x*x`. Hvis potensen derimod er et **real** tal, bliver operationen udført ved at gange logaritmen med potensen, efterfulgt af antilogaritmen. Dette betyder i praksis, at hvis tallet, der skal opløftes, er negativt, kan det ikke lade sig gøre, fordi man ikke kan have logaritmen af tallet – selvom man ved, at operationen principielt er i orden. For eksempel er kubikroden af et negativt tal defineret. Men hvis x er negativ, vil en sætning som

```
y = x**(1.0/3.0)
```

føre til en fejlbesked. Her skal man være lidt opfindsom for at tvinge programmet til at udføre operationen alligevel.

Med hensyn til division er der en ting man skal passe på. Når et heltal divideres med et andet heltal, er resultatet også et heltal. Hvis resultatet egentligt ligger imellem to heltal, så bliver det rundet ned til det lavere af de to (mere præcist: til det nærmeste heltal, rettet mod nul; det inkluderer derved de negative resultater). For eksempel er værdien af udtrykket `3/4` lig med nul og `-8/3` lig med `-2`. Det er grunden til, at potensen `1/3` i det ovenstående eksempel, skrives som `1.0/3.0`; det sikrer, at resultatet er af typen `real`, ligesom de to tal. Dette skal der siges mere om nedenfor, under blanding af typer i et udtryk.

En hyppigt forkommende fejl er at skrive

```
z = x / a*b
```

når $x/(ab)$ er ment. Der er to måder at skrive udtrykket rigtigt på : `x / (a*b)` eller `x / a / b`.

Det er tilladt at blande de forskellige numeriske typer, dvs., heltal, reals og complex inden

for et udtryk. Det er sådan, at de forskellige numeriske typer er lagret i maskinen på vidt forskellig vis, og faktisk kan maskinen ikke operere to tal af forskellig type med hinanden, på grund af det. For at gøre det muligt bliver de inkompatible værdier konverteret for at gøre dem kompatible. Der er en rangfølge, fra de simplere typer til de mere komplexe: `integer<real<double precision<complex`. Når man blander to typer i et udtryk, så bliver den mindre komplekse konverteret til samme form som den mere komplekse, før evaluering finder sted. Resultatet er så også af den mere komplekse type. Alt dette sker dog kun parvis, to variabler eller konstanter ad gangen. Her kommer vi ind på rækkefølgen af evaluering af et udtryk. Et længere udtryk som for eksempel `i / j * x * a**2` (`i` og `j` `integer`, `x` og `a` `real`), evalueres trinvis. Standarden foreskriver, at rækkefølgen generelt skal være fra venstre mod højre, dog med hensyn til operatorprioritet og parenteser, samt en undtagelse, se nedenfor. Maskinen ville altså evaluere `i / j` først, og så kunne der opstå den ovenstående fejl. Den kan man definitivt forhindre ved enten at anordne variablerne forskelligt, for eksempel som `i * x / j * a**2`, eller tvinge den ønskede rækkefølge med parenteser: `i / (j * x) * a**2`. Så er det lige meget, i hvilken orden evalueringen foregår. Venstre-mod-højre-reglen bliver vendt om hvis der er to `**` operatorer; udtrykket `2**3**4` er altså ækvivalent med `2**(3**4)`.

Logiske udtryk

Ligesom aritmetiske udtryk, består logiske udtryk af konstanter, variabler og operatorer. De evalueres altid til en logisk værdi, altså `.false.` eller `.true.` (der er kun de to). Det simpleste logiske udtryk er altså en af de to konstanter, eller en enkelt variabel. For det meste har man dog den form der sammenligner andre typer, og her bruger man de relativerende operatorer (eng.: **relative operators**). Et eksempel er et udtryk som `x < 0.0`, som så at sige stiller spørgsmålet, om x er mindre end 0.0; det kan enten være falsk eller sandt. Af denne slags operatorer er der følgende (med alternativer i parenteser):

`>`	(`.gt.`)	"er større end"
`<`	(`.lt.`)	"er mindre end"
`>=`	(`.ge.`)	"er større end eller lig med"
`<=`	(`.le.`)	"er mindre end eller lig med"
`==`	(`.eq.`)	"er lig med"
`/=`	(`.ne.`)	"er ikke lig med"
`.eqv.`		"har samme logiske værdi"
`.neqv.`		"har ikke samme logiske værdi"

De første seks har en gammel form (vist i parenteser) som nu er forældet, men stadig må bruges. De sidste to ligner i deres funktion henholdsvis `==` og `/=`, men de kan kun bruges til at sammenligne logiske udtryk. Man kan fx bruge udtrykket `i == j` (hvor `i` og `j` er numeriske variabler), men hvis man vil sammenligne logiske værdier, skal man bruge de to, for eksempel

```
l1 .eqv. (x<0.0)
```

hvor `l1` er en logisk variabel, som sammenlignes med den logiske værdi af sammenligning af `x` og `0.0`. Det bemærkes at det i dette udtryk ikke var nødvendigt med de to parenteser, men de blev brugt for at gøre tydeligt hvad der menes.

Fra den booleske algebra, der befatter sig med logiske størrelser, ved vi at der skal flere operatorer til, og også de findes i Fortran. Det er den gruppe, der opererer på to logiske værdier (binære operatorer), og den (de unitære) der opererer på en enkelt ad gangen. De er:

`.not.`	"ikke (eller modsat)"
`.and.`	"og"
`.or.`	"eller"

og de har faldende priotitet i denne rækkefølge. Operatorerne `.and.` og `.or.` bruges til at kæde logiske udtryk sammen. Skriver vi for eksempel `i/=j .and. y>0.0` så mener vi at begge logiske udtryk, `i/=j` og `y>0.0` skal holde for at hele udtrykket er sandt; i modsat fald, falsk. Skriver vi derimod `i/=j .or. y>0.0` så er værdien sand hvis enten det ene eller det andet af de to udtryk er sandt. På den måde kan man kæde betingelser sammen.

Operatoren `.not.` inverterer den logiske værdi; den er en unitær operator, i slægt med minustegnet. Har vi for eksempel en logisk variabel `OK` som indikerer om et eller andet er i orden, så indikerer udtrykket `.not.OK` det modsatte.

Indtil nu blev der kun vist numeriske variabler eller konstanter i logiske udtryk, men andre kan også bruges (sammenlignes). Det er ikke så sjældent, for eksempel, at sammenligne tegn med tegnkonstanter, som i

```
char == "#"
str(1:3) == "abc"
```

Det er også i orden at bruge operatorerne <, > osv. med tegn. Således giver udtrykket

```
tegn>='a' .and. tegn<='z'
```

et udsagn om, om variablen `tegn` er i det lille alfabet.

Eksempler på disse udtryks brug i `if`-sætninger ses senere i dette kapitel.

Character-udtryk

I sammenhæng med tegn og tegnfølger, er der kun én operator i udtryk, sammenkædningsoperatoren `//`. Som navnet antyder, kæder den tegn[følger] sammen. Således fører udtrykket `"ABCD" //"EF"` til den nye tegnfølge `"ABCDEF"`. Her skal man blot være klar over, at der ingen mellemrum er sat ind. Udtrykket `'Kurt' // 'Jensen'` bliver altså til strengen 'KurtJensen'.

Eksempler på tilordningssætninger

Alle udtryksformer er nu samlet, og vi kan give nogle eksempler på sætninger, der bruger dem. Her er nogle aritmetiske tilordningssætninger:

```
i = 1;   j = 3
k = i + j
i = i + 1
k = i / j

x = i
y = a + b*x + c*x**2
k = y
z = x / (a*b)
z = x / a / b
```

hvor vi antager at `i, j, k` er heltal og `x, y, z, a, b, c` er af typen `real`.

Med den fjerde sætning er det værd at sige, at resultatet er 0! Den næste sætning lægger heltalsværdien af `i` i `x`; her skal heltallet først konverteres til formen `real`. I den efterfølgende sætning får `y` en værdi, givetvis imellem to heltal. Derefter bliver denne værdi sat ind i heltalsvariablen `k`, og så må den først nedrundes til det lavere af de to heltal. Som en øvelse, prøv de følgende linjer:

```
x = 2
y = SQRT(x);  y = y**2
i = y
```

Prøv at printe de sidste to værdier, `y` og `i`. Det kan overraske, at `y` formentligt printes ud som `2.000000`, mens `i` bliver til `1`. Dette skyldes, at `print`-sætningen afrunder værdier, men i virkeligheden er `y` jo en anelse mindre end 2, og når dens værdi er sat ind i `i`, bliver den nedrundet til `1`.

De sidste to sætninger (`z =...`) siger det samme.

Her er nogle logiske sætninger (`OK, cond, peak` er af typen `logical`, og de andre er som ovenstående, samt de nye `reals x1, x2` og `x3`):

```
OK = i /= k
cond = i>j .and. x>0.0 .or. y>x
peak = x1<x2 .and. x2>=x3
```

Hvis man er i tvivl om, hvorvidt parenteser er nødvendige i den midterste sætning, så må man godt bruge dem til at gøre det tydeligt, hvad man mener. De to forskellige former,

```
cond =  i>j .and. x>0.0  .or. y>x
cond = (i>j .and.  x>0.0) .or. y>x
```

er identiske, fordi operatoren `.and.` har en højre prioritet end `.or.`, så parenteserne ikke er nødvendige.

Her til sidst følger nogle sætninger med tegn og tegnfølger. Antag, at alle variabler er af typen `character` (og af en tilstrækkelig længde):

```
a = "flaske"
b = "post"
c = a // b           ! "flaskepost"
d = a(3:6) // b(1:2) &
          // "t"    ! "askepot"
```

2.4 IF, CASE

Ud over det, at en regnemaskine kan udføre mange operationer på kort tid, har den også den stærke facilitet, at den kan forgrene eksekution til forskellige grene, afhængigt af visse betingelser. I Fortran er der to måder at forgrene på: med hjælp af **if**-sætninger, og ved brug af **case**. Det sidste kan anses som en udvikling af det første, og vi tager dem i denne rækkefølge. Desuden findes der den forældede `goto`-form, men den kan ikke anbefales.

IF-sætninger

Den mest simple **if**-sætning er

```
if (betingelse) sætning
```

hvori *betingelse* er et logisk udtryk (se ovenfor), og *sætning* kan være en hvilken som helst sætning, men kun en enkelt. Læg mærke til parenteserne, hvori det logiske udtryk skal indrammes. Et eksempel kan være,

```
if (x > 0.0) y = SQRT(x)
```

Nogle puristermener, at denne form for **if**-sætning burde undgås, men der er situationer, hvor den er på sin plads, og den fylder mindre end den mere generelle form, der siges at være bedre. Hvis det, der skal udføres hvis betingelsen holder (udtrykket har værdien `.true.`), er mere end en enkelt sætning, er man nødt til at bruge følgende generelle form:

```
if (betingelse) then
  sætning 1
  sætning 2
  ...
endif
```

Et eksempel på dette er:

```
if (x < 0.0) then
   y = 0
   print *, " x<0: y sat til nul"
endif
```

Bemærk, at så snart man bruger `then`, skal der skiftes til en ny sætning. Det kan dog lade sig gøre på en linje ved at bruge flere sætninger per linje, for eksempel

```
if (x<0.0) then; y=0
print *, " x<0: y sat til nul"; endif
```

(bemærk semikolonerne). Det er ikke altid en god ide.

I de ovenstående eksempler er det sådan, at hvis en betingelse holder, skal der udføres en eller flere sætninger. Programmet fortsætter derefter med nye sætninger, uanset betingelsen. Men det er ofte nødvendigt at udføre nogle sætninger kun i modsat tilfælde, altså hvis betingelsen ikke holder. Til dette formål udvides **if**-formen til:

```
if (betingelse) then
   sætning 1
   sætning 2
   ...
else
   alternativ sætning 1
   alternativ sætning 2
   ...
endif
```

Dette kunne for eksempel være

```
if (antal == 0) then
   print *, "Ingen fundet."
else
   print *, antal, " fundet."
endif
```

som forhindrer den lidt grimme udskrift

```
    0 fundet.
```

Det er muligt at have **if**s inden for **if**s. Vi udvider det ovenstående eksempel til:

```
if (antal == 0) then
   print *, "Ingen fundet."
else
   if (antal < 1000) then
      print *, antal, " fundet."
   else
      print *, " > 1000 fundet."
   endif
endif
```

Der henvises til indrykningen, som gør det tydeligt, hvad der foregår. Denne er ikke påkrævet; man må godt skrive det hele som:

```
if (antal == 0) then
print *, "Ingen fundet."
else
if (antal < 1000) then
print *, antal, " fundet."
else
print *, " > 1000 fundet."
endif
endif
```

men det er langt mindre overskueligt. Overskueligheden er en vigtig ting, især når man skal opspore fejl i programmet. Så hjælper det enormt, at programmets struktur er klar. Indrykning anbefales altså kraftigt.

Når man har flere niveauer af `if`-konstruktioner inde i hinanden, kan det – ud over indrykningen – være en idé at markere, hvilken `if` en given `endif` hører til. Her kan man bruge **labels**. En label er et symbol efterfulgt af kolon. Det ovenstående eksempel kunne se sådan ud:

```
ydre:  if (antal == 0) then
         print *, "Ingen fundet."
       else
indre:   if (antal < 1000) then
           print *, antal, " fundet."
         else
           print *, " > 1000 fundet."
         endif indre
       endif ydre
```

hvor symbolerne `ydre` og `indre` så at sige indrammer deres `if`-niveauer. Det kan især være en god ide, når de enkelte konstruktioner bliver længere og måske strækker sig over mere end en side af en udskrift (eller mere end en skærmfuld) og man derved mister overblikket.

Det kan ske, at der er flere end to forgreninger, altså flere muligheder, med hver sin(e) tilhørende sætning(er). I så fald kan man benytte sig af den mest udvidede form:

```
if (betingelse 1) then
   sætning 1
   sætning 2
   ...
else if (betingelse 2) then
   alternativ sætning 1
   alternativ sætning 2
   ...
else if (betingelse 3) then
   alternativ sætning x
   alternativ sætning x+1
   ...
else
   ...
endif
```

osv. Dette bør ikke drives for langt, fordi man – i visse tilfælde – har en bedre form til det, **case**-formen, se nedenfor. I simple tilfælde kan den dog bruges, for eksempel

```
if (x < 0.0) then
  print *, " Ingen kvadratrod mulig"
  z = SIN(x) / x
else if (ABS(x) < TINY(x)) then
  y = 0
  z = 1
else
  y = SQRT(x)
  z = SIN(x) / x
endif
```

I dette lidt kunstige eksempel forhindrer vi henholdsvis et negativt argument til kvadratroden, og division med nul i $sin(x)/x$-funktionen.

CASE-konstruktionen

Når det bliver lidt for meget med `if...else if` osv., går man over til `CASE`. Den kan dog kun anvendes for typerne `integer`, `logical` og `character`. Konstruktionen er formelt

```
select case (var)
   case (værdi 1)
    sætning 1.1
    sætning 1.2
    ...
   case (værdi 2)
    sætning 2.1
    sætning 2.2
    ...
   case (værdi 3)
    sætning 3.1
    sætning 3.2
    ...
 case default
    defaultsætning
 end select
```

Den sidste del (`case default`) kan udelades, men er ofte nyttig; den foreskriver, hvad der skal ske, hvis ingen af de nævnte tilfælde indtræffer. Her ser programmet på de forskellige værdier af variablen *var* og udfører de passende sætninger. Denne konstruktion er nyttig, når der er mange diskrete mulige værdier for en variabel. Et eksempel er:

```
select case (tal)
 case (-1)
  print *, " Tallet er -1"
 case (0)
  print *, " Tallet er nul"
 case (1)
  print *, " Tallet er +1"
 case default
  print*," tallet uden for [-1..+1]"
end select
```

Dette (lidt søgte) eksempel kan også illustrere yderlig en facilitet ved `CASE`; nemlig den, at i stedet for en enkelt værdi kan man angive et område af værdier. Så kunne man udvide eksemplet til

```
select case (tal)
  case (:-1)
    print *, " Tallet er negativt"
  case (0)
    print *, " Tallet er nul"
  case (1:)
    print *, " Tallet er > 0"
end select
```

hvor udtrykket `:-1` angiver alle værdier fra den laveste (negative) værdi op til -1, og `1:` betyder alle positive værdier fra +1 opad. Man kan selvfølgelig vælge bestemte områder, som for eksempel `-10:-1`, `1:1000` osv.

Som sagt ovenfor, gælder denne form kun for de tre nævnte typer. Det er næppe sandsynligt, at man vil bruge den med logiske værdier, eftersom der jo kun er to muligheder, og en simpel `if`-form vil gøre det. Men i forbindelse med typen `character` giver den gode muligheder. I et konkret eksempel har forfatteren tre muligheder for at løse en differentialligning med enten Euler-, 2.-orden Runge-Kutta eller det der hedder BI (for backward implicit). Disse er angivet via en `character`type med navnet `method`, og forløbet er styret med følgende konstruktion:

```
select case (method)
  case ("EU")
    call EULER (...)
  case ("RK")
    call RUNGE_KUTTA (...)
  case ("BI")
    call BI (...)
  case default
    print *, " Ugyldig metode!"
end select
```

Ligesom med `if`-konstruktioner, kan man benytte labels sammen med `case`; men det forekommer lidt overflødigt, fordi man sjældent lægger `case`-konstruktioner ind i hinanden, og overblikket er for det meste klart.

2.5 Løkker (DO-sætning)

I mange programmer skal noget (en eller flere sætninger) udføres gentagne gange; enten et vist

antal gange, eller indtil (eller mens) en vis betingelse er opfyldt. Der er altså tale om tre forskellige slags løkker: en **tælleløkke**, en **do-while** og en **repeat-until** løkke. Alle tre slags løkker realiseres med **DO**-konstruktionen.

Cycle og exit

De to sætninger er knyttet til løkker. `Cycle` betyder at en ny runde skal begynde der, hvor sætningen står; dvs. programmet skal genstarte fra løkkens begyndelse. Derimod bevirker `exit` at der hoppes helt ud af løkken. Der bliver vist eksempler på begge disse sætninger i det følgende.

Tælleløkke

Den generelle form af tælleløkken er følgende:

```
do tæller = startværdi , slutværdi &
                        [,skridtlængde]
  sætning 1
  sætning 2
  sætning 3
  ...
enddo
```

Her er *tæller* en variabel og bør være af typen `integer`. Denne tæller skal løbe fra *startværdi*en til (og med) *slutværdi*en. Som de firkantede parenteser indikerer, kan man desuden specificere skridtlængden hvis man vil; undlades den, bliver `1` brugt. For eksempel ville følgende start af en løkke,

```
do i = 1, 10
```

løbe alle `i`-værdier igennem, 1, 2, ... 10, mens

```
do i = 1, 10, 3
```

ville løbe igennem i = 1, 4, 7 og 10. Reglen er, at når tælleren overgår slutværdien, udføres løkken ikke igen. Det vil sige, at

```
do i = 2, 10, 3
```

løber igennem `i` = 2, 5 og 8, men ikke 11, som jo er større end den øvre grænse.

Det er ikke tvunget at starte tælleren med `1`, og heller ikke at tællingen skal foregå fremad. Følgende `do`-starter er lovlige:

```
do i = -10, 10
```

og

```
do k = 100, 1, -1
```

hvor den første ingen forklaring behøver, og den anden får `i` til at løbe igennem 100, 99, ..., 2, 1.

Selve tælleren er tilgængelig og må godt bruges i udtryk inden for løkken; bare der ikke røres ved den, dvs. den ikke ændres. Et eksempel på anvendelse af tælleren er, når man vil have en anden, `real` variabel til at løbe en række værdier igennem. Lad os sige, vi vil have x gående fra 0.1 til 1.0 i skridt på 0.1. Dette kan lade sig gøre med

```
do i = 1, 10
  x = i * 0.1
  ...
enddo
```

Faktisk tillader Fortran-standarden (og derfor oversættere) brugen af en `real` tæller:

```
do x = 0.1, 1.0, 0.1
  ...
enddo
```

men det er en meget dårlig ide (se kap. 14).

Ligesom med `if`-sætninger, kan man have løkker inde i løkker. I matrixregning skal man ofte gennemgå en matrix helt, og det programmeres ofte med en ydre løkke, der behandler alle rækker, og en indre løkke, der, inden for en given række, behandler alle rækkeelementer:

```
do i = 1, n
  do j = 1, m
    ...
  enddo
enddo
```

Den indre løkke afsluttes med det første af de to `enddo`, efterfulgt af den anden med det andet `enddo`. Igen viser indrykningen klart, hvad der menes.

Hvis man har flere niveauer af løkker inde i hinanden, kan det – ligesom med `if` – være en god ide at mærke de forskellige niveauer med labels. Man mærker hver start på en `do` med sin label, og tilsvarende hver tilhørende `enddo` med samme label. Det ovenstående eksempel bliver så for eksempel

```
L1:  do i = 1, n
  L2:  do j = 1, m
         ...
       enddo L2
     enddo L1
```

Det er tilladt at blande løkker med `if`-sætninger. Generelt kan man sige, at der er sætninger inde i begge konstruktioner, og disse kan være alle lovlige sætninger, inklusive andre `if`- eller `do`-sætninger. Reglen er bare, at de afsluttes de rigtige steder. Altså en kombination som

```
do i = 1, 100
  if (i == 50) then
    ...
  endif
enddo
```

er i orden, men

```
do i = 1, 100
  if (i == 50) then
     ...
enddo
  endif
```

er ikke tilladt. Sådan et sygeligt tilfælde bliver opdaget af oversætteren.

Fortran-standarden kræver, at ved en tælleløkke, er tælleren defineret, efter programmet er gået ud af løkken. Det ville sige, at i det ovenstående (korrekte) eksempel er tælleren `i` defineret og må bruges. Her er det dog for en gangs skyld værd at vide lidt om, hvad der foregår internt; efter forfatterens mening tjener det som en advarsel mod at benytte sig af denne facilitet. Lade os tage et nyt eksempel for at illustrere de ting, der kan ske:

```
do i = 1, n
  ...
  if (...) exit
  ...
enddo
```

Allerførst, hvis `n` = 0, bliver løkken aldrig gennemført, men `i` ville i så fald være sat til 1. Det er nemlig sådan, at det allerførste, der sker, når programmet går ind i løkken er, at tælleren bliver forøget med 1. Derefter bliver dens nye værdi sammenlignet med den øvre grænse, her altså `n`. Hvis tælleren ikke overstiger `n`, bliver løkken udført. Skulle `exit`-sætningen blive udført, så virker dette også efter hensigt; for eksempel, hvis dette skete den første gang, ville man have `i` lig med 1, som forventet. Men, hvis løkken bliver udført alle `n` gange (`n` > 0), så vil der ske det, at efter at løkken er udført den n'te gang, går den ind igen, øger `i` med 1 til `n`+1, hvorefter programmet hopper ud af løkken. Er man klar over alt dette, kan man risikere at bruge værdien af tælleren. Men, er man usikker, er det en bedre ide selv at hold øje med hvor mange gange løkken er kørt, med sin egen tæller:

```
taeller = 0
do i = 1, n
  taeller = taeller + 1
  ...
  if (...) exit
enddo
```

Dens værdi vil så afspejle præcis hvad der foregik, efter løkken er afsluttet.

Der er to konstruktioner, hvor man hopper **ud** af en løkke, og de er nødvendige. Formelt kaldes de *do-while* og *repeat-until*, og kan noteres i de formelle former (som *ikke* er Fortran!)

```
while betingelse &
  do sætning[er] ;
```

og

```
repeat sætning[er] &
  until betingelse ;
```

De ligner hinanden i, at i begge tilfælde gentages en eller flere sætninger (med en løkke) indtil en betingelse er opfyldt. Forskellen er, at ved en *do-while*, bliver betingelsen prøvet først for at se, om løkken skal fortsætte, mens alle sætningerne bliver udført og betingelsen prøvet sidst ved en *repeat-until*. I begge tilfælde skal der hoppes ud af en ellers uendelig løkke. Her er en (formel) realisation af en *do-while* i Fortran:

```
do
  if (betingelse) then
    sætning 1
    sætning 2
    ...
  else
    exit
  endif
enddo
```

Her er det sætningen `exit`, der udløser hoppet ud af løkken. Bemærk at der ingen tæller er efter `do`; det er oplæg til en uendelig løkke, og det ville blive det, hvis ikke det var for `exit`-muligheden. Den ovenstående form kan forenkles til noget mere elegant ved at vende om på betingelsen:

```
do
  if (.not. betingelse) exit
  sætning 1
  sætning 2
  ...
enddo
```

Den formelle form af en *repeat-until* i Fortran er

```
do
  sætning 1
  sætning 2
  ...
  if (betingelse) exit
enddo
```

Her kan man give et realistisk eksempel; at finde en funktions rod ved brug af Newtons metode. Lad funktionen være $y(x)$, og dens afled y' mht x `dydx`, og vi starter med et første gæt på x, `x_gammel`, som hver gang i løkken bliver forbedret til `x_ny` med den velkendte Newtonformel gengivet i alle bøger om numerisk regning, $x_{n+1} = x_n - y(x)/y'(x)$ hvilket kan implementeres i Fortran som en *repeat-until*:

```
x_gammel = ...
do
  y =...
  dydx = ...
  x_ny = x_gammel - y/dydx
  if(ABS(x_ny-x_gammel)<small) exit
  x_gammel = x_ny
enddo
```

Der er udeladt de ting, der er specifikke for det aktuelle problem. Den ovenstående løkke gentages indtil ændringen i x – fra `x_gammel` til `x_ny` – ikke længer er større end `small`, som man har valgt passende lille.

I nok de fleste anvendelser er det lige meget, om man bruger en *do-while* eller en *repeat-until*, men der er situationer, hvor den ene eller anden konstruktion er bedre eller nødvendig. En sædvanlig anvendelse af *do-while* er det, der kaldes en *read-while* i denne bog, og det bliver beskrevet senere i dette kapitel.

Sætningen `cycle` bevirker, at resten af løkken skal springes over, og løkken skal gentages. Et godt eksempel på brugen af dette kan være en datafil, hvor der er linjer, der ikke skal læses, for eksempel de, der har tegnet `#` i første position. Her ville man læse hver linje som en tekst, og kun hvis linjen består af tal, piller man så disse tal ud af teksten. Det kan se sådan ud:

```
character(len=80) :: line
integer           :: ios
...
do
  read (*,'(a80)',iostat=ios) line
  if (line(1:1) == '#') cycle
  ...
enddo
```

Vi har taget forskud på `iostat`-faciliteten i en `read`-sætning, som først bliver (kort) forklaret nedenfor, samt en formatbeskrivelse (`'(a80)'`), som også kommer senere. I det hele taget er det sjældent nødvendigt at bruge `cycle`. Det kan med lethed erstattes af en `if`-sætning, for eksempel:

```
character(len=80) :: line
integer           :: ios
...
do
  read (*,'(a80)',iostat=ios) line
  if (line(1:1) /= '#') then
  ...
  endif
enddo
```

2.6 Simpel I/O (print, read)

Her gennemgås de simple former af `read` og `print`, uden formatspecifikation eller `write` og filer. Betegnelsen I/O står for *input/output*.

Read

Sætningen `read` i sin mest simple form læser input fra tastaturet (eller fra en såkaldt shellfil eller kommandoprocedure, afhængigt af, hvilket operativsystem man bruger). I det simple demoprogram i kapitel 1 ses sådan en `read`-sætning, som indlæser to tal, `a` og `b`:

```
read *, a, b
```

Tegnet `*` betyder, at vi ikke specificerer et bestemt format for det, der indtastes; computeren skal selv finde ud af formen. Der er nogle enkelte regler:

- Hver `read` starter på en ny linje (hvis der altså er linjer); ved et tastatur betyder det begyndelsen af en ny linje; det vil sige det, der kommer efter en vognretur;
- input-items skal separeres enten med mellemrum eller kommaer;
- input-tallene behøver ikke have den rigtige type;
- input-items kan være på flere linjer; `read` vil blive ved med at lede efter dem, indtil alle er fundne.

At hver `read` starter ved en ny linje har betydning, hvis man indtaster flere data på en linje, end programmet er bedt om at indlæse. Kommer der så en ny `read`, bliver de overskydende data set bort fra. Det med typerne vil sige, at for eksempel datatal der er erklæret som typen `real`, gerne må indtastes som heltal.

Tegnstrenge

Med denne type inputdata er den simple form for `read` ikke så bekvem at bruge. Eftersom de jo kunne indeholde mellemrum og kommaer, kan de ikke afgrænses med disse, og man er derfor tvunget til at mærke en streng ved at sætte anførselstegn omkring den. Glemmer man det, går det galt ved indlæsningen.

Brug af iostat – *read-while*

Det er ofte sådan, at et program skulle kunne finde ud af, at der ikke længere er data at indlæse, og skulle holde op. Det er styret af det såkaldte *end-of-file*-mærke, eller `eof`. Hvis for eksempel et program opfordrer til datainput fra tastaturet, skal

man kunne indtaste noget, der svarer til dette `eof`; det gængse her er `^d` ("control-d"). Hvis data input kommer fra en shellfil, er der forskellige måder at indikere `eof`; men der er altid en måde at vise det på. I selve programmet er det så *iostat* inden for `read`-sætningen, der detekterer det. Lade os antage, at der, i et givet program, kommer et antal af inputtal `x`, og at programmet skal blive ved med at læse dem ind, indtil der er *eof*. Det er klart en *do-while* struktur, som her skal kaldes *read-while*. I algoritmisk sprog, ser det sådan ud (underforstået at `ios` er heltal):

```
while more input do
  ...
endwhile
```

og oversat til Fortran,

```
do
  read (*,*,iostat=ios) x
  if (ios < 0) exit
  ...
enddo
```

Den første af de to stjerner i `read` siger, at inputtet skal komme fra tastaturet (eller hvad standard-input er, måske en shell- eller batchfil); den anden siger at der ikke er et format specificeret. Ordet `iostat` er et nøgleord, mens `ios` er valgt af programmøren og er navnet på den tilsvarende variabel. Den får altså værdien af I/O'ens status, og Fortran specificerer, at værdien er mindre end nul hvis der er *eof*. Når der **er** inputdata, er værdien af `ios` lig med 0. Navnet af variablen `ios` er vilkårligt valgt; den kunne hedde hvad som helst. Her er et lille eksempel på brugen af en *read-while*. Der skal indlæses et antal N af heltal i et array, men uden at vide i forvejen, hvor mange der kommer fra inputtet.

```
program READ_WHILE
  implicit none
  integer :: fifi, N, arr(1000)

  N = 0
  print *, " Indtast tal, afslut med CTRL-d:"
  do
    read(*,*,iostat=fifi) arr(N+1)
    if (fifi /= 0) exit ! Exit ved eof
    N = N + 1
  enddo
  print '(" Der blev", i6, &
          & " tal laest ind.")', N
end program READ_WHILE
```

Her brugtes der navnet `fifi` for at illustrere frihedenved valget. Eksemplet læser kun et antal tal i et array, men gør ikke noget mere ved det. Der bruges `print`-sætningen, som bliver beskrevet i næste afsnit, og et array, som bliver beskrevet i næste kapitel.

Print

Som modstykke til `read`, har vi `print`-sætningen, som skriver noget, normalt til skærmen. Formen er:

```
print *, list
```

hvor *list* er en liste over de ting, der skal skrives ud. Typerne i denne liste bestemmer, hvordan udskrivningen kommer til at se ud.

Listen kan indeholde forskellige ting, såsom variabler og konstanter af forskellige typer. Det er for eksempel ofte en blanding af tegnstrenge og variabler, som i

```
print *, "x er lig med", x
```

Normalt får tal så mange decimaler, som de har, og det kan være mere, end man har lyst til at se. Desuden er de placeret tilfældigt på linjen, så det er umuligt at lave en pæn tabel med denne form for `print`. Her er for eksempel sådan en udskrift af en 4*4 matrix, der blev udregnet ved at gange en matrix med dens invers. Det burde have været enhedsmatricen, men pga. afrundingsfejl er "off-diagonal"-elementerne ikke helt 0, og det kommer ud som

```
  1.00000     -9.536743E-07  1.907349E-06 -4.768372E-07
 1.788139E-06  0.999997     -1.907349E-06  5.662441E-07
-3.814697E-06  5.245209E-06   1.00000     -1.668930E-06
-2.050400E-05  2.193451E-05  2.956390E-05  0.999993
```

mens det, hvis man formaterer tallene, mere ser ud som man ville ønske:

```
 1.00000    0.00000    0.00000    0.00000
 0.00000    1.00000    0.00000    0.00000
 0.00000    0.00001    1.00001    0.00000
-0.00002    0.00002    0.00003    0.99999
```

Ligesom `read`-sætningen, begynder hver `print` en ny linje.

2.7 STOP

Hvis der i et program kan opstå en så alvorlig fejl, at der ikke er mening i at fortsætte, så er det oplagt at bruge `STOP`-sætningen. Den kan bruges alene, eller med en tekststreng, altså enten som, for eksempel

```
if (n > nmax) STOP
```

eller, med en besked,

```
if (n>nmax) STOP " n>nmax; abort."
```

Kapitel 3
Arrays

3.1 Definitioner

Det engelske ord **array** betyder anordning. Arrays er defineret som en samling, under et symbol, af et antal **skalarer**, alle af samme type og form. Oftest svarer arrays til de matematisk kendte vektorer, matricer osv., men ikke nødvendigvis. Man kan for eksempel lægge et antal studenternavne i et array af tegnstrenge. Arrays, der består af et antal **arrayelementer**, har et tilsvarende antal værdier, i modsætning til **skalarer**, som hver har en enkelt værdi (dog betragtes komplekse tal som skalarer, og i et senere kapitel bliver **records** beskrevet, som faktisk kan indeholde arrays, men regnes blandt skalarer). Der er desuden en række begreber, der skal nævnes. Det er muligt at henvise til en del af et array, kaldt et **arrayudsnit** (eng.: **array section**); hvordan det foregår, kommer senere. Det samlede antal elementer i et array hedder **størrelse** (eng.: **size**), mens **længden** (eng.: **extent**) er antallet af elementerne langs en bestemt dimension. For eksempel, et $3 * 4$ array har størrelse 12 og længde 3 i den første og længde 4 i den anden dimension.

Dimension (eng.: **rank**) er antallet af **dimension**er, dvs. i praksis antallet af **indeks** (eng.: **indices**) der er nødvendigt for at specificere hvilket element der menes i et givet tilfælde. **Form** (eng.: **shape**) er arrayets form, givet af dets dimension og dimensionernes længder. Således har en $3 * 4$ matrix samme størrelse (12) som, men en anden form end en $2 * 6$ matrix. De to matricer siges (på engelsk) ikke at være **conformable**, eller **konform**eller **kompatibel**. Ordet betyder at to arrays passer sammen eller kan lægges oven på hinanden. Det er kun arrays med samme størrelse og form, der kan det. Der er dog en undtagelse: skalarer er konform med alle arrays (se afsnit 3.6).

3.2 Erklæringer af arrays

For at gøre maskinen i stand til at sætte plads til side til et array i hukommelsen, skal man erklære det, med de fornødne oplysninger til oversætteren; det vil sige navn, dimension og de enkelte længder. Her er nogle eksempler på arrayerklæringer:

```
integer ::iarr(1000),jarr(0:1000),&
          karr(-100:100), larr(201)
real           :: x(100), A(10,10)
real,dimension(20,20) :: y, z, u, v
character(len=40) :: stud_navn(100)
```

Den første viser de muligheder man har med indeksgrænserne. Variablen `iarr` er erklæret med dimension 1 og længden 1000. Det betyder, at maskinen sætter plads til side for elementerne `iarr(1)` til `iarr(1000)`. Det første indeks er altsa lig med `1`, når det ikke er angivet ("per default"). Ofte vil man imidlertid starte med en anden værdi, meget ofte nul. I så fald skal der bruges erklæringsformen *lavere:øvre*, som kan ses for `jarr`, og for `karr`, hvis indekser går fra -100 til +100; størrelsen af `jarr` er altså 1001, og for `karr` er den 201. Arrayet `larr` belyser noget om konformitet: selv om `karr` har andre indeksgrænser end `larr`, er de to konforme, fordi de har samme dimension og længde. Når de to opererer med hinanden (se nedenfor hvordan), så parres elementerne `karr(-100)` med `larr(1)`, osv.

Selvom det måske virker mærkeligt at ville det, er det tilladt at specificere et array med længden 1 eller nul. Det har faktisk sine anvendelser, for eksempel i programmering af lineær algebra.

De to linjer med `real` arrays viser to forskellige måder at angive dimensioner på; det kan lade sig gøre på venstre eller højre side af `::`. Det nemmeste er normalt at gøre det på den højre side som vist i de første tre linjer i eksemplet. Så kan man specificere flere forskellige arrays i en enkelt linje. Hvis der derimod er flere med samme specifikationer, er det lettere at bruge `dimension(..)`-måden som vist i den næste linje, som ellers ville blive noget repetitiv.

Den sidste linje viser et eksempel på et array hvis elementer består af tegnstrenge. Også de er defineret som skalarer og kan derfor danne arrays.

Et array må højst have 7 dimensioner. Faktisk går man sjældent over 2 eller 3.

Når man har med arrays at gøre, bliver brug af `parameter` meget aktuelt, som følgende eksempel viser. Lad os sige, vi skal have de følgende arrays:

```
real :: a(10), b(1024), c(1025)
```

hvor 10-tallet sådan set er basis for de andre, som er 2^{10} og $2^{10} + 1$ i række. Hvis man senere fortry-

der, og vil udvide alle arrays til et nyt basistal 12, så må man jo ændre alle tre tal. Eksemplet er faktisk temmelig enkelt; sådanne afhængighedsrækker forekommer ret meget og kan omfatte langt mere end de tre arrays. Ved at bruge `parameter` kan man generalisere erklæringerne:

```
integer,parameter :: basis=10
real  :: a(basis),b(2**basis),c(2**basis+1)
```

eller, ved at bruge flere `parameter`:

```
integer,parameter :: basis=10,          &
                     tal=2**basis,      &
                     tal_plus_1=tal+1
real              :: a(basis), b(tal), &
                     c(tal_plus_1)
```

Hvis man nu vil skifte fra 10 til 12, behøver man i begge tilfælde kun at ændre en enkelt værdi (for `basis`); alle andre følger automatisk med.

3.3 Fysiske og aktuelle længder, elementorden

Normalt er det sådan, at der i et givet program er brug for arrays med de samme dimensioner, men forskellige længder fra en kørsel til den anden. Den simpleste måde at gøre dette på er at beslutte sig for en maksimumstørrelse i alle dimensioner; dvs. at lade oversætteren lægge tilstrækkelig meget plads til side for at dække alle mulige behov. I det overstående eksempel er `iarr` 1000 elementer lang; men ved en given kørsel bliver der muligvis brugt færre end alle de tusind. Tallet 1000 siges at være den **fysiske** længde af arrayet, mens det *aktuelle* brugte antal siges at være den **aktuelle** længde. Dette gælder selvfølgelig også for arrays med flere dimensioner. Man erklærer for eksempel et todimensionalt array (en matrix) som `A(10,10)`, men under en aktuel kørsel bruger man måske kun `A(3,4)`. At sætte maksimumgrænser er kun en måde, at gøre det på, den anden er allokering (se kap. 7).

Arrays gemmes væk (lagres) i maskinens hukommelse; brugeren behøver ikke at vide, hvordan og i hvilken rækkefølge. Man forestiller sig nok hukommelsen som en lineær følge af dueslag, hver med sin "adresse", altså i numerisk rækkefølge. Men det kan være ganske anderledes internt i maskinen, og kommer ikke brugeren (programmøren) ved. Men, man er nødt til at enes om en bestemt rækkefølge at gå et array igennem i. For eksempel, hvilket element skal være det næste efter `iarr(17)`? I dette tilfælde er det logisk, at det er elementet `iarr(18)`, og det er det. Der er dog andre muligheder for at gå et endimensionalt array igennem, element for element; man har bare valgt den logiske måde, dvs. stigende indekser for endimensionale arrays. Ved flerdimensionale arrays er der flere logiske muligheder, og forskellige programmeringssprog har da også valgt forskellige normer. I Fortran er reglen, at ved en gennemtælling af alle elementer, varierer indekserne for de dimensioner, der står til venstre, hurtigere end dem, de står til venstre for, i rækken af dimensioner angivet i en erklæring. For eksempel, hvis et array er erklæret som `A(10,20)`, så er rækkefølgen af alle elementer A(1,1), A(2,1), ..., A(10,1), A(1,2), A(2,2), ..., etc. Hvis man opfatter A som en matrix (som man ofte gør), betyder det at man tager arrayet en søjle ad gangen, startende med den til venstre. For arrays med flere dimensioner end to, skal man bare holde sig til reglen om, at den venstre dimension gennemgås hurtigst, osv.

Selvom der ingen garanti er for, hvordan sådan et array rent faktisk er placeret i maskinens hukommelse, er det højst sandsynligt, at det er placeret i samme rækkefølge som denne definerede, og logiske, orden. Vi kommer til at se i afsnittet om array I/O, hvorfor ordenen er vigtig.

3.4 Arraykonstanter

Ligesom for skalarer kan man have konstanter for arrays; de omfatter selvfølgelig et antal elementer. Der findes kun endimensionale. De er udtrykt som en følge, indrammet af `(/` og `/)`. Konstanten

```
(/ 17, 23, 11, 49 /)
```

specificerer et endimensionalt array af heltallene 17, 23, 11 og 49. Det kan bl.a. bruges til at initialisere et array med, dvs. give et array startværdier samtidigt med at det erklæres:

```
integer ::iarr(4)=(/ 17,23,11,49 /)
real    ::xarr(3)=(/ 1.0,2.0,5.0 /)
```

(Bemærk, at i tilfældet af det andet, REAL, array, er de enkelte elementer angivet som REAL, i og med at de har decimalpunkter efter). Flerdimensionale konstanter skal konstrueres af disse endimensionale, ved brug af functionen `RESHAPE` (se kap. 4 og Bilag A).

Der er endnu en måde at skrive en arraykonstant på, hvis dens elementer udgør en regulær progression. Hvis for eksempel det var konstanten `(/ 1, 2, 3, 4, 5 /)`, så kan man bruge en `implied-do` (se også afsnit 3.7) i stedet: `(/ i, i = 1, 5 /)`, hvilket giver det samme resultat. Det ligner en `DO`-løkke, og der er i eksemplet underforstået et spring med 1, altså `(/ i, i = 1, 5, 1 /)`. Derfor giver for eksempel konstantudtrykket `(/ i, i = 1, 5, 2 /)` det samme som `(/ 1, 3, 5 /)` osv.

3.5 Arrayudsnit

Det er muligt at henvise til en del af et array. Det engelske ord dertil er **section** eller (på dansk) **arrayudsnit**. Hvis man har erklæret et array, så mener man hele det erklærede array når man refererer til det ved blot at nævne dets navn (i det næste afsnit kommer vi til, hvordan man bruger sådan noget). Vil man derimod henvise til kun en del, skal man logisk nok angive, hvilken del man mener. Her har man mulighed for, for hver af dimensionerne af et array, at angive nogle eller alle ved *<første:sidste:hop>*, hvor *første* er det indeks man vil starte med, *sidste* er slutindekset, og *hop* er skridtlængden imellem. For eksempel betyder `1:10:2` elementerne 1, 3, 5, 7 og 9. Det er ikke påkrævet at angive skridtlængden, som bliver lig med 1 hvis man udelader den. Desuden kan man udelade et tal for at sige "alle": dvs. `:10` betyder alle op til 10, `1:` betyder alle fra 1 opad, og `:` betyder alle i den dimension. Her er nogle eksempler på arrayudsnit. Der forudsættes erklæringerne

```
integer :: iarr(10)
real    :: xarr(10,20)
```

og nogle muligheder er:

`iarr(1:10)`	hele `iarr`
`iarr(:5)`	svarer til 1:5
`iarr(2:7)`	kun rækken af elementer 2 til 7
`xarr(1:4,1:4)`	et 4*4 underarray inden for `xarr`
`xarr(1:10,2)`	alle 10 elementer i anden søjle i `xarr`
`xarr(1,:)`	alle (10) elementer i første række i `xarr`

Det ses, at man har mulighed for, at henvise til en enkelt række eller søjle af et todimensionalt array, dvs. til en endimensional struktur inden for en flerdimensional variabel.

Når man specificerer `xarr(1:4,1:4)`, så skaber man faktisk en ny størrelse; den har fysiske *og* aktuelle længder `4*4`. Det er vigtigt at være klar over dette forhold, når man bruger sådanne konstruktioner i sammenhæng med procedurer (kap. 4).

3.6 Arraysætninger og –operationer

Arrays kan manipuleres, enten som hele arrays, som udsnit eller med deres enkelte elementer. Det første er lettest, og ligner meget de operationer, man har med skalarer. Her er nogle arraysætninger af denne type, med kommentarer:

```
integer :: IA(101), JA(-50:50), &
           KA(0:100)
real    :: A(10,10), B(10,10), &
           C(20,20)
...
IA = 0      ! Hele IA nulstilles
JA = 1      ! Alle JA-elementer = 1
IA(1:4) = (/ 1, 2, 11, 17 /)
! (de foerste 4 = [1,2,11,17])
KA = IA + JA
! (Alle tre arrays kompatible)
A = 1       ! Alle 100 elementer = 1
B = 2       ! Alle 100 elementer = 2
C(:10,:10) =   A(:10,:10) &
             * B(:10,:10)
! (Tre kompatible arrayudsnit)
```

Det ses, at hele arrays kan gives værdier, ved at bruge tilordningssætninger ligesom for skalarer; i eksemplet bliver arrayerne `IA`, `JA`, `A` og `B` sat lig med nogle konstanter (0, 1 og 2). Dette betyder at hvert element af `IA` for eksempel får værdien 0 i den første sætning, osv. Dette er tilladt, fordi arrays er kompatible med skalarer. Den tredje sætning er interessant: de tre arrays `IA`, `JA` og `KA` har forskellige indekssekvenser, henholdsvis 1...101, -50...+50 og 0...100) men samme antal elementer (101), og de er derfor kompatible med hinanden. Den ene sætning svarer til de 101 sætninger,

```
KA(0)   = IA(1) + JA(-50)
KA(1)   = IA(2) + JA(-49)
...
KA(99)  = IA(100) + JA(49)
KA(100) = IA(101) + JA(50)
```

- eller (mere praktisk) til en løkke, se nedenfor. Det væsentlige her er, at når man opererer arrays med hinanden med sådanne aritmetiske sætninger (andre typer operationer er mulige), er det et antal operationer af de tilsvarende enkelte elementer med hinanden. Det samme gælder arrays med flere dimensioner, som det ses ovenfor med arrays A, B og C. Man skal altid passe på, at de arrays, man opererer sammen, er kompatible. Arrays A og B er kompatible (konforme) med hinanden, men ikke med C; derfor var det nødvendigt at bruge arrayudsnit. Strengt taget ville det have været nok at angive et arrayudsnit for C (`C(:10,:10)`), men i sådan en sætning er det god praksis at angive indeksrækker for alle arrays i sætningen, som vist. I praksis er det også ret sjældent, at man opererer med konstanter som dimensioner. Normalt er de jo kun maksimum-(fysiske) længder, og de aktuelle er noget mindre. Lad os sige, at der er N*N i stedet for 10*10 eller 20*20, og så ville den sidste af de ovenstående sætninger blive de tre kompatible arrayudsnit:

```
C(:N,:N) = A(:N,:N) * B(:N,:N)
```

Det ovenstående er den letteste måde at håndtere arrays på. Det kan imidlertid være nødvendigt at manipulere de enkelte elementer i et array, og så skal der løkker til. Et eksempel er, at sætte enhedsmatricen, dvs. et 2-D array hvis elementer alle er nul, undtagen diagonalelementerne, som skal have værdien 1. For array `A` (ovenfor):

```
A = 0
do i = 1, 10
  A(i,i) = 1
enddo
```

En sidste bemærkning om den ovenstående sætning, der ganger alle elementer i `A` og `B` og lægger produkterne ind i `C`, altså

```
C(:N,:N) = A(:N,:N) * B(:N,:N)
```

Det svarer ikke til matrixmultiplikation, som er defineret på en helt anden måde. Men, eftersom matrixmultiplikation jo er en operation, der hyppigt udføres, gøres der et lille sidespring her for at illustrere, hvordan mulighederne er. Matrixmultiplikation er defineret sådan, at det (i,j). element af produktmatricen `C` er summen af de elementvise produkter af række nummer i af `A` med søjle nummer j af `B`. Lad os antage, at matricen `A` er $n_A \times m_A$, og `B` er $n_B \times m_B$, og at produktet `C` bliver $n_A \times m_B$. (Det er klart, at $m_A = n_B$; det bør testes i praksis). Den elementvise måde at programmere det på er, for eksempel:

```
do i = 1, nA
  do j = 1, mB
    sum = 0
    do k = 1, mA
      sum = sum + A(i,k)*B(k,j)
    enddo
    C(i,j) = sum
  enddo
enddo
```

(Der er mere effektive måder at gøre det på, men denne er den lettest gennemskuelige). Alt dette kan erstattes af den indbyggede function `MATMUL`:

```
C(1:nA,1:mB) = MATMUL(A(1:nA,1:mA), &
                      B(1:nB,1:mB))
```

- hvilket er meget lettere. Eksemplet illustrerer både brugen af elementvis gennemgang af arrays, og helarrayoperationer. Det er især i forbindelse med matricer og vektorer, at man bliver glad for Fortran 90's helarraymuligheder. Et sidste eksempel: Der er en vektor `X`, med den aktuelle dimension N, og man vil have dens norm, dvs. summen af kvadraterne af alle dens elementer. Dette kan også beregnes med en enkelt sætning:

```
norm = SUM (X(1:N)**2)
```

hvor `SUM` er endnu en indbygget function. Set i detaljer er det, der sker her, at der (effektivt) dannes et nyt array, i hvilket hvert element er kvadratet af det tilsvarende element af `X`; og at alle disse kvadrater bliver lagt sammen til at danne `SUM`.

3.7 Input/output af arrays

Her skal man være opmærksom på den ovenfor diskuterede orden-overenskomst, ellers kan det gå galt. Vi starter, for at illustrere dette, med et array `A(3,4)`, som repræsenterer en matrix, og har fået følgende værdier:

$$A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

hvor de indsatte værdier klart blev valgt for at hentyde til de tilsvarende elementer af matricen `A` (a_{11}, a_{12}, osv.). Man kan udskrive hele matricen med sætningen

```
print *, A
```

(under antagelsen at `A` er erklæret med `(3,4)` som dimensionerne). Her er resultatet:

```
 11    21    31    12    22    32
 13    23    33    14    24    34
```

(der er blevet snydt lidt med afstandene). Tallene er skrevet i en lang række, nødvendigvis delt op i to linjer. Det vigtige er, at rækkefølgen ikke er den, vi muligvis forestiller os (rækkevis), men følger den ovenfor definerede arrayorden, dvs. en søjle ad gangen. Samme orden gælder selvfølgeligt, når vi vil indlæse værdierne. Det forekommer os mest naturligt at indtaste værdierne en række ad gangen. Dette kan lade sig gøre, for eksempel med tre sætninger for output, og brug af arrayudsnit:

```
print *, A(1,1:4)
print *, A(2,1:4)
print *, A(3,1:4)
```

hvilket giver det ønskede resultat. Oftest er arrays jo større, og så ville vi bruge løkker; antag, at `A` nu er `(N,M)`, og vores outputsætning bliver:

```
do i = 1, N
  print *, A(i,1:M)
enddo
```

- forudsat, at der er plads på linjerne til M tal. En tilsvarende løkke kan bruges til at indlæse en matrix, men med `print *` erstattet af `read *`.

Endnu en måde at udskrive nogle arrayelementer på er den såkaldte `implied-do`, som allerede blev nævnt i afsnit 3.4, i forbindelse med arraykonstanter. I betragtning af de muligheder arrayudsnit giver, kan det virke lidt overflødigt, men kan være et alternativ i visse situationer. Man kunne også udtrykke det ovenstående eksempel (`print`-linjen), som

```
print *, (A(i,j), j = 1, M)
```

som ligner en slags `DO`-løkke, og betyder `A(i,j)`, med `i` konstant, men `j` løbende fra 1 til `M` (underforstået, præcist som med `DO`-løkker, med spring interval lig med 1).

En `implied-do` kan have flere variabler løbende, som løkker kan. For eksempel svarer udtrykket `((i,j,i=1,3), j=1,2)` til sekvensen `1,1, 2,1, 3,1, 1,2, 2,2, 3,2` – altså alle `(i,j)` for `j` lig med 1 og `i` løbende (`1,2,3`), og derefter det samme for `j` lig med 2. Det bruges nok mest nu til at danne arraykonstanter (afsnit 3.4), og ikke så meget i forbindelse med `i/o`, da man har udsnit.

Den slags output, man får fra en `print *`, er ikke så pæn, eftersom maskinen bestemmer, hvor værdierne kommer til at stå på linjen. Det er ikke så slemt med heltal, men med realtal bliver det nok så grimt. Det har man mulighed for at styre, og det skal beskrives udførligt i kapitel 5, om filer og avanceret i/o.

Kapitel 4
Procedurer (underprogrammer)

En procedure er en stump programkode der ligger et eller andet sted og kan bruges til at udføre en given opgave ved at invokere stumpen. Det kan for eksempel være kvadratroden; hvis man i et program skal beregne kvadratroden af forskellige variabler flere steder, ville det være ineffektivt hver gang at skrive den kode der beregner roden, eftersom de samme trin skal gås igennem. I stedet skriver man sig en **procedure** og lader den udføre arbejdet hver gang. Det, der skal udføres, kan være mange forskellige ting; det kan være at beregne en enkelt numerisk værdi fra en eller flere variabler, eller transformere en eller flere værdier til andre, eller udskrive en samling informationer på en bestemt måde, eller give en information om noget. Der er altså en del forskellige typer procedurer, nogle af dem allerede i maskinen og nogle af dem programmeret af brugeren (programmøren) selv.

I det følgende bliver ordet **programenhed** (engelsk: **program unit**) brugt en del, og det skal forklares. Programmer kan deles op i selvstændige enheder, som enten kan behandles samlet, eller oversættes individuelt og senere kædes sammen. Hver enhed har et hoved og en afsluttende `END`-sætning. Der findes de tre enheder hovedprogram, procedure, og module. Hvis de alle er kædet sammen i en fil, skal modulerne komme først, efterfulgt af hovedprogram og procedurerne, alle "indrammet" med deres hoved- og `END`-sætninger.

Der er flere slags procedurer, og de kan deles i de tre: en **statement function**, en almindelig **function** og den mere generelle **subroutine**. Den første er ret uoversættelig; man kunne også kalde den for en enkeltlinjefunktion, men det virker for klodset og kunstigt; **function** er ligeledes bedre lige som det er, så kan den let skelnes fra en funktion; men **subroutine** kan godt oversættes til **underprogram**. Disse tre typer procedurer gennemgås herunder.

4.1 Statement functions

Det er den simpleste form for procedure i Fortran; en statement function må kun fylde en enkelt linje og være en function, hvilket vil sige, at den bruger en eller flere inputværdier til at producere en enkelt resultatværdi. Her er et eksempel:

```
program XYZ
  real :: x, y, arg, COT
  COT (arg) = 1.0 / TAN(arg)
  ...
  y = COT(x)
  ...
end program XYZ
```

Definitionen af statement function `COT` er placeret lige efter erklæringerne hvilket er påkrævet; den *skal* være mellem erklæringerne og den første udførbare sætning. Den er kun en linje lang; og den definerer en ny funktion, her cotangens, som ikke findes blandt de indbyggede functions. Den har et formelt argument `arg`, og det kan ses, at det bliver erstattet med den aktuelle variabel `x` i den sætning, der benytter sig af `COT`. Det vil sige, at variablen `x`'s værdi er sat ind i `arg`, funktionen $1/tan(arg)$ bliver evalueret, og resultatet er værdien af udtrykket på den højre side af sætningen `y = ...` og bliver sat ind i variablen `y`. Statementfunctionen bliver altså brugt bare ved at nævne (invokere) den.

Det man angiver som argument (i ovennævnte tilfælde `x`) kan være et hvilket som helst udtryk. Det vil sige, at en konstant eller et udtryk, der evalueres til en værdi, kan også bruges her. Restriktionen her er, at eftersom `arg` er erklæret som en `REAL`, så skal værdien også være af denne type, lige meget om argumentet er en variabel, konstant eller et kompliceret udtryk. For eksempel er de følgende kald af `COT` lovlige, og i alle tilfælde er resultatet, lige som argumentet, `real`:

```
y = COT (1.0)      ! REAL konstant
x = COT (3*x)      ! x REAL
y = COT(1+SQRT(x)) ! x REAL
```

Det er begrænset, hvad man kan med en statement function, når man kun har en linje at definere den på. I vores eksempel risikerer man at hele programmet går ned, når man kalder `COT` med et argument lig med nul. Der mangler en test i dette tilfælde, en `if`-sætning. Det er muligt med en rigtig **function**.

4.2 Functions

En **function** er en procedure som, ud fra et eller flere argumenter, producerer og afleverer en enkelt værdi tilbage, som er en funktion af disse argumenter. Ordet "værdi" betyder forskellige typer, inklusive array-værdier (men kun for visse slags functions, se afsnit 4.11). En **function** er en selvstændig programenhed og vil typisk være ekstern til et program der kalder den (men ikke nødvendigvis, se senere under **interne** procedurer). "Ekstern" her kan enten være ekstra tekst efter selve programteksten (men i samme fil), eller i en anden fil og oversat samtidigt med programmet. Et typisk eksempel er at functionen "hænger efter" programmet. Lad os udvide det ovenstående `COT`-eksempel til en rigtig function, med check for nul-argument, komplet med et hovedprogram, der invokerer functionen:

```
program ABC
  real :: x, y, COT
  ...
  y = COT (x)
  ...
end program ABC

real function COT (arg)
! Cotangensfunktion.
  real :: arg
  if (ABS(arg) > 1.0E-10) then
    COT = 1.0 / TAN (arg)
  else
    print *, "Nul argument!"
    COT = 0
  endif
end function COT
```

Læg mærke til `end`-sætningerne, og kommentarlinjen. Functionen har nu indbygget en test for nul-argument og sætter resultatet til nul i så fald. Hele definitionen af function `COT` ligger uden for hovedprogrammet. En interessant ting er, at symbolet `COT` repræsenterer noget der ligner en variabel; det får, for eksempel, en værdi i procedurens krop, lige som en variabel gør. En hyppig fejl er at skrive den linje som

```
COT (arg) = 1.0 / TAN (arg)
```

men det går ikke godt; det svarer nemlig til at functionen kalder (invokerer) sig selv – hvilket kan have mening, men ikke her. Enhver function skal altså et sted (eller flere steder) i sin definition få givet en værdi; det er denne værdi, den afleverer til det kaldende program.

Hvis vi indtil videre ser bort fra functions der må kalde sig selv (de **rekursive** functions), så gælder følgende regler:

- Functionens type bør erklæres både i det kaldende program, og i selve function-definitionen. De to typeerklæringer skal stemme overens.
- Typeerklæringerne for argumentet eller argumenterne skal ligeledes stemme overens.
- En function kan ikke kalde sig selv direkte eller indirekte (undtagen en rekursiv function).
- En function kan kalde andre functions.

Ordet "bør" i den første af reglerne skyldes at Fortran har implicit typing. Hvis man altså giver functionen et navn der implicit betyder typen REAL, så har den denne type. Dette er dog dårlig praksis. I eksemplet ovenfor er `COT` erklæret som værende REAL i hovedprogrammet `ABC`, og i definitionen af function, ved at `function`-linjen begynder med `real`. Alternativt kan man gøre det på en anden linje:

```
function COT (arg)
  real :: COT, arg
  ...
end function COT
```

Det er en smagsag. Det er også vigtigt at functionen og det kaldende program (rettere sagt: enhed) har samme opfattelse af, hvilken type argumentet har.

Tredje regel skal forklares lidt. En direkte kalden sig selv er klar; men et indirekte kald betyder, at functionen kalder en anden function, som kalder den første (eller en længere række af disse). Dette kan godt lade sig gøre, men ikke uden videre, og bliver diskuteret i kapitel 8, under **rekursive procedurer**.

Læseren har bemærket, at mens functionen blev invokeret med argumentet `x`, står der `arg` i selve definitionens krop. De to bliver ækvivalente, når functionen bliver invokeret. Symbolet `arg` har navnet "dummyvariabel", og kan altså stå for mange forskellige rigtige variabler eller udtryk. Her kommer begrebet **scope** ind, som oversættes til **rækkevide**. I det ovenstående eksempel er variablen `x` kendt i hele hovedprogrammet `ABC`, men ikke uden for hovedprogrammet. På samme måde er variabler, der muligvis bliver brugt inden for `COT` kun kendt der. Hvis `COT` for eksempel var skrevet sådan:

```
real function COT (arg)
! Cotangensfunktion.
  real :: arg, x
  if (ABS(arg) > 1.0E-10) then
    x = TAN (arg)
```

```
    COT = 1.0 / x
  else
    print *, ‘‘Nul-argument!’’
    COT = 0
  endif
end function COT
```

- så var x, som erklæret i `COT` en variabel lokal til selve `COT`, og har intet at gøre med variablen med samme symbol i hovedprogrammet. Forbindelsen mellem de to enheder – hovedprogrammet, og procedureren `COT` – er udelukkende skabt via det "overførte" **argument** som her forbinder `x` i hovedprogrammet med `arg` i functionen. Der er andre muligheder for at skabe forbindelser, ved hjælp af moduler, mere om disse nedenfor.

4.3 Subroutines – underprogrammer

En **subroutine** eller et **underprogram** skal *kaldes* med `CALL`. Oftest fører det, underprogrammet skal lave, til nye værdier, og disse, samt de, der skal bruges til at skabe dem, går alle mellem det kaldende program og underprogrammet via de overførte argumenter, eller via globale størrelser i moduler (se senere).

Ligesom en function, er et underprogram en selvstændig enhed med hoved og bund, og den formelle form er

```
subroutine <navn> (<p1>, <p2>, <p3>, ... )
  ...
end subroutine < navn>
```

hvor `p1` osv. er de formelle argumenter der opereres med. De kan normalt alle bare aflæses, eller også ændres af underprogrammet (i et senere kapitel vises der muligheder for at styre dette). Et underprogram skal kaldes ved en sætning som

```
  call <navn>  (<q1>, <q2>, <q3>, ...)
```

hvor `q1` osv. er de argumenter, underprogrammet skal operere med. Navnene på disse behøver ikke være (og er normalt ikke) de samme som dem specificeret i underprogrammet som formelle argumenter; de skal bare sættes ind i deres sted når underprogrammet bliver kaldt. Ligesom med functions skal hver især være af samme type som det tilsvarende. At navnene ikke er de samme betyder, at sammenkoblingen af et givet argument med dets modpart i underprogrammet skal foregå på anden vis; det er rækkefølgen, der giver denne sammenkobling. Her er et eksempel på et hovedprogram og et underprogram kaldt af det:

```
program ABC
  logical :: fejl
  real    :: x, y
  ...
  call COTANG (x, y, fejl)
  if (fejl) then
    ...
  endif
  ...
end program ABC

subroutine COTANG (arg, result, err)
! Beregn cotangens af arg, -> result. Er der
! en fejl, err = .true., ellers .false.

  logical :: err
  real    :: arg, result

  if (ABS(arg) < 1.0E-10) then
    err = .true.
  else
    err = .false.
    result = 1 / TAN(arg)
  endif
end subroutine COTANG
```

Der er tre overførte (dummy) argumenter, `arg`, `result` og `err`; og de svarer, ved kaldet, til de tre overførte, henholdsvis `x`, `y` og `fejl`. Alle er erklæret passende "begge ender". En af dem, `x` (eller `arg`) bliver kun "læst" af `COTANG`, mens de to andre, `y` (eller `result`) og `fejl` (eller `err`) bliver ændret af underprogrammet. De to er henholdsvis **input** og **output** til underprogrammet. Argumenter kan også være begge dele. Som med functions, kan argumenter være variabler, konstanter, eller udtryk for disse.

Underprogrammer bliver brugt til at dele et program op i overskuelige dele. En af de vigtigste ting ved programmering er overskuelighed, og at dele et program op i mere eller mindre lukkede selvstændige dele øger overskueligheden. En tommelfingerregel er, at når et program (eller et underprogram) er ved at blive større end der kan være på en side i print, burde det deles op i underprogrammer. Det vil altså sige, at et underprogram må godt i sig selv kalde et underprogram, osv. Men, lige som functions må et underprogram ikke kalde sig selv direkte eller indirekte, medmindre man har erklæret det til et rekursivt et (se kap. 8).

4.4 Argumentoverførsel: arrays

Især i forbindelse med overførsel af arrays til og fra procedurer, er det gavnligt at vide lidt om, hvad der sker, når noget overføres. Principielt er der flere muligheder, men i Fortran er der kun brugt en af dem.

Alle variabler er lagret i maskinens hukommelse; når et program opsøger en af dem, skal det være fastlagt, hvor i hukommelsen den er. Det er dens **adresse**, som kan opfattes som det unikke nummer, hukommelsesstedet har. Har man adressen, ved man, hvor variablen er. Det er adresserne, der bliver meddelt til procedurer. I det ovenstående eksempel, underprogrammet `COTANG`, som modtager variablerne `x`, `y` og `fejl`, er det altså disse variablers adresser, der bliver givet til `COTANG`. Konstanter eller udtryk, som jo også kan overføres, bliver evalueret ved procedurens kald, og får en (muligvis midlertidig) adresse, som er overført.

Problemerne dukker op når man vil overføre arrays. Hvad er adressen på et array? Hvor er elementerne af et flerdimensionalt array? Det første spørgsmål er let besvaret: et array er gemt væk i hukommelsen i en lang stribe, lige meget hvor mange dimensioner det har. Det er derfor logisk at overføre adressen på det første element, samt nogle oplysninger om formen (dimension) og muligvis længden. Det er her, at **orden** af elementerne kommer ind i billedet, og her, hvor forskellen mellem de erklærede og aktuelle dimensioner bliver relevant. Vi tager det fra den lette ende, 1-D arrays først.

Endimensionale arrays

Et endimensionalt array har fysiske længder (som erklæret), og aktuelle, som varierer fra kørsel til kørsel. Men med disse arrays behøver man som regel kun fortælle et underprogram om de aktuelle, eftersom alle elementer jo i forvejen er en lang stribe. Underprogrammet behøver ikke at "vide", at der er flere elementer end det antal, det er bedt om at gøre noget ved. Derfor kan man her nøjes med adressen og den aktuelle længde. Det kan gøres på forskellig vis: her et kort eksempel:

```
program UVW
  integer :: N
  real    :: x(100)
  ...
  N = 20
  call UNDER (x, N, ...)
  ...
end program UVW

subroutine UNDER (arr, N, ...)
  integer :: N
  real    :: arr(N)
  ...
end subroutine UNDER
```

Det ser ud til, så vidt `UNDER` får at vide, at `arr` har `N` elementer. Men man må godt erklære `arr` i underprogrammet som

```
  real    :: arr(*)
```

eller bare

```
  real    :: arr(:)
```

eller

```
  real, dimension(:) :: arr
```

Tegnet `*` eller : siger til `UNDER`, at det er et array med en dimension; antallet N er også kendt, eftersom det blev overført. Sådan en erklæring lader det være underforstået, at indekserne af arrayet starter med tallet 1. Vil man noget andet, skal det angives. Man begynder tit arrays med indeks nul:

```
  real    :: z(0:100)
```

og det skal et underprogram også specificere:

```
  real    :: arr(0:*)
```

eller

```
  real    :: zarr(0:)
```

Man kunne nøjes med, som, i eksemplet, at erklære `arr` i underprogrammet som

```
  real    :: arr(100)
```

Men nu står 100-tallet to gange; en gang i hovedprogrammet, og igen i underprogrammet. Hvis man nu fortryder og vil lave tallet om til 1000, skal man huske at lave det om i begge programenheder. Lidt senere kommer vi til modules, som gør alt dette automatisk, men her nøjes vi med at sige, at den uspecificerede arraylængde gør programmer mere fleksible. Det er nemlig desuden sådan, at mange underprogrammer er skrevet af andre end den, der skriver hovedprogrammet, og oversættes særskilt. Der burde altså ikke være nogen unødvendig deling af information (100-tallet).

En effekt af disse uspecificerede dimensioner er, at man inden for et underprogram ikke kan bruge et arraynavn alene i et udtryk, som

```
  ... = arr
```

fordi det jo ikke er klart hvor meget af `arr` man her mener. Her skal man bruge arrayudsnit, som `arr(1:N)`, hvilket løser problemet.

Der er visse finesser i Fortran 90, som kan gøre det ovenstående endnu mere fleksibelt og generelt. En af dem (modules) nævnes i afsnit 4.9 nedenfor, og andre (`ALLOCATE`, `SIZE`, mm.) i kapitel 7.

Hvor starter et overført array?

Indtil nu er arrays blevet overført ved deres navn, hvilket betyder, at arrayet bliver brugt (behandlet) i underprogrammet, der modtager det, fra det allerførste element af, om det nu er indeks 0, 1 eller noget andet. Men man kan også specificere et stykke af et array ved at referere til et bestemt element ved kaldet. Et kald som

```
call SUB (X(20), N)
```

bliver modtaget af underprogrammet `SUB` som om arrayet starter med element nr. 20; men underprogrammet opfatter det som det første (eller nulte, alt efter hvordan det er erklæret der), og `N` derfra. På samme vis kan man også overføre et enkelt element, hvis det altså i underprogrammet er erklæret som en skalar. Alt dette er klart, hvis man husker, at det er adressen, der bliver overført til et underprogram.

Flerdimensionale arrays

Når et array har flere dimensioner, bliver dets overførsel til et underprogram lidt mere indviklet. For at forstå hvorfor, skal man mindes om, hvordan disse arrays (antagelsesvis) er ordnet i hukommelsen. Lade os sige, at der er et array A, erklæret som

```
real :: A(20,20)
```

og at vi i en given kørsel kun benytter det som et $5 * 5$ array. Problemet er: hvor i hukommelsen befinder sig nu et givet element? Når der henvises til et vist element, skal maskinen jo finde det. For eksempel, hvor i den orden, elementerne af `A` er ordnet i, er `A(2,3)`? Det blev nævnt i kapitel 3, at todimensionale arrays er ordnet en søjle ad gangen, i dette tilfælde altså i søjler af 20 stk. Element (2,3) er (af os) opfattet som værende i tredje søjle og anden række. Ifølge maskinens ordning af arrayet placeres elementet altså i position $2 * 20 + 2$ (regn selv efter!). Mere generelt, hvis et array er erklæret med fysiske længder `(Nmax,Mmax)` og der ledes efter element `(i,j)`, så er positionen `k` af dette element beregnet med formlen

$$k = (j - 1) * Nmax + i$$

(regn selv efter igen!) Det ses, at i hvert fald `Nmax` spiller en rolle. For at et underprogram kan finde de rigtige elementer, skal det derfor kende de fysiske længder. Den simpleste (og mest klodsede) måde at løse problemet på er, for et todimensionalt array, at overføre fire tal, som i dette eksempel:

```
program XYZ
  integer :: N, M
  real    :: A(20,20)
  ...
  call SUBROUT(A,20,20,N,M,...)
  ...
end program XYZ

subroutine SUBROUT &
   (X, Ndecl, Mdecl, N, M, ...)
  integer :: Ndecl, Mdecl, N, M
  real    :: X(Ndecl,Mdecl)
  ...
end subroutine SUBROUT
```

De to fysiske længder bliver overført som konstanter, samt de aktuelle, `N` og `M`. Denne måde at gøre det på bliver hurtigt kedelig, især når der overføres flere arrays ved et kald. Der er bedre måder, og de bliver forklaret i et senere afsnit i kapitlet. Den vågne læser vil nok have bemærket, at længden `Mmax` slet ikke optræder i det ovenstående regnestykke. Generelt gælder for flerdimensionale arrays, at den sidste dimension i rækken ikke er brugt, og formelt derfor ikke behøver at blive overført. Man kunne altså godt gøre eksemplet sådan:

```
program XYZ
  integer :: N, M
  real    :: A(20,20)
  ...
  call SUBROUT (A, 20, N, M, ...)
  ...
end program XYZ

subroutine SUBROUT(X,Ndecl,N,M,...)
  integer :: Ndecl, N, M
  real    :: X(Ndecl,:)
  ...
end subroutine SUBROUT
```

- men det gavner ikke særlig meget.

Der er en anden mulighed for at overføre kun den del af et array, man vil have behandlet, ved brug af arrayudsnit. Hvis vi for eksempel kun vil give `SUBROUT` lige så meget af `A` som underprogrammet skal tage sig af, kan man kalde det sådan:

```
  call SUBROUT (A(1:N,1:M), N, M, ...)
```

i hvilket tilfælde man kan skrive underprogrammet lidt mere kortfattet:

```
subroutine SUBROUT(X,N,M,...)
  integer :: N, M
  real    :: X(N,M)
  ...
end subroutine SUBROUT
```

eller sågar

```
subroutine SUBROUT(X,N,M,...)
  integer :: N, M
  real    :: X(N,:)
  ...
end subroutine SUBROUT
```

Det kan lade sig gøre, fordi et arrayudsnit er en selvstændig størrelse, som har de fysiske længder, der er specificerede, her `(1:N,1:M)`, og de er ens med de aktuelle.

4.5 Argumentoverførsel: tegnstrenge

Der er nogle særlige tricks man kan anvende i forbindelse med overførsel af tegnstrenge, som er værd at nævne. En tegnstreng består af en følge af tegn (typen `character`), og man erklærer sådan en med en given længde. Oftest fyldes den med færre tegn, end det tilladte (erklærede) maksimum antal; dvs., der er en hale af ophold (spaces). Fortran 90 har nogle indbyggede functions for at finde ud af, både hvilken længde en streng er erklæret med, og hvor lang den ikke-tomme del faktisk er. Disse functions gør det muligt at overføre en tegnstreng, uden at angive disse størrelser til proceduren. Her er et eksempel på, hvordan det fungerer:

```
  ...
  character(LEN=100) :: s
  ...
  call CHARSUB (s, ...)
  call CHARSUB ("En tegnstreng",...)
  ...

subroutine CHARSUB (str, ...)
  character(LEN=*) :: str
  integer          :: fysL, aktL

  fysL = LEN(str) ! fysiske laengde
  aktL = LEN_TRIM(str) ! klippet
  ...
end subroutine CHARSUB
```

I selve underprogrammet er variablen `str` erklæret som typen `character`, men med en uspecificeret længde (`LEN=*`). Rutinen finder ud af begge tal ved at kalde de to indbyggede functions `LEN` og `LEN_TRIM`. Bemærk at `CHARSUB` bliver kaldt to gange; første gang med `s`, som er en streng med maksimalt 100 tegn, og derefter med en tegnstrengkonstant, angivet i anførselstegn. Den vil blive overført som en tegnstreng både erklæret med maksimum 13 tegn (tæl selv efter!), og en aktuel længde også lig med 13.

4.6 Et nyttigt eksempel: underprogrammet MATOUT

Her gives der et eksempel på et underprogram, der kombinerer overførsel af et todimensionalt array (en matrix) samt en tekst. Det er noget der kan bruges i programmer med matrixregning. På dette stadium i bogen er eksemplet ikke helt heldigt, men det skal forbedres i senere afsnit, når de relevante redskaber bliver behandlet.

```
subroutine MATOUT &
    (streng, arr, Nmax, Mmax, N, M)
! Dette underprogram udskriver, paa
! standardenheden (skaerm),
! matricen ARR, ved simpel print,
! samt en tekst i foerste linje.

  implicit none

  character(LEN=*) :: streng
  integer          :: Nmax,Mmax,N,M
  real             :: arr(Nmax,Mmax)

  integer          :: L, i

  L = LEN_TRIM (streng)
  print *, streng(1:L)
  do i = 1, N
    print *, arr(i,1:M)
  enddo

end subroutine MATOUT
```

Et lille testprogram blev skrevet for at teste denne rutine:

```
program MATOUT_TEST
  real :: A(10,10)
  A(1,1:4)=(/11.0,12.0,13.0,14.0 /)
  A(2,1:4)=(/21.0,22.0,23.0,24.0 /)
  A(3,1:4)=(/31.0,32.0,33.0,34.0 /)
  call MATOUT ("En 3*4 matrix", &
                A, 10, 10, 3, 4)
end program MATOUT_TEST
```

og en kørsel af dette kunne give følgende output:

```
 En 3*4 matrix
   11.0000  12.0000  13.0000  14.0000
   21.0000  22.0000  23.0000  24.0000
   31.0000  32.0000  33.0000  34.0000
```

Der er et par ting, som er utilfredsstillende. Den simple `print *` sætning gav ingen mulighed for at bestemme hvor meget plads der er mellem tallene, og vi ville nok ikke have valgt at have alle nullerne med. Og, som nævnt før, virker det lidt klodset at skulle overføre hele fire tal for at specificere dimensionerne af arrayet. Alle disse mangler kan vi gør noget ved, som det skal vises.

4.7 Argumentoverførsel: procedurer

Ligesom variabler, som har (eller får) en given værdi, kan der også overføres navne på procedurer, og også de får nye, formelle ("dummy"-) navne. Et godt eksempel på brug af denne facilitet er et underprogram, som beregner integralet af en hvilken som helst funktion (selvsagt, mellem to grænser). Når man kalder den procedure, skal man fortælle den, hvilken function der skal integreres. Lad os sige, at vi har defineret en function med navnet `FUNC`, og at proceduren `PROC` skal bruge den. Det ser sådan ud:

```
  ...
  call PROC (FUNC, ...)
END ...

subroutine PROC (FN, ...)
  external :: FN
  ...
END subroutine PROC
```

Her kender `PROC` den overførte function `FUNC` under det generelle navn `FN`. For at functionen bliver anerkendt *som* procedure, skal sætningen med `external` være der, hvilket betyder at navnet specificerer en procedure. Resten af indholdet af `PROC` benytter sig så af `FN`, som om det er navnet.

4.8 Interne *vs* eksterne procedurer

I de foregående afsnit i dette kapitel blev kun de såkaldte **eksterne** (engelsk: **external**) procedurer nævnt. De hænger normalt efter et program, men de kan også være (og er meget tit) helt uafhængige enheder, som bliver oversat særskilt; de kan typisk være skrevet af en anden person end den, der skriver den enhed (program eller procedure) der kalder dem. De er altså lukkede enheder, som intet "ved" om hvad der er i de andre enheder der kalder dem. Al kommunikation mellem dem og de kaldende enheder foregår (hovedsagelig, men jævnfør modules nedenfor) gennem de overførte argumenter.

Der er også **interne procedurer**. En intern procedure er en, der ligger inde i en programenhed, og er altid placeret sidst i enheden, lige før `END`-sætningen. Den skal angives som en intern procedure med `CONTAINS`-sætning, for eksempel:

```
program XYZ
  ...
  contains
  <procedure definition>
end program XYZ
```

Der må godt være flere end en procedure, og de må være både functions og subroutines.

Reglerne her er

- En intern procedure kan kun kaldes fra den enhed, den er intern i.
- En intern procedure har kendskab til alle variabler og modules i den enhed den er "gæst" i, medmindre den interne procedure erklærer sine egne variabler af samme navne.
- Interne procedurer må ikke i sig selv indeholde interne procedurer.
- Mens `END`-sætninger normalt må stå som bare `END`, er det påkrævet at hænge enten `function` <*navn*> eller `subroutine` <*navn*> efter `END`, med interne procedurer.

Interne procedurer er altså enheder, som er afskåret totalt fra andre end deres "gæstgiver"-enheder. Det tillader for eksempel, at forskellige enheder kan have interne procedurer i sig med de samme navne, men forskellige funktioner. Det lyder muligvis ikke så interessant, men kan være det, hvis man er en del af et større team der skriver et stort program. Man kan ikke være sikker på, at andre ikke vælger nogle symboler for procedurer, man selv har valgt. Hvis de procedurer ikke har interesse for andre og ikke skal bruges i andre eksterne enheder man selv skriver, er de bedst placeret internt. Eftersom en intern procedure jo har fri adgang til alt i den ydre enhed, behøver den ikke at kaldes med argumenter – de er jo kendt i forvejen. Der opstår dog situationer, hvor man vil foretrække at overføre nogle argumenter. Sidst, men ikke mindst, udføres interne procedurer hurtigere end eksterne.

4.9 Modules

Som nævnt, findes der yderligere en enhed med navnet **modul** (engelsk: **module**). Ved hjælp af moduler kan man, i andre enheder som f.eks. hovedprogram eller procedurer, vælge eller fravælge at gøre information i modulen kendt (tilgængelig). Det bruges her først til `PARAMETER`-konstanter; moduler kan bruges til meget andet, men det er gemt til senere kapitler. Det blev klart i forbindelse med todimensionale arrays, at overførsel af disse til procedurer var lidt af et problem, med de mange størrelser, procedurer skal kende. Hvis man i et program bruger mange arrays, hvis dimensioner oftest hænger sammen, så er moduler vejen. Et eksempel gør det klart. Vi gentager eksemplet fra før, men nu med en module:

```
module DEFS
  integer,parameter :: Nmax=20,
  integer,parameter :: Mmax=20
  real,parameter    :: pi=3.14159
END module DEFS

program XYZ
  use DEFS
  integer :: N, M
  real    :: A(Nmax,Mmax)
  ...
  call SUBROUT (A, N, M, ...)
  ...
end program XYZ

subroutine SUBROUT (X, N, M, ...)
  use DEFS
  integer :: N, M
  real    :: X(Nmax,Mmax)
  ...
end subroutine SUBROUT
```

Bemærk modulen, som definerer de to heltalskonstanter `Nmax` og `Mmax`, samt tallet π. Disse konstanter er tilgængelige i alle de enheder, der indeholder (som første erklæringssætning) sætningen

```
  use DEFS
```

hvor `DEFS` er modulens navn. Brug af `USE` bevirker, at de erklæringer der er i modulen, er aktive i enheden. På denne måde kan flere enheder dele definitioner i modulen, og det betyder, at vi ikke længere behøver at overføre de fysiske længder `Nmax` og `Mmax` (dvs., konstanterne 20 og 20) ved kald af procedurer. Det sparer ikke bare indtastning, det gør andre ting mulige. Det sker ikke så sjældent, at man inde i en procedure har brug for ekstra arrays, hvis størrelse afhænger af størrelsen af et overført array, eller flere af dem. Hvert array skal jo erklæres en gang for første gang, med rigtige dimensioner. Når man overfører et array til en procedure, er det allerede erklæret, og proceduren skal bare have de nødvendige oplysninger om, hvordan arrayet er erklæret i forvejen. Men et lokalt array i proceduren skal erklæres der. Hvis man dertil bruger konstanter, er der en skjult sammenhæng med den kaldende programenhed. Hvis man kobler en procedure til en modul med `USE`, kan man bruge de konstanter der er fastsat i modulen, og oprette nye lokale arrays. For eksempel:

```
module DEFS
  integer,parameter :: Nmax=20
  integer,parameter :: Mmax=20
  real,parameter    :: pi=3.14159
END module DEFS

program XYZ
  use DEFS
  integer :: N, M
  real    :: A(Nmax,Mmax)
  ...
  call SUBROUT (A, N, M, ...)
  ...
end program XYZ

subroutine SUBROUT (X, N, M, ...)
  use DEFS
  integer :: N, M
  real    :: X(Nmax,Mmax)

  real    :: Help_Vector (2*Nmax),&
             Help_Array (Nmax,Mmax)
  ...
end subroutine SUBROUT
```

Det fordelagtige her er, at man let kan ændre værdien af en given konstant en gang for alle, ved at ændre den i modulen; alle enheder følger så med (efter en ny oversættelse, naturligvis). Moduler kan indeholde andre ting end konstanter (og andre slags konstanter, for eksempel, `KIND`-konstanter, beskrevet i kapitel 6). Der kan være interne procedurer, som bliver tilgængelige for andre enheder der kobles til modulen. Noget lidt mere indviklet er de såkaldte **interface**s (se kapitel 8).

4.10 Biblioteker

Procedurebiblioteker består som regel af procedurer, der bliver brugt i mange forskellige programmer. Der findes for eksempel procedurer til at løse ligningssystemer eller differentialligninger, eller procedurer der udfører Fourier-transformationer; der er et hav af professionelle underprogrammer at benytte sig af, de fleste gennemtestede og optimale. Det er endda muligt at koble programmer sammen med procedurer skrevet i andre sprog end Fortran. Mange programmører udvikler desuden deres egne rutiner, eftersom de har brug for de samme procedurer i mange forskellige programmer. Disse procedurer bliver lagt sammen i færdigoversatte biblioteker. Hvordan disse oprettes og bruges i praksis, er forskelligt fra maskine til maskine og fra operativsystem til operativsystem. Sådanne (meget) eksterne procedurer kan ikke dele ens egne moduler, men man kan bruge moduler til at sikre sig (via interfaces, se kapitel 8), at man kalder dem ordentligt.

4.11 Indbyggede (intrinsic) procedurer

I Fortran 90 er der indbygget en del procedurer man bare kan kalde (eller invokere) uden selv at definere dem. Vi har allerede set nogle eksempler på disse, såsom `SIN` (sinus), og `SQRT` (kvadratrod). De udgør gruppen af **intrinsics**, hvilket kan oversættes til **indbygget**. Der er functions og underprogrammer, og de er delt op i grupper, efter hvad de udfører; de har at gøre med tal, tegn, systemparametre, typer, eller arrayegenskaber med mere. Her kommer der to nye begreber ind: **generisk** (engelsk: **generic**) og (kun engelsk, oversættes ikke her) **elemental**.

Generiske functions

Visse functions kan tage tal af forskellige typer som argument, og resultatet retter sig efter typen af argumentet. For eksempel kan man tage kvadratroden af en enkel `REAL` værdi, eller af en med dobbelt præcision, eller af et komplekst tal; roden får samme respektive type. Forklaringen er, at den intrinsic function `SQRT`, når den kaldes med de forskellige typer argument, i sig selv udvælger den rigtige function for at løse opgaven (i dette tilfælde, henholdsvis enten `SQRT`, `DSQRT` eller `CSQRT`). Programmøren behøver ikke at vide, at de er der; valget foregår automatisk.

Elementals

Intrinsic functions har mulighed for at udføre deres funktion på alle elementer af et array. Det er parallelt med en sætning som

```
A = 1
```

hvor alle elementer af arrayet A bliver sat til 1. Kalder man en function som, for eksempel, `SIN` eller `SQRT` med et array som argument, så er resultatet et nyt array, hvori hvert element er resultatet af functionens opereren på det tilsvarende element i argumentarrayet. Det vil sige, at elementerne bliver behandlet individuelt. Blandt de elemental functions er der `SQRT`, `EXP`, `LOG`, `SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, osv. (tabellerne specificerer denne egenskab).

Grupper af intrinsics

Intrinsic procedurer kan deles i grupper. Der er både functions og subroutines. De er, delt op i grupper:

Numeriske functions, matematiske functions, tegnmanipulerende (character) functions, `KIND`-functions, en enkelt `logical` function, inquiry functions, bitmanipuleringsfunctions, array functions, vektor- og matrixfunctions, intrinsic subroutines og de nye Fortran 95 procedurer. Forskellen mellem de numeriske og matematiske functions er, at de numeriske udfører konverteringer (for eksempel `real` til `integer` eller omvendt), eller andre simple operationer med tal, mens de matematiske beregner funktioner af argumenter. De såkaldte **inquiry functions**, eller spørgsmålsstillende functions besvarer spørgsmål om variablerne eller systemparametre stillet af programmøren. For detaljer, se Bilag A. Her følger lister over intrinsic procedurer, delt op i grupper. For detaljer, se Bilag A.

De **numeriske functions** er alle **generic** og **elemental**:
`ABS`, `AIMAG`, `AINT`, `ANINT`, `CEILING`, `CMPLX`, `CONJG`, `DBLE`,
`DIM`, `DPROD`, `FLOOR`, `INT`, `MAX`, `MIN`, `MOD`, `MODULO`, `NINT`, `REAL`, `SIGN`.

De **matematiske functions** er alle **generic** og **elemental**:
`ACOS`, `ASIN`, `ATAN`, `ATAN2`, `COS`, `COSH`, `EXP`, `LOG`, `LOG10`, `SIN`, `SINH`, `SQRT`, `TAN`, `TANH`.

Tegn-functions er alle **generic** og **elemental**, undtagen `REPEAT` og `TRIM`, som ikke er elemental:
`ACHAR`, `ADJUSTL`, `ADJUSTR`, `CHAR`, `IACHAR`, `ICHAR`, `INDEX`,
`LEN`, `LEN_TRIM`, `LGE`, `LGT`, `LLE`, `LLT`, `REPEAT`, `SCAN`, `TRIM`, `VERIFY`.

KIND-functions:
`KIND`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`.

Der er en enkelt **LOGICAL function**; den er generic og elemental, og den har navnet `LOGICAL`.

Inquiry functions giver svar på inquiries, dvs. på forespørgsler. De er af naturen generiske:
`ALLOCATED`, `ASSOCIATED`, `BITSIZE`, `DIGITS`, `EPSILON`, `EXPONENT`, `FRACTION`, `HUGE`, `LBOUND`, `MAXEXPONENT`, `MINEXPONENT`, `NEAREST`, `PRECISION`, `PRESENT`, `RADIX`, `RANGE`, `RRSPACING`, `SCALE`, `SET_EXPONENT`, `SHAPE`, `SIZE`, `SPACING`, `TINY`, `UBOUND`.

Functions til bitmanipulation behandler enkelte bits inden for tal. De opererer alle på heltal og er alle elementals:
`BTEST`, `IAND`, `IBCLR`, `IBITS`, `IBSET`, `IEOR`, `IOR`, `ISHFT`, `ISHFTC`, `NOT`.

Arrayfunctions har alle et resultat med færre dimensioner end argumentet, ofte skalar. De første tre opererer på arrays af typen `logical`. De sidste fire opererer med alle elementer i arrayet, muligvis indskrænket af de valgfrie argumenter, `DIM` og `MASK`:

ALL, ANY, CSHIFT, COUNT, EOSHIFT, MERGE, MAXLOC, MINLOC, MAXVAL, MINVAL, PACK, PRODUCT, RESHAPE, SPREAD, SUM, TRANSFER, TRANSPOSE, UNPACK.

Vektor- og **matrixfunctions**: DOT_PRODUCT, MATMUL.

Til sidst er der nogle **intrinsic subroutines**. Disse adskiller sig fra de ovenstående i at de ikke er functions, men subroutines, og skal kaldes med CALL ...:
DATE_AND_TIME, MVBITS, RANDOM_NUMBER, SYSTEM_CLOCK.

Der er desuden de nye **Fortran 95 procedurer**, CPU_TIME og NULL; og de som er ændret fra Fortran 90, CEILING, FLOOR, MAXLOC, MINLOC og SIGN.

Kald af intrinsics

Alle intrinsic procedurer kan kaldes på to måder, enten ved "positional arguments" eller ved "argument keywords". Den første af de to er den, der er blevet vist indtil nu. For eksempel tager functionen ATAN2 to argumenter, y og x. Man kan kalde den ved ATAN2(y,x), hvor det er deres position i rækkefølge, der meddeler functionen, hvilken der er y (den første), og hvilken der er x (den anden). Men man kan også kalde functionen ved for eksempel ATAN2(y=1.0,x=1.0) (eller med udtryk for værdierne, hvad de ellers skal være). I dette tilfælde er y og x såkaldte keywords, specifikke for functionen; man kan altså ikke opfinde sine egne navne, de er fastlagt. Det virker muligvis lidt klodset. Der er imidlertid en fordel ved det: for det første kan man bytte rundt på rækkefølgen, hvilket ikke er så meget værd i sig selv. Men, hvad der er langt mere brugeligt, man kan udelade nogle argumenter i de procedurer, hvor nogle argumenter er valgfrie. Et godt eksempel er proceduren DATE_AND_TIME, som har de fire keywordargumenter DATE, TIME, ZONE (alle tre af typen character) samt VALUES (integer). Alle fire er valgfrie. Her er nogle mulige måder at kalde denne procedure:

```
  ...
  character(LEN=10) :: dato, tid, z
  integer           :: tal(8)
  ...
! Positional, alle:
  call DATE_AND_TIME (dato, tid, z, tal)
! keyword, kun dato:
  call DATE_AND_TIME (DATE=dato)
! keyword, kun zone:
  call DATE_AND_TIME (ZONE=z)
! keyword, tid og dato
  call DATE_AND_TIME (TIME=tid,DATE=dato)
```

Kapitel 5
I/O formats og filer

5.1 Formats

I de foregående kapitler brugtes der kun de to I/O-sætninger, `read` og `print`, begge med et '`*`' bag. Det tegn siger, at systemet selv skal finde ud af (for en `read`), hvordan et givent input kommer fra inputenheden (oftest tastaturet) eller (for en `print`) bestemme hvordan det kommer til at stå på skærmen. I kapitel 4 blev der vist et eksempel på et underprogram, der printer en matrix ud, og der kunne ses en del uønskede nuller.

Når man selv vil bestemme formatet af et output, eller specificere formatet af et input, skal man altså have en formatspecifikation med i I/O-sætningen. De generelle former for de to slags sætninger (vi holder os foreløbigt til tastatur og skærm) er:

```
read <fmt>, <liste>
print <fmt>, <liste>
```

hvor <*fmt*> nu erstatter tegnet '`*`'. Der er to måder at specificere formatet på: enten på stedet som en tegnstreng, eller som en henvisning til en `format`-sætning med en label, som skal være et tal. De to har hver sine fordele. Et simpelt eksempel giver en ide om mulighederne: lad os sige, vi har et antal heltal i et array `tal(1:4)`: 10, 11, 12, 13; og vi vil printe dem. En sætning som

```
print *, tal(1:4)
```

giver

```
        10          11          12          13
```

hvor antallet af mellemrum mellem tallene blev bestemt af maskinen. Det kan være, at man vil pakke tallene tættere sammen. Så kan vi skrive

```
print '(4i4)', tal(1:4)
```

og resultatet bliver nu

```
10  11  12  13
```

Det samme får man, hvis man bruger en `format`-sætning:

```
  print 100, tal(1:4)
100 format (4i4)
```

I begge eksempler er strengen `'4i4'` den, der specificerer formatet. 100-tallet i den variant, der benytter en `format`-sætning, er den såkaldte **label**, som mærker linjen med `format`-specifikationen. Sådan en linje skal altid starte med et tal som label, ordet `format` og specifikationen indrammet af parenteser, som vist. Bruger man den anden form, indrammes specifikationen af de samme parenteser og yderligere af et par anførselstegn (enkelte eller dobbelte). Fordelen ved at bruge en tegnstreng umiddelbart efter `print` er klarhed; læseren af programmet kan straks se, hvilket format der skal udskrives med. Brugen af en `format`-sætning gør det nødvendigt først at lede efter den (den kan stå langt fra I/O-sætningen); men fordelen er, at hvis der er flere `print`-sætninger, alle med den samme formatspecifikation, behøver man kun skrive den en enkelt gang, og alle `print`s kan henvise til den via dens label.

Der er forskellige vaner for hvordan `format`-sætninger er placeret. De er kun tilgængelige i den enhed, hvor de er specificerede, men kan stå hvor som helst i enheden, blandt de udførbare sætninger. Det vil sige, de må ikke stå foran en erklæringssætning, og de skal komme før en eventuel `contains`- eller `END`-sætning. Mange programmører samler dem i en gruppe i bunden af en enhed.

En formatspecifikation (på stedet eller i en formatsætning) indeholder en række såkaldte **format items**, hvilket ikke oversættes her. Disse består af **edit items**, som skal beskrives udførligt efterfølgende, og tegnstrenge som skal (for eksempel) printes ud sammen med variabelværdier. De ovenstående fire tal kunne således printes med en kommentar i form af en tegnstreng:

```
print '(" De fire tal er:", 4i4)', tal(1:4)
```

hvor de to "-tegn fungerer fordi hele formatstrengen er indrammet af enkelte; eller som

```
print "("" De fire tal er:"",4i4)",tal(1:4)
```

hvor indramning er af samme dobbelte og de derfor skal indtastes to gange hvert sted. Det letteste er at indramme med en, når den anden skal bruges. I begge tilfælde er resultatet

```
 De fire tal er:  10  11  12  13
```

Vi har set en enkelt edit item, `4i4`, hvilket betyder fire gange `i4`. Editering, i denne sammenhæng, hentyder til, at værdiernes udseende, når de udskrives eller indlæses, skal bestemmes efter opskrift. Der er en række edit items, for de forskellige typer. I alle tilfælde hvis, ved udskrivning, det der skal udskrives ikke passer i det specificerede format, bliver der udskrevet en række `'*'` tegn.

Ved input er der dog nogle lempelser. Ofte kommer inputdata fra tastaturet eller fra en batchfil, der spiller tastaturets rolle. Her er man ikke glad for, at datatallene skal ligge i et fastlagt format, og det er heller ikke nødvendigt. En måde at få maskinen til at være mere tolerant på er at indsætte tallene afgrænset fra hinanden med kommaer; det er i hvert tilfælde et signal til maskinen om, at her slutter det ene, og begynder det næste, tal. Det kan dog ikke lade sig benytte i tilfældet af for eksempel tegndata. I de fleste tilfælde, hvis det er tal man vil indlæse, er der ingen mening i at bruge andet end frit-format input, altså som brugt i tidligere kapitler, `read *,...` formen. Der er to undtagelser: Det hænder, at man har data fra et instrument, som muligvis pakker tal tæt op ad hinanden, uden ophold imellem. I så fald kan frit-format input ikke bruges. Man er nødt til at give maskinen oplysning om, hvor mange positioner de enkelte tal fylder, hvilket kræver en formatbeskrivelse. Det andet tilfælde er indlæsning af tegnstrenge. Bruger man formatløs input, er man normalt nødt til at markere starten og slutningen af strengene med anførselstegn. Det kan være irriterende, og kan forhindres ved at specificere formatet. Der gives eksempler nedenfor. I nogle af dem bliver tegnet ␣ brugt til at repræsentere et ophold for at gøre det synligt i denne tekst.

I edit item

Den vedrører heltal og har formen `I`w[.m] hvor [.m] betyder, at den del er valgfri. Tegnet w fastlægger **feltbredden**, det vil sige, det antal positioner i alt tallet skal optage ved print, eller optager i inputenheden. Ved output, hvis tallet fylder mindre end w, bliver det skubbet til højre i feltet og den venstre del er ophold. Det hænder, at man vil have førende nuller foran tallet (sommetider brugt af regnskabsmæssige grunde) og i så fald kan man bruge m-delen, som siger, at tallet skal vises med m cifre, lige meget hvor mange cifre det faktisk har. Har det færre end m, bliver der fyldt op med nuller. Det siger sig selv, at m skal være mindre end eller lig med w. Hvis tallet er større end hvad der kan være i w positioner, udskrives der en stribe asterisker, `*`.

Som anført ovenfor, er der nogen fleksibilitet i den eksakte måde, tallene må stå på, selv med formatangivelse. For eksempel, hvis sætningen

```
read '(2i4)', i, j
```

bliver brugt til at indlæse de to variabler, så giver de følgende inputstrenge følgende resultater:

Inputstreng	i	j
␣␣␣1␣␣␣2	1	2
1␣␣␣2	1	2
␣␣␣1␣␣␣␣␣␣␣2	1	0
1,2	1	2
1,␣␣␣2	1	2
1,␣␣␣␣␣␣2	1	0
12345678	1234	5678

Ved brug af komma er det sådan, at kommaet signalerer starten af et nyt inputfelt, og `i4` tæller derfra. Derfor virker det første og andet eksempel med komma, fordi `j` befinder sig inden for `i4`-feltet; hvorimod det i det tredje eksempel er for langt ude efter kommaet. Men der er ingen grund til at indtaste tallet så langt ude.

Det sidste eksempel er et af de få, hvor brug af format er nødvendig. De to tal, som kunne komme fra et instrument, ligger tæt op ad hinanden og det er kun formatspecifikationen, der gør maskinen i stand til at pille dem fra hinanden. Sådanne eksempler (samt det nedenunder, om strenge) er den eneste grund til at bruge formats ved input.

B, O og Z edit items

Disse tre, af formen `B`w, `O`w og `Z`w, specificerer heltalsformat for henholdsvis binære, oktale og hexadecimale tal, dvs. tal baseret på baserne 2, 8 og 16 i stedet for 10. Således bliver tallet 37_{10} (hvor subscriptet giver basen) repræsenteret som `100101` i det binære, `45` i det oktale og `25` i det hexadecimale system. Ligesom ved `I`-format kan der fås førende nuller ved at tillægge .m til formen.

F edit item

Denne edit item er til "floating point" eller `real` tal. Den bruges til at beskrive tal der har decimaler, og har den generelle form `F`w.d, hvor w igen er feltbredden, dvs. antal positioner, hele tallet skal ligge i; og d er antallet af decimaler efter punktummet. Hvis (ved output) d er sat til nul, printes tallet med et punktum, men ingen decimaler derefter.

Ved ouput bliver tallet skubbet til højre i feltet, dvs. den venstre del er fyldt med ophold; det er en måde at skabe plads i outputlinjen. Ved input

(hvis altså man føler sig nødt til at specificere format) leder maskinen efter tallet i feltet af bredden w, men retter sig efter, hvor punktummet er for at fortolke tallet. Igen, et komma efter hvert tal starter et nyt felt.

E og D edit items

Formen er `E`$w.d[Ee]$, hvor w igen er feltbredden for hele tallet. Den er egnet til `real` eller (i tilfældet `D`) `double precision` tal. Det hænder, at output (eller input) bliver så stort eller så lille, i forhold til 1, at `F`-formatet ikke egner sig til at udtrykke det. I så fald skal man bruge den lidt mere besværlige måde, som svarer til notationen som for eksempel 2.998×10^8 eller 1.602×10^{-19}. Der er flere konventioner for at skrive sådanne tal, og de adskiller sig derved, at brøkdelen ligger i forskellige områder. Vælger man `E`- eller `D`-format, ligger brøken mellem 0.1 og 0.999..., med d decimaler efter punktummet, og eksponentdelen indikeret med bogstavet `E` efterfulgt af to eller tre cifre muligvis med fortegn. De ovennævnte eksempler ville således blive printet, hvis formatet var `E15.4`, henholdsvis `0.2998E+09` og `0.1602E-18`. Nogle systemer tillægger et nul foran eksponenten, for at gør den trecifret. Hvis eksponenten nødvendigvis er trecifret (enten positiv eller negativ), får den de tre. Det er også muligt at tvinge et bestemt antal e cifre i eksponenten, ved at vedhænge `Ee`, for eksempel formatet `E15.4E4` giver, for de to tal, henholdsvis `0.2998E+0009` og `0.1602E-0018`.

`D`-format er det samme som for `E`, men det indikerer, at tallet var af dobbelt præcision. Det er mere eller mindre fossilsk i Fortran 90 nu, og det bliver forklaret i kapitlet om `KIND`.

EN edit item

Formen er `EN`$w.d[Ee]$, og den er næsten ligesom `E`-format, med den forskel, at brøken ligger i et større område. Den er også kaldt formatet for ingeniører, som åbenbart kan lide, at eksponenten er delelig med 3 og at brøken er $>$ 1 og derfor ligger mellem 1 og (lidt under) 1000. De to ovenstående taleksempler vil så blive printet henholdsvis som `299.8000E+06` og `160.2000E-21`. Læg mærke til, at antallet af decimaler (`d`) efter punktummet er uændret.

ES edit item

Formen er `ES`$w.d[Ee]$; denne form ligner mest den normale `E`-form, men brøken skal ligge mellem 1.0 og (lidt under) 10. Den kaldes for den naturvidenskabelige form (S for scientific). I dette system bliver vores tal til `2.9980E+08` og `1.6020E-19`, hvilket mange nok ville foretrække.

G edit item

Denne item, kaldt den generelle formatbeskrivelse, kan bruges til at specificere både heltal og `REAL` tal. Den er dog ikke særlig nyttig i forbindelse med heltal. Den har formen `G`$w.d$ eller (analog til `E`-) `G`$w.d$`E`e, men d-delen bliver overset hvis det drejer sig om et heltal, og den fungerer ligesom `I`-item. Den bliver interessant med `REAL` tal. Formats som `F`- og `E`-items egner sig udmærket til at lave tabeller med, men man skal altså beslutte i forvejen, om man vil have den ene eller den anden. Det er imidlertid ofte, at man vil have et tal udskrevet, hvis størrelse man ikke kender i forvejen, og, hvis tallet ligger ikke langt fra 1, vil man foretrække `F`-format, ellers `E`. `G`-format kan begge dele. Kort sagt, hvis tallet kan skrives ud i `F`-format, bliver det skrevet ud i denne form, men her betyder d antallet af alle cifre, ikke, som ved `F`-format, antallet efter decimalpunktet. Hvis dette ikke kan lade sig gøre, bliver `E`-format brugt. I alle tilfælde kommer tallet til at stå tættere på det foregående output. Nogle eksempler vil gøre alt dette klart: I tabellen blev formatet `G12.4` brugt til alle værdier:

Tal	Udskrift
0.0005	0.5000E-03
0.1	0.1000
1.0	1.000
99.0	99.00
100.0	100.0
2000.0	2000.
20000.0	0.2000E+05

A item

Skriver man tegn ud eller læser man dem ind, skal dette format bruges. Formen er `A`w og reglerne er følgende.

Ved input: Lad inputstrengen være n tegn lang, og strengvariablen, som den bliver læst ind i, erklæret med `LEN=`l, hvor $n \leq l$ (men se nedenfor), mens der indlæses med specifikationen `Aw`:

- ingen w: de n inputtegn pakkes i venstre del af strengen;
- ingen w, $n > l$: de første l tages, resten bliver overset;
- $0 < w \leq l$: kun de første w bliver læst ind;
- $w < n$: kun w læst ind;
- $w > l$: virker som om de n tegn i inputtet ligger i den venstre ende af w tegn, og de *sidste* l bliver læst ind. Noget går altså tabt.

Alt dette, som nok er ret abstrakt, kan gøres mere klart med nogle eksempler. I følgende tabel indlæses der en inputstreng i en variabel erklæret som `character(LEN=3)`, dvs. $l = 3$:

Inputstreng	Format-spec	Resultat
`abc`	`A`	`abc`
`ab`	`A`	`ab`
`abc`	`A3`	`abc`
`abc`	`A2`	`ab`
`abc`	`A10`	

Det ses, at det ikke kan betale sig at bruge andet end simpel `A`; bruger man, som i tilfældet `A2`, en w-værdi mindre end n så bliver kun w indlæst. Bruger man en for stor w, går det ikke godt.

Ved output: Her er forholdene lidt mere enkle. Her betyder n den ikke-blanke del af strengen, dvs. længden af den del der kommer foran blankhalen. Bruger man bare formen `A`, bliver hele strengen printet ud i sin erklærede længde; dvs., hvis $n < l$, så bliver de n tegn sat til venstre i de i alt l tegn, og resten fyldt med ophold, som bliver skrevet ud. De er i sig selv usynlige, men hvis der kommer noget efter, kan man se deres tilstedeværelse ved afstanden.

Det kan altså ses, at i både input og output er der gode grunde til altid at bruge den simpleste `A` form, uden nogen w. Læg mærke til, at i alle de ovenstående eksempler bliver der ikke brugt anførselstegn, hvorimod de skal bruges i input-data hvis den simple form, `read *...` bliver anvendt. Den er normalt den mest fornuftige at bruge, men ikke ved indlæsningen af tegnstrenge, medmindre man har lyst til at indtaste anførselstegn hver gang.

L edit item

Dette format bliver brugt med typen `logical`. Formen er `L`w og skriver (ved output) `F` til højre i et felt med bredden w hvis værdien er `.false.`, eller `T` til højre i feltet hvis værdien er `.true.`. Det forekommer yderst sjældent, at man indlæser en logisk værdi.

T edit items

Der er yderligt tre format items, som man *kan* bruge, selvom det er lidt svært at se fornuften i dem. De er alle ment til input. Der er formen `T`n og den betyder, at den næste item refererer til et punkt i den nuværende linje, n tegn fra begyndelsen af linjen, som er defineret som venstre TAB-position. Det er for det meste det første tegn i linjen. Man kan altså, på et hvilket som helst tidspunkt, bestemme, hvor det næste input skal komme fra. Tallet n er en absolut position. Med formen `TL`n kan man bestemme positionen relativt til, hvor indlæsningen nu er, n tegn til venstre. Logisk nok siger formen `TR`n at der rykkes n tegn til højre før næste læsning.

S, SP, SS edit items

Hvis der står `SP` i rækken af nogle format items, betyder det at tallene derefter bliver forsynet med et +-tegn hvis de er positive (det er normalt undertrykt). Står der `SS`, bliver alle plustegn undertrykt derefter. Tegnet `S` sætter igen det der er standard, at systemet bestemmer om der skal bruges plustegn, eller ej.

Andre format-items

Ud over de ovennævnte format items, der specificerer, hvordan variabelværdier skal se ud ved input eller output, er der nogle flere, der hjælper med – mest ved output – at placere dataene og tillader tekst osv. Her er de:

- n`X` sætter n ophold ind;
- "<*tekst*>" eller '<*tekst*>' sætter teksten ind;
- / starter en ny linje;
- : (dvs. kolonet) terminerer outputtet hvis der ikke er mere variabel-output (se senere).

Anførselstegn i en tekst

Hvis man vil have anførselstegn med i en tekst i en formatbeskrivelse (f.eks. `x' =`), gælder der andre regler end de, som er nævnt i kapitel 1 (eksemplet `"Don't"`). Kort sagt skal man sætte to anførselstegn ind i tekststrengen, eksempelvis,

```
print '(" x'' =", f10.3)', xmaerke
```

Sammensætning af format-items

Oftest printes der flere værdier i en enkel `print`, eller læses flere fra en inputlinje. Det sidste er normalt nemmest ved brug af `read *...`, så der fokuseres her på output. Ud over variabelværdier vil man ofte også have tekst blandet sammen med disse. Det hele gøres ved at kæde et antal format-items sammen i en formatspecifikation, separeret med kommaer. Den mest anskuelige måde at gøre dette klart på er at give et eksempel. Variablen `kroner` (en `REAL`) bliver printet ud her, i to forskellige former, sammen med tekst og et linjeskift:

```
print '(" Antal kr : ", f10.2/ &
      & " (Afrundet:)", i7)',  &
        kroner, NINT(kroner)
```

og resultatet kan for eksempel være:

```
Antal kr :      457.25
(Afrundet:)     457
```

Det kan være god *programmeringsstil* at bruge fortsættelseslinjer, for at programmet afspejler outputtet. Bemærk, at tegnet, &, brugt til at indikere fortsættelsen af den første linje, bruges igen lige før formatspecifikationen fortsættes der. Det er fordi hele formatspecifikationen faktisk er en lang tegnstreng, og hvis man udelader &-tegnet på den anden linje (hvilket man godt må), bliver hele strengen unødvendigt lang; visse systemer giver en advarsel om dette ved oversættelse, hvilket kan være irriterende. Bemærk desuden, at linjeskift-tegnet, `/`, følger direkte efter `f8.2`, uden komma ind imellem. Det komma er ikke nødvendigt med `/`. Hvis vi ønsker at indrykke outputtet på linjen, kan vi tilføje nogle ophold, ved brug af `X`-item:

```
print '(20X, "Antal kr : ", f10.2/ &
      & 20X, "(Afrundet:)", i7)',  &
        kroner, NINT(kroner)
```

hvilket giver

```
                    Antal kr :      457.25
                    (Afrundet:)     457
```

Repeat-tal og kolon-item

Kolon-format item skal forklares nærmere. Først, den såkaldte **repeat**-facilitet: hvis der er flere inputs eller outputs efter hinanden, der specificeres med samme formatbeskrivelse, kan man nøjes med en enkelt, og repeat-tal foran. Det sås allerede i det sidste eksempel, hvor der står `20X`, altså 20 gange `X` eller, som det ses i starten af dette kapitel, `4i4`. Man kan også gange op hele grupper af format-items ved at indramme dem med parenteser og et repeat-tal foran. Hvis man ved programmeringstidspunktet ikke kan forudse hvor mange gange noget skal gentages, kan man ofte bruge et tal, der er stort nok til alle tilfælde. I det følgende eksempel skal der printes en række heltal ud med komma efter hvert, men helst bortset fra den sidste. Første forsøg kan se sådan ud:

```
integer :: tal(10), n
tal = (/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 /)
n = 6
print '(" Tallene er:", 10(i3,","))', &
          tal(1:n)
```

og det producerer outputtet

```
Tallene er:  1,  2,  3,  4,  5,  6,
```

Det sidste komma er ikke pænt, og det kan forhindres ved en kolon-item:

```
print '(" Tallene er:", 10(i3, :, ","))', &
          tal(1:n)
```

hvilket giver det mere tilfredsstillende

```
Tallene er:  1,  2,  3,  4,  5,  6
```

I det første program bliver hele det indrammede format gentaget, indtil der ikke er flere værdier at printe; mens der i den anden form bliver set frem, om der kommer mere, før teksten efter tallet bliver printet. Hvis der ikke er flere, holder printeren der.

Regler vedrørende formatspecifikationer

Der inkluderes her frit format (`*`-format). Selvom programmøren ikke specificerer formatet i dette tilfælde, er det alligevel en slags format, bestemt af maskinen (ved output) eller af hvordan inputdata ser ud.

1. De format items, der er i rækken af items i formatstrengen, bliver brugt en for en, i takt med elementerne i listen; hvert item/element- par skal være typekompatibel (`I` med heltal, `F` med `REAL` tal, osv.).

2. Hvis der er et antal elementer i en inputliste og frit format er brugt, så leder maskinen efter alle disse elementer, selv hvis det indebærer, at der skal læses fra flere linjer for at finde alle elementer i listen.

3. Hvis et format er specificeret for indlæsning, og der er færre data på den pågældende linje, end antallet af elementer i listen, så bliver de manglende sat til nul (men se under `PAD`, afsnit 5.2).

4. Hvis et format er specificeret for indlæsning, og der er flere data på den pågældende linje, end antallet af elementer i listen, så bliver lige så mange elementer læst ind, som er i listen, og resten af data bliver overset; derefter er næste linje klar til læsning.

5. Antallet af formatspecifikationer må godt være flere end antallet af listeelementerne; de bliver "brugt" en ad gangen indtil alle elementer i listen er læst ind eller skrevet ud.

6. Hvis antallet af formatspecifikationer er mindre end antallet af elementerne i listen, bliver format items "genbrugt" fra starten (eller fra den nærmeste venstre parentes til

venstre). Ved genstart tages det næste input fra en ny linje, eller output udskrives på en ny linje. Det er dog vigtigt, at formatitems er typekompatible med de variabler, data skal læses ind i eller skrives ud fra. En undtagelse her er, at hvis format item er en simpel `a`, sker der ikke et linjeskift.

7. Hver `print` eller `read` begynder en ny linje (record).

Den første regel er uproblematisk. Man skal sørge for, at de rigtige specifikationer bliver brugt. Det plejer kun at give besvær når en formatrække bliver genstartet, hvis den indeholder items for forskellige typer. Mest skyldes det en programmeringsfejl.

Reglen 3 forudsætter faktisk, at `PAD`-er specificeret som `YES` (se nedenunder for en forklaring); og det er den per default. Mekanismen er, at hvis en inputlinje er for kort, bliver den forlænget med ophold, og input læses fra den del; det giver nul.

Reglen 5 kan bruges, når der skal udskrives et i forevejen ukendt antal elementer (af samme type), med et repeat-tal stort nok til at dække alle tilfælde. Det vises nedenunder i den forbedrede version af MATOUT.

Reglen 6 kan gøres forståelig med et eksempel. Følgende programstump,

```
integer :: n(10)
N = (/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 /)
print '(4i4)', N
read '(4i4)', N
print '(20i4)', N
```

indlæser disse data:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
11, 12, 13, 14, 15, 16, 17, 18, 19, 20
21, 22, 23, 24, 25, 26, 27, 28, 29, 30
```

Resultatet (output) er:

```
 1   2   3   4
 5   6   7   8
 9  10
 1   2   3   4  11  12  13  14  21  22
```

Det ses, at `print`-sætningen skriver tallene 1-10 ud på tre linjer, fire ad gangen, til sidst kun to. Indlæsningen foregår ligeledes fire tal fra hver linje, og linjeskift derefter, kun to er læst fra den sidste. Den sidste linje i outputtet viser de 10 værdier, som variablen består af, selvom `repeat`-tallet er 20 (reglen 5). I dette eksempel var genstartpunktet starten af hele formatstrengen. Hvis der er en gruppe i parenteser, bliver den genstartspunktet. Det illustreres bedst ved eksemplet

```
print '(i4, 2(f10.3, i5))', &
       77, 10.1, 2, 4.7, 1, 12.4, 5, 5.2, 6
```

som giver outputtet

```
 77    10.100    2     4.700    1
   12.400    5     5.200    6
```

Tallet `12.4` bliver formateret med `f10.3` efter linjeskift, og ikke med `i4` – hvilket heller ikke ville være kompatibelt. Det er klart, at der er fælder i denne egenskab af Fortran, og man skal være varsom hvis man vil benytte sig af den.

Undtagelsen i reglen 6 (brug af `a`-item) illustreres i følgende afsnit.

MATOUT igen

I kapitel 4 blev der vist et underprogram, `MATOUT`, til at printe en matrix ud med. Outputtet virkede lidt klodset med de unødvendige nuller, som skyldtes det fri format. Nu gentages eksemplet her, med formats. Hovedprogrammet er det samme, og her vises kun selve underprogrammet:

```
subroutine MATOUT (streng,arr,Nmax,Mmax,N,M)
! Matricen ARR udskrives paa standardenheden
! (skaerm) ved brug af formateret print,
! med en tekst i foerste linje.
  implicit none
  character(LEN=*) :: streng
  integer          :: Nmax, Mmax, N, M
  real             :: arr(Nmax,Mmax)
  integer          :: L, i
  L = LEN_TRIM (streng)
  print '(1X, a)', streng(1:L)    ! A-format
  do i = 1, N
    print '(10f8.2)', arr(i,1:M)  ! F-format
  enddo
end subroutine MATOUT
```

Det nye output ser sådan ud:

```
 En 3*4 matrix
   11.00   12.00   13.00   14.00
   21.00   22.00   23.00   24.00
   31.00   32.00   33.00   34.00
```

hvilket er langt bedre. Der er en ting der stadig skal forklares: hvorfor `1X` blev brugt foran `A`-format. Forklaringen er i næste afsnit. Men det simple `a` kunne få en til at tro, at hvert tegn bliver sat på en ny linje; undtagelsen betyder dog, at det ikke er tilfældet, selv med `1x` foran; hele strengen kommer ud på samme linje (heldigvis).

Fortran printer control

Per tradition har Fortran en mærkelig facilitet, som af og til irriterer: når noget printes ud, bliver

det første tegn ikke printet, men taget som kontroltegn, der skal sige noget om "papir"- fremrykkelse, eller *carriage control*, som det hedder på engelsk. Der er nogle standard tegn med bestemte effekter. Altså, hvis man allerførst i en outputlinje har et vist tegn, sker der bestemte ting. De er:

Tegn	Aktion
␣ (ophold)	default, ny linje
`0`	spring en linje over
`1`	ny side
`+`	blive på samme linje

Printer man på en skærm, er det systemafhængigt, om systemet fortolker det første tegn som carriage-control, eller bare printer tegnet som det er. I det første tilfælde forsvinder tegnet. I al fald er det en god vane ikke at bruge den første position ved output. En måde at sikre dette på er altid at starte en formatstreng med `1X`. Det får det første tegn til at være et ophold, hvilket resulterer i det, man normalt vil – en ny linje. Andre tegn end dem i tabellen kan give mærkelige, systemafhængige resultater. Tegnet + brugtes engang (af nogle) til at lægge til linjer, som var skrevet på i forvejen; men det har altid kunnet lade sig gøre langt mere elegant på anden måde. Det er et fossil i sproget. I afsnit 5.3 vises der en bedre måde at opnå undertrykkelse af linjeskift på (nonadvancing i/o). Printes der en tom linje, virker den som et enkelt ophold; dvs. en ny, tom linje. Det sker, at man vil netop dette. Det kan lade sig gøre med sætninger som

```
print *
write (17,*)
```

(hvor `lun` 17 bare er et eksempel).

5.2 Filer

Input og output er overførsel af data mellem maskinen og ydre enheder. De kan være terminalens tastatur (for input til maskinen) eller skærmen (for output). Men data kan også overføres til og fra såkaldte **filer**, som nu i dag mest ligger på en disk (andre muligheder er, eller har været, magnetbånd, hullet papirstrimmel, hulkort m.m.). Data på sådanne enheder er samlet i filer, og hver fil har et **filnavn**, hvis form er karakteristisk for de forskellige operativsystemer. Det er i al fald filnavnet, der identificerer (peger på) en bestemt fil. Filer er delt op i **records**; når der er tale om filer bestående af tekst (altså altsammen tegn), er records ensbetydende med linjer. Når der overføres data, overføres der altid en record ad gangen, ikke mindre. En record er mærket ved enden på en ikke nærmere bestemt måde med et **end of record** mærke, eller **eor**. Hele filen er desuden forsynet, ved enden af filen, med et **end of file** mærke, eller **eof**. Disse begreber kommer ind i billedet når sammenkoblingen af filer med I/O-sætninger bliver behandlet.

En fils records kan enten alle have samme længde ("fixed record length") eller forskellige. Desuden skelner man mellem filer der skal gennemgås sekventielt ("sequential access") – altså fra starten frem til enden – eller med hvad der hedder "random access", hvor access kan hoppe til et udvalgt sted. Den slags fil, ofte brugt i administrative programmer, bliver oprettet med en fast recordlængde (logisk nok), og hele filen bliver faktisk skabt som et tomt skelet, som programmet så kan fylde lidt efter lidt. Normalt er filer skabt med sekventiel access, og så kan records have (og har oftest) den længde, de får, når de skrives ud, en efter den anden.

I afsnit 5.3 bliver de generelle I/O-sætninger beskrevet (`read`, `print`, `write`). Disse sætninger har ingen mulighed for at specificere filnavne; de refererer til numre, kaldt (på engelsk) **logical unit numbers**, eller **lun**. Det er altså nødvendigt at skabe en forbindelse mellem en fil og en **lun**, hvilket gøres med `OPEN`-sætningen; samtidig meddeles til systemet en del om filens egenskaber og hvad man vil tillade systemet at gøre med filen, mm.

Filer kan sammenlignes med mapper. Hvis man vil have information (data) ud af en mappe, eller lægge nye data ind i den, skal man åbne (den rigtige) mappe først, derefter gøre det man vil, og til sidst lukke den igen. Man kan også sikre sig, at en bestemt mappe overhovedet findes, og man kan beslutte hvad man ville gøre, hvis den mappe, man vil åbne, ikke er der, eller er tom, osv. Det kan være, at man har lov til kun at læse i en mappe, eller også skrive i den. Alle disse aktiviteter findes i forbindelse med datafiler i Fortran 90.

I de følgende underafsnit bliver de forskellige filsætninger beskrevet. De er: `OPEN`, `CLOSE`, `REWIND`, `BACKSPACE`, `ENDFILE`, `INQUIRE`, og I/O-sætningerne `read`, `print` og `write`.

OPEN-sætning

`open`-sætningen har den generelle form

```
OPEN (<list>)
```

hvor <*list*> er en række elementer, der informerer systemet om filen. De har hver den formelle form

```
[<keyword> =] <spec>
```

og der er en stribe af disse. Som indramningen med [...] indikerer, kan man dog undvære

<*keyword*>-delen i mange tilfælde, og mange specs kan udelades, overlades til systemets default. En komplet, men kortfattet liste af keywords, med antydninger af de tilhørende specs (de er alle detaljeret senere) er:

```
UNIT =  tal
FILE =  filnavn
STATUS =  status spec
IOSTAT =  integer variabel
ERR =  label
FORM =  formbeskrivelse
ACCESS =  access spec
RECL =  recordlængde
BLANK =  blank spec
POSITION =  position
ACTION =  action spec
DELIM =  delim spec
PAD =  padding spec
```

Disse enkelte specifikationer bliver nu alle gennemgået. Det bliver også nævnt, hvad default er, hvis der er en default. Ordet betyder det, der bliver sat af systemet, hvis man ikke selv specificerer noget. Defaults gør en programmørs liv nemmere.

UNIT: Der skal angives en heltalsværdi mellem 1 og 99; enten som konstant, variabel eller et udtryk. Det er det mest centrale led i forbindelsen af en fil med det nummer, der identificerer filen fra da af. Tallet kaldes også **lun** (for logical unit number). Hvis denne spec er den første i rækken (som den i reglen er) kan man udelade `UNIT=` og bare indsætte tallet (værdien). Denne spec er nødvendig; der er ingen default.

Der er ofte nogle standard-units; ofte er det underforstået i systemer, at unit 5 er både tastaturet og skærmen, og nogle systemer gav unit 6 til printeren. Disse defaults er stort set fossilske nu. Man giver normalt ikke et program lov til direkte at printe til en printer, mens tastatur og skærm kan klares uden noget `lun` (og uden en `OPEN` – de er åbne i forvejen).

FILE er det, der specificerer filens navn. Det skal være en tegnstreng, og det kan være enten en konstant, som `'instrum.dat'`, eller en `character`-variabel som indeholder navnet. Det sidste er også det mest fleksible, så har programmet mulighed for at køre med flere end den ene fil. Det er på sin plads at give et eksempel her, der gør forskellen tydelig. Antag, at et program skal læse data fra en datafil. Vi beslutter, at `lun` 1 skal bruges. Den relevante programkode kan være:

```
OPEN (1, file='instrum.dat',...)
```

som knytter `lun` 1 til den ene fil hver gang programmet kører. Vil man læse fra flere filer, kan man gøre som:

```
character(LEN=20) :: filnavn
...
print '(" Indtast filnavn:")'
read '(a)', filnavn
OPEN (1, file=filnavn,...)
```

Strengen `filnavn` blev erklæret med 20 tegns længde, hvilket burde være nok. Læser man en ind der er kortere end de 20, får den en hale med blanktegn, som `OPEN` ser bort fra. Der er ingen defaults.

STATUS: Dette keyword siger, hvordan filen skal benyttes. Der er fem mulige specs, og de er alle tegnstrenge:

- `'old'` – filen eksisterer i forvejen; det betyder mest at den skal læses fra, men ikke altid.
- `'new'` – filen eksisterer ikke endnu, og bliver skabt her. Det betyder nødvendigvis at den skal skrives i.
- `'scratch'` – filen skal oprettes, bruges under programkørsel og forsvinde igen bagefter (efter `CLOSE`).
- `'unknown'` – det er uspecificeret, afhængigt af systemet. Normalt betyder det det samme som `'old'` hvis filen eksisterer, eller `'new'` hvis ikke. Men eftersom det er uvist, er `'unknown'` ikke at anbefale.
- `'replace'` – filen bliver tømt for indhold hvis den eksisterer i forvejen, eller oprettet hvis ikke. Dette kan være farligt.

Hvis denne spec er udeladt, er defaultreaktionen, at systemet sætter `'unknown'`. Det er derfor bedst at skrive noget her.

IOSTAT: Der skal være navnet på en integervariabel efter `iostat =`. Denne variabel får en værdi som er afhængig af hvor godt det gik med åbningen af filen. Der er flere muligheder for fejl. Hvis alt går godt, og filen blev koblet sammen med den angivne `lun`, er variablen sat til nul. Er der et problem (filen findes ikke eller er allerede åben, m.m.), har variablen en positiv værdi; den eksakte værdi afhænger af systemet. Ved brug af `IOSTAT` kan man sikre sig, at en filåbning er i orden, og at det er værd at fortsætte. Her er et eksempel:

```
OPEN (17,file='tal.dat',iostat=ios,...)
if (ios /= 0) &
  STOP " Noget galt med filen tal.dat."
```

STOP-sætningen, som aborterer hele programmet, bliver sprunget over, hvis ios er nul. IOSTAT dukker op igen i forbindelse med i/o-sætninger, i afsnit 5.3. Udelader man denne spec (som man tit gør), og hvis der så er en fejl, giver systemet en besked, og programmet standser. Det kan opfattes som det samme som en STOP; men den besked man får, er ikke altid så forståelig som den, man selv programmerer.

ERR= er en fossilsk form, der overlever fra ældre udgaver af Fortran; dens funktion er nu overtaget af IOSTAT. Argumentet til ERR er en label (et tal), som programmet skal hoppe til, hvis der opstod en fejl ved filåbningen. I dag bruger man meget sjældent labels, og denne form er ikke anbefalet.

FORM: Der er kun to muligheder: 'formatted' eller 'unformatted'. Det første er det normale, og betyder, at filen består af tekst, altså stort set af tegn man kan indtaste. Se nedenfor for den anden type filer, i afsnit 5.3. Defaultreaktionen her er, at filen opfattes som en tekstfil ('formatted').

ACCESS tillader valget mellem 'sequential' (default) og 'direct'. De to er allerede forklarede ovenfor.

RECL: Recordlængden skal angives for filer med 'direct' access, for at systemet kan oprette det ovenfor beskrevne tomme filskelet. Dens længde skal specificeres i enheder bestemt af systemet. De fleste ville nok bruge enheden *byte*, dvs. en 8-bits længde, det samme som et tegn normalt fylder i maskinen. For 'sequential'-access filer, er det normalt ikke nødvendigt at specificere 'recl', medmindre man har grund til at fastlægge længden. Maskinen (systemet) har som regel sin foretrukne defaultværdi, og den er oftest så liberal at den ikke giver problemer. Problemer skyldes mest programmeringsfejl. De opstår når man skriver, i en outputfil, en record (linje) som er for lang. Men det er normalt svært at gøre, når den tilladte længde er, for eksempel, 2048 bytes lang. Normalt kan man altså glemme denne spec og overlade den til dens default.

BLANK: Når der indlæses fra en linje ved formateret (tekst-) input, og der er blanktegn, kan man her vælge, hvordan sådanne blanktegn bliver fortolket. Specificerer man 'null' bliver de sprunget over, mens de bliver fortolket som nuller, hvis man specificerer 'zero'. Et eksempel kan belyse forskellen:

```
program BLANK_TEST
  integer :: n, m
  OPEN (1,file='null.dat',status='old',&
            blank='null',action='read')
  OPEN (2,file='zero.dat',status='old',&
            blank='zero',action='read')
  read (1,'(i4)'), n
  read (2,'(i4)'), m
  print *, " n (blank='null') =", n
  print *, " m (blank='zero') =", m
end program BLANK_TEST
```

og begge datafiler indeholder det samme, som består af en enkelt linje med kun et tegn på, i første position:

```
4
```

Resultatet af programkørsel er:

```
  n (blank='null') =           4
  m (blank='zero') =        4000
```

I begge tilfælde, eftersom der kun er et tegn, men fire er efterlyst, bliver linjen udvidet med ekstra tre tegn. I filen 'null.dat' bliver disse sprunget over, og resultatet er værdien 4. Men i filen 'zero.dat' bliver udvidelsen fortolket som nuller. Det er nok sjældent, at optionen 'zero' er noget man vil vælge.

POSITION: Når man læser fra eller skriver i en fil, er der altid en "pegepind" der peger på det sted (positionen) man befinder sig i. Normalt, når en fil åbnes, står pegepinden ved starten af filen. For en fil med sekventiel gennemgang (access) flyttes positionen til starten af næste linje efter en læsning eller skrivning. Startpositionen kan imidlertid sættes af programmøren i OPEN-sætningen med denne spec. Valgene er: 'rewind', 'append' eller 'asis'. Det første navn stammer fra den tid, hvor filerne lå på magnetbånd, som skulle spoles tilbage (engelsk: rewind), dvs. positionen skal være starten af filen. Det kan være, at en fil findes i forvejen men åbnes for WRITE; man vil for eksempel lægge til filen. Så er 'append' relevant, som beder om at positionen placeres efter sidste record (lige før end-of-file mærket). Hvis man glemmer at specificere positionen som 'append' bliver den sat til starten, og man overskriver det gamle indhold. 'asis' er et lidt underligt valg. Det er tilladt i Fortran, at genåbne en fil som er åben i forvejen, sålænge man nævner samme lun. Når man så specificerer 'asis', så bliver positionen som den var. Default her er 'asis'.

ACTION har de tre muligheder 'read', 'write' og 'readwrite', og de siger programmet, hvad man vil med den pågældende fil. Det hænder, at man vil læse fra en fil som en anden bruger stiller til rådighed, men det er sjældent, at man samtidig får lov til andet end at læse fra sådan en fil. Selvom programmet er skrevet med kun read-sætninger med hensyn til den fil, ville OPEN ikke være en succes, medmindre ACTION='read' er specificeret, fordi det er her at programmet 'lover' den anden bruger hvad det skal med filen. Indeholder programmet så alligevel sætninger med write, så lykkes oversættelsen givetvis ikke – oversætteren checker altså, om i/o stemmer sammen med OPEN. Vælges der 'write', så er kun dette tilladt. Vil man både læse fra og skrive i en fil, vælger man 'readwrite', hvilket i de fleste systemer også er default. Det er en misforståelse, hvis man tror, at når man åbner en fil med STATUS='old', betyder det at man kun kan læse fra den. Alle disse ACTIONs skal naturligvis passe sammen med de egenskaber, systemet har givet en bestemt fil. Det er jo uafhængigt af et Fortranprogram, om en fil er sat med lov til at skrive i den eller ej, eller hvem der har adgang til filen.

DELIM: Når en tegnstreng skrives ud, kan man her vælge, om den skal indrammes med enkelte anførselstegn ('apostrophe'), dobbelte ('quote'), eller ingen ('none'). Defaultværdien er den sidste. Alt dette gælder kun output; ved input er denne spec overset.

PAD: Som nævnt i forbindelse med formateret read, hvis en record, der læses fra, er kortere end formatet forlanger, bliver den normalt udvidet med blanktegn; det er defaultaktionen, eller hvis PAD='yes' bliver sat. Sætter man i stedet PAD='no', så sker udvidelsen ikke – men så må den situation ikke opstå, alle records skal i så fald være tilstrækkelig lange; ellers opstår der en fejl.

Defaults: Det ovenstående kan virke lidt skræmmende, hvis man mener at skulle specificere alt; heldigvis har mange specs fornuftige defaults i de fleste systemer, så det, en programmør er tvunget til at nævne, reduceres til ganske lidt – normalt til kun tre specs: logical unit number (lun) skal selvsagt altid være der, og filens navn (men selv det bliver sat til et standardnavn af nogle systemer hvis man udelader det); og så er det en god ide at angive STATUS, og i visse tilfælde ACTION. Det er sjældent, at man har brug for mere. For et eksempel, se det ovenstående lille program, BLANKS_TEST under afsnittet om BLANK =.

CLOSE

Hvis man åbner en fil, så er det logisk, at man lukker den igen efter man er færdig med den. Gør man det, frigør man samtidig denne fil samt det lun, den var knyttet til, og begge kan derefter bruges igen i samme program. Der er dog ikke mangel på luns (der er 99 af dem), så det er en smule akademisk. CLOSE har, lige som OPEN, flere items der kan specificeres, og ligeledes defaults der gør det langt nemmere end det umiddelbart virker. Keywords er:

- UNIT =
- IOSTAT =
- ERR =
- STATUS =

Den første er, logisk nok, påkrævet (det er jo klart, at det skal siges, hvilken fil der skal lukkes), men, som ved OPEN, kan selve keyword udelades. IOSTAT og ERR er lige som beskrevet før, og det er kun STATUS der behøver nærmere forklaring. Her er der kun to muligheder: 'keep' eller 'delete'. De skal dog stemme overens med det der blev specificeret under STATUS i OPEN-sætningen for den fil. Hvis man har sagt 'scratch' der, kan man ikke sige 'keep' her. Det er imidlertid sådan, at selvom man har åbnet filen med 'new' (hvilket oftest betyder, at man vil oprette en ny fil og også beholde den), bliver den slettet hvis man her siger 'delete'.

I praksis har man allerede i OPEN besluttet filens endelige skæbne, og man kan så nøjes med så lidt som lun, som igen kan forkortes til bare lun-tallet selv uden keyword, for eksempel

```
CLOSE (17)
```

som lukker den fil, der var knyttet til lun 17. Desuden er det i de fleste systemer sådan, at filer, der ikke udtrykkeligt lukkes i et program, bliver lukket når programmet er kørt; systemet lukker alle filer selv, som regel.

REWIND

Hvis man vil genstarte en fil (måske læse fra den efter at have skrevet i den), kan man "spole" den tilbage. Det vil sige, placere positionen til filens begyndelse. Det gøres med REWIND, som for det meste har den simple form, for eksempel

```
REWIND 17
```

Man kan også, hvis man vil, bruge keywords og parentes, hvor der er tre mulige keywords: UNIT =, IOSTAT = og ERR =.

BACKSPACE
Det blev nævnt i et tidligere afsnit, at der er en slags pegepind associeret med en fil, når der læses fra eller skrives i den. Normalt (undtagen en fil med `'direct'` access) flyttes positionen af denne imaginære pind, efter aktionen er færdig med en record, til den næste. Man kan flytte den tilbage til den forgående record – den, man lige har læst – ved en `BACKSPACE`-sætning. Således kan man læse en record flere gange. Det er ikke tilladt at backspace fra en record skrevet i frit-format eller ved brug af `NAMELIST` (se senere).

ENDFILE
I starten af dette hovedafsnit blev det nævnt, at hver fil er mærket ved enden, dvs. efter den sidste record, med et `eof`-mærke (**end of file**). Dette mærke bliver brugt af, for eksempel, `IOSTAT` i en `read`-sætning (se næste hovedafsnit). Det bliver normalt hæftet ved filen af systemet, efter programmet er færdigt med at køre. Men man kan udtrykkeligt sætte det selv. Formen er simpel:

```
ENDFILE 1
```

hvor altså den fil, der er knyttet til `lun` 1 forsynes med et `eof`. Det gælder naturligvis kun filer, programmet har skrevet i, altså dem, der har `ACCESS='write'` (eller `'readwrite'`).

INQUIRE
Denne sætning bliver erfaringsmæssigt meget sjældent brugt, muligvis fordi den er meget indviklet. Men i sammenhæng med store programmer med mange filer kan den sikkert bidrage til gnidningsfri kørsel. Den beskreves her noget kortfattet. Kort sagt, kan et program stille spørgsmål til systemet om en fil, som kan specificeres enten ved `lun` eller ved navn; hver `INQUIRY`-sætning har lov til at spørge om en ting ad gangen. De to former er altså

```
INQUIRE ([unit =]<unit-nummer>, &
         < inquire-spec>, <svar-del>)
INQUIRE (file=<filnavn>,     &
         <inquire-spec>, <svar-del>)
```

hvor de firkantede parenteser ([...]) betyder, at man kan udelade den del (men ikke `file =`). Eftersom der stilles spørgsmål, kommer der et svar, og det skal placeres et eller andet sted; det er i hvert tilfælde en variabel man selv nævner. Nogle af disse er `integer`, nogle er af typen `logical`, nogle (de fleste) er tegnstrenge. En undtagelse er inquiry-spec `ERR`, som har en label som argument. De mulige inquiry-specs og deres argumenter/resultater (<*svar-del*>) er:

- `IOSTAT =` <integervariabel> (tallene som før)
- `ERR =` <label> (hopper dertil hvis der er en fejl)
- `EXIST =` <logical variabel> (`.true.` hvis filen eksisterer, ellers `.false.`)
- `OPENED =` <logical variabel> (`.true.` hvis filen er åben, ellers `.false.`)
- `NUMBER =` <integervariabel> (det `lun`, filen er knyttet til. Hvis der ikke er et `lun`, er resultatet -1).
- `NAMED =` <logical variabel> (`.true.` hvis filen har et navn, ellers `.false.`)
- `NAME =` <streng> (filens navn)
- `ACCESS =` <streng> (filens access som i `OPEN`)
- `SEQUENTIAL =` <streng> (svar er enten 'YES' eller 'NO')
- `DIRECT =` <streng> (svar er enten 'YES' eller 'NO')
- `FORM =` <streng> ('FORMATTED', 'UNFORMATTED' eller 'UNDEFINED')
- `UNFORMATTED =` <streng> (svar er enten 'YES' eller 'NO')
- `RECL =` <integervariabel> (recordlængde)
- `NEXTREC =` <integervariabel> (position (i antal records) af næste record; kun for `'direct'` access)
- `BLANK =` <streng> ('NULL', 'ZERO' eller 'UNDEFINED')
- `POSITION =` <streng> ('REWIND', 'APPEND' eller 'UNDEFINED')
- `ACTION =` <streng> ('READ', 'WRITE' 'READWRITE' eller 'UNDEFINED')
- `READ =` <streng> ('YES', 'NO' eller 'UNKNOWN')
- `WRITE =` <streng> ('YES', 'NO' eller 'UNKNOWN')
- `READWRITE =` <streng> ('YES', 'NO' eller 'UNKNOWN')
- `DELIM =` <streng> ('APOSTROPHE', 'QUOTE' eller 'NONE')
- `PAD =` <streng> ('YES' eller 'NO')

Et par eksempler:

```
logical          :: file_open
integer          :: lun, reclen
character(LEN=9) :: actionstreng
...
INQUIRE (1, OPENED=file_open)
INQUIRE (1, RECL=reclen)
INQUIRE (1, ACTION=actionstreng)
INQUIRE (file='num.dat', NUMBER=lun)
```

5.3 I/O sætninger

Der findes tre I/O-sætninger: `READ`, `PRINT` og `WRITE`. De to første er blevet brugt før dette afsnit, men i en begrænset form, som nu skal udvides til de mere generelle former. Der er dog ikke så meget i forbindelse med `print`, som inkluderes her for ordens skyld.

PRINT
Denne sætning har den simple generelle form

`print` <*format-spec*>`,` <*liste*>

hvor *format-spec* er som beskrevet (og kan være `'*'` som allerede beskrevet), og *liste* er listen af de værdier, der skal skrives. Der skrives altid på skærmen eller det der erstatter skærmen når programmet kører i batch-mode (det kan være en log-fil eller en resultatfil, afhængigt af systemet og ens eget valg). Denne sætning kan altså ikke bruges til at skrive i en fil.

READ og WRITE
Disse to sætninger har en del fælles, og optionerne beskrives samlet her. `READ` kan læse fra tastaturet men også fra en fil, og `WRITE` kan skrive til skærmen (som `PRINT`) og til en fil. Vil man bare læse fra tastaturet (dvs. indtaste data), har `READ` den simple form

`read` <*format-spec*>`,` <*liste*>

hvilket minder meget om `print`. Som med `print` kan <*format-spec*> enten være en af de mange formatspecifikationer, eller bare `'*'`, som beskrevet før. Den mere generelle form for `READ/WRITE` med filer eller tastatur/skærm er:

`read` (<*kontrolspecs*>) <*liste*>
`write` (<*kontrolspecs*>) <*liste*>

med følgende muligheder for *kontrolspecs*. Nogle af disse er obligatoriske, andre kan overlades til defaults. Nogle gælder kun for `READ`:

- [UNIT =] <`lun`-værdi>
- [FMT =] <formatspecs>
- [NML =] <namelist navn>
- IOSTAT = <integer variabel>
- ADVANCE = 'YES' eller 'NO'
- REC = <record-nummer>
- SIZE = <integer variabel> (kun for `READ`)
- ERR = <label>
- END = <label> (kun for `READ`)
- EOR = <label> (kun for `READ`)

De første tre er anført med [...], hvilket betyder, at man kan specificere disse uden deres keywords, så længe man holder sig til den samme rækkefølge som i listen. Alle andre specs skal nævnes med deres keywords.

UNIT betyder det `lun`-tal, filen er knyttet til. Hvis man vil læse fra tastaturet eller skrive til skærmen, skriver man `*` her. Ellers kan værdien angives som en konstant, integer-variabel eller et udtryk der evalueres som heltal.

FMT er formatspecifikationen, som er beskrevet ovenfor. Frit format er tilladt, og er indikeret med `*`.

NML er en anden måde at præsentere data på. Normalt ligger de data, der skal indlæses, som værdier, og der er normalt lige så mange (når alt går godt) som programmet indlæser. Læser et program, for eksempel, tre heltal, *i*, *j* og *k* ind fra en fil med `lun` 17, kunne man have en sætning som

```
read (17,*) i, j, k
```

hvilket betyder, at tallene læses fra `lun` 17 med frit format. Tallene står i filen, måske som:

```
1, 2, 3
```

Bruger man derimod `NML`, er tingene lidt anderledes. `NML` erstatter `FMT =` (de kan ikke begge bruges). Først skal man have oprettet en liste af variabelnavne der skal kunne indlæses med `NML`; det gøres med en `NAMELIST`-sætning:

`NAMELIST /` <*nml navn*> `/,` <*variabelliste*>

Denne sætning skal være lige efter erklæringerne, eller i en module brugt af enheden. Der gives et eksempel om lidt. Dataene i filen står på en speciel måde:

`&`<*nml navn*> `var1=`<*værdi*>`, var2=`<*værdi*>`, ...`

og `read`-sætningen skal også indikere nml-navnet. Et simpelt eksempel skal gøre det hele klart. Følgende program indlæser, tre gange, de tre variabler *i*, *j* og *k*; første gang normalt, med frit-format, derefter to gange med `NML`:

```
program NML_TEST
 namelist / ijk / i, j, k
 OPEN (unit=1, file='nml.dat', status='old')
 read (1,*) i, j, k
 print '(" read, i, j, k    =", 3i6)', i,j,k
 read (1,ijk)
 print '(" NML read, i, j, k ", 3i6)', i,j,k
 read (1,ijk)
 print '(" NML read, kun i, k", 3i6)', i,j,k
end
```

Datafilen `nml.dat` ser sådan ud:

```
1, 2, 3
&ijk i = 4, j = 5, k = 6 /
&ijk i = 7, k = 9 /
```

Linjerne to og tre starter med `'&ijk'`, hvilket indikerer, at de er i den `NML` stil der har navnet `ijk`. Bemærk, at alle tre variabler har fået værdier i den første af disse to linjer, og at formen altid er <varnavn> = <værdi>. I den anden af de to linjer mangler der *j*; det betyder, at kun *i* og *k* får nye værdier i denne omgang. For at systemet kan finde ud af, hvor sådan en datalinje er slut, skal de alle afsluttes med skråstregen `/`. Resultatet af en kørsel af programmet er:

```
 read, i, j, k    =     1     2     3
 NML read, i, j, k      4     5     6
 NML read, kun i, k     7     5     9
```

Denne facilitet kan virke lidt underlig, men kan have sine anvendelser. Eksempelvis vil man beregne en funktion af mange parametre ved at variere nogle af dem ad gangen. Ved `NML` kan man så skrive programmet sådan, at det indlæser dataene igen og igen, og i selve datafilen bestemme, hvilke data skal erstattes med nye, og nævne udelukkende dem. Uden `NML` ville man være tvunget til enten at indlæse alle parametre hver gang, eller skrive programmet på en specifik måde, som ikke er ret fleksibel. `NML` kan bruges med `WRITE`, som skriver ud i samme form som en datafil der skal indlæses med `NML`; men det er nok sjældent at det bliver brugt.

IOSTAT: Her skal der angives en integervariabel, hvis værdi, efter at `read` (eller `write`) er passeret, kan inspiceres. Hvis alt gik godt, er dens værdi lige 0. Ved `READ`, hvis der var en fejl, er værdien > 0, mens hvis der var en end-of-file, men ellers ikke en fejl, bliver den < 0, og til sidst en anden < 0 værdi hvis der var en end-of-record uden anden fejl. Selve værdierne er uspecificeret i standarden og kan altså være systemspecifikke. Ofte er det nok at se, om variablen er nul eller ej. Her er et eksempel på et program, der benytter `IOSTAT` for at indlæse et antal tal i et array uden at "vide" i forvejen, hvor mange tal der er i datafilen:

```
program IOSTAT_TEST
  integer :: arr(100), n, ios
  OPEN (9, file='tal.dat', status='old')
  n = 0
  do
    read (9,*,IOSTAT=ios), arr(n+1)
    if (ios /= 0) exit
    n = n + 1
  enddo
  print '(" antal tal er:", i6)', n
end program IOSTAT_TEST
```

Programmet har en tilsyneladende uendelig løkke, men hopper ud af den, når end-of-file får variablen `ios` til at afvige fra nul. Løkken kan kaldes en *read-while* kode.

Det er en god lejlighed at pointere, at det samme som det ovenstående kan lade sig gøre ved læsning fra tastaturet i stedet for fra en fil. I så fald er der selvsagt ikke en `OPEN`-sætning, og `read`-sætningen bliver til

```
read (*,*,IOSTAT=ios), arr(n+1)
```

Det er vigtigt med de to `*`, hvor den første indikerer tastaturet, og den anden frit-format.

ADVANCE = er speciel. Det blev nævnt før, at efter en `read` eller `write` er næste record klar. Det kan imidlertid hænde, at man vil fortsætte der, hvor man lige var, og det kan lade sig gøre ved at sætte dette argument til `'NO'`. Denne facilitet er praktisk i forbindelse med `write`. Med en `read` må ikke-advancing input kun bruges ved læsning fra en ekstern fil, og et format skal specificeres. Det vil sige, at det ikke kan anvendes ved læsning fra tastaturet, og ikke med frit-format.

REC = bruges til at specificere en bestemt record i en fil med `direct` access. Med sådan en fil er det muligt at springe rundt i den ved bare at specificere nummeret på den ønskede record. Sådanne filer bruges i administrations-edb – personalelister m.m.

SIZE = er relevant i sammenhæng med ikke-advancing input, og returnerer i den variabel, der stilles til rådighed, antallet af tegn, der blev læst denne gang for at stykke indlæsningen sammen.

Det der bliver læst kan være mere end bare data-items selv, der kan være ekstra tegn som blanktegn eller kommaer, som bliver læst men ikke alle brugt til at evaluere værdierne.

ERR =, END = og EOR = skal alle have en label efter sig, tester for henholdsvis en fejl, end-of-file og end-of-record, ved indlæsning. Hvis disse sker, hopper programmet til den angivne label. De er fossilske i Fortran 90, en overlevelse fra tidligere udgaver af sproget, og er ikke anbefalet.

Nogle få eksempler kan hjælpe med disse kontrolspecs:

```
read (*,*,IOSTAT=ios) x
write (UNIT=17,FMT='(4i4)',ADVANCE='no') A
write (17,'(4i4)',ADVANCE='no') A
```

hvor `ios` er en `integer` variabel og `A` er et heltalsarray med fire elementer. De sidste to sætninger virker ens.

Interne filer

Indtil dette afsnit, når der var tale om en fil, var der tale om såkaldte **eksterne filer**, som ligger uden for programmet og skal forbindes med programmet via en `OPEN`-sætning. Der er også **interne filer**, som er et teknisk navn for hvad faktisk er tegnstrenge. Det er muligt at foretage I/O til og fra tegnstrenge, dvs. læse data fra, eller skrive data i en streng. Der er flere programmeringssituationer, hvor det er en nyttig facilitet. Eksemplerne er kommentarlinjer til datafiler for plotprogrammer, hvor man tit blander tekst med variabelværdier, og output fra instrumenter, hvor man muligvis skal lede efter informationer på en linje uden at vide i forvejen, hvor på linjen de er, og så læse dem ind.

Det er meget nemt i princippet: Her er det ikke en fil, man har at gøre med, og derfor knytter man ikke et `lun` til en fil. I stedet for at nævne et `lun` (altså `UNIT =`-delen), nævner man navnet på den pågældende tegnstreng. Her er et lidt kunstigt eksempel. Vi vil indlæse tre heltal fra en inputlinje (fra tastaturet); i stedet for direkte at indlæse dem, læser vi hele linjen som en tegnstreng, og læser så de tre tal fra den:

```
character(LEN=80) :: linje
integer           :: i, j, k
...
read '(a)', linje
if (LEN_TRIM(linje) > 0) then
  ! Internal read:
  read(linje, '(3i6)'), i, j, k
else
  ...
endif
```

Eksemplet viser en af de ting man kan med dette: se på inputlinjen for at teste den for flere ting, før man prøver at læse værdier fra den. Internal-read-linjen er mærket med passende kommentar, og det ses, at der står `linje` der, hvor der normalt ville stå et `lun`-tal.

Unformatted I/O

Indtil dette afsnit har der været tale om input eller output i form af tekst. Tekst siges at have format, selv om der bruges frit format, hvor det er systemet, der bestemmer formatet eller finder ud af det (ved input). I kapitel 14 gås der i detaljer om, hvordan information er lagret i maskinen, men her er det nok at sige, at når der udskrives et tal som, for eksempel, `1.234E+05`, så er der tale om en række teksttegn, her bestående af ni tegn (bytes). Selve værdien, der bliver udskrevet på den måde, ligger imidlertid i maskinens hukommelse i en mere kompakt form, som et såkaldt binært tal eller mønster (f.eks. 32 bits eller fire bytes). Hvis man vil have en mere kompakt datafil udskrevet, kan man vælge at udskrive alt i denne kompakte binære form, uden at "formatere" det om til tekst. Det gør man ved at bruge, i `OPEN`-sætningen,

```
OPEN (..., FORM='unformatted')
```

Eftersom der intet format er, skal (må!) der selvfølgelig heller ikke angives en formatbeskrivelse. En `write`-sætning kan så være så simpel som

```
write (17) i, j, k
```

og en `read`-sætning ligeledes. Fordelen ved dette er, som nævnt, at sådan en datafil er mindre i omfang med en faktor ca. 3-10. Der er imidlertid nogle ulemper. For det første kan man ikke umiddelbart læse en uformateret fil, eftersom den ikke består af teksttegn. Det kræver et andet program at få indholdet at se (programmet bruger tilsvarende `read(17)`, for eksempel, og kan så printe tallene som tekst). Desuden er tekstfiler som regel transportable fra en maskine til en anden, selv hvis de to bruger forskellige operativsystemer, fordi der er en høj grad af standardisering mellem maskiner for tekst. Det er der ikke for binære data, og man er nødt til, ved transport, at skrive et program på den ene maskine, der læser datatallene uformateret og skriver dem ud igen som tekst, før man kan overføre disse til den anden maskine.

I dag har man ofte forholdsvis meget diskplads, og binære filer er ved at blive mere sjældne, efter at motiveringen for dem – pladsbesparelse – er mindre påtrængende.

Kapitel 6
KIND

6.1 Hvad er KIND?

I tidligere kapitler blev det nævnt, at der er forskellige **typer** variabler og konstanter – `integer`, `real`, `logical` og `character`, såvel som den gamle type, `double precision`, som nu bliver gjort helt overflødig i dette kapitel, og der skal gives en forklaring på hvorfor. Der er også typen `complex`, men det er praktisk at anse den som en parring af to `real`, hvilket den også er.

Inden for disse typer er der flere varianter. Heltal (`integer`s) kan have forskellige maksimum antal cifre; man kunne forestille sig et talområde (`range`) af, for eksempel, $-999\ldots+999$. Sådan et område er karakteriseret af det maksimale antal cifre, her 3. De fleste maskiner tilbyder et antal forskellige områder for heltal, hvor ofte det normale (default) er ca. 5 eller ca. 10 cifre. De såkaldte 16-bits-maskiner har en standardform for heltal, der har området $-32768\ldots+32767$. Disse forskellige områder for heltal danner forskellige slags `integer`, og de er kaldt **KIND**s i Fortran 90. De forskellige KINDs er alle nummererede, og det er muligt at operere udelukkende med numrene. Hvis man for eksempel ved, at heltal med 5 cifre har KIND nummer 1 og de med 10 cifre nummer 2, så kunne man erklære sig variabler med disse antal cifre med sætninger som disse:

```
integer(KIND=1) :: i, j, k(100)
integer(KIND=2) :: ii, ji
```

osv. Konstanter kan ligeledes angives med KIND, ved at hæfte KIND-tallet efter konstanten, kædet til den med en understreg: `100_1` eller `987654321_2`, osv. Der er imidlertid et problem, eller flere. De forskellige maskiner (systemer) tilbyder forskellige, og forskellige antal af, heltalsområder; Fortran 90 sætter ikke en standard for disse. I lyset af dette, kan der heller ingen standard være for, hvilket KIND-nummer der svarer til hvilket heltalsområde. Hvert system har sit eget sæt. En løsning på problemet kunne være, at brugeren skaffer sig, en gang for alle, den fornødne information om, hvilke KIND-numre svarer til hvad, og programmerer passende derefter; der er intrinsics der kan levere den slags information. Problemet med det er, at man ofte vil flytte sine programmer til en anden maskine med et andet system – det kan også ske, at man, samme sted, går over til en ny, bedre maskine med et andet operativsystem. Så gælder de KIND-numre ikke længere, og man er nødt til at genetablere numrene, og – hvad værre er – at kæmme alle programmer for alle steder, hvor numrene skal ændres. Det vil normalt føre til fejl, og er desuden meget ineffektivt.

Der er en løsning, i form af en intrinsic function, som skal beskrives i et senere afsnit, som gør det hele elegant og transportabelt fra system til system. Men her skal vi først blive færdig med KIND for de andre typer, og hvad de betyder.

Lige som for heltal (typen `integer`), er der forskellige KINDs for typen `real`. Værdier af denne type er repræsenteret todelt i maskinen (se kapitel 14 for en nærmere diskussion): en brøkdel, og en multiplikator i eksponentialform. De to lagt sammen minder meget om den "naturvidenskabelige" måde at skrive store eller små tal på, som for eksempel 0.602×10^{24}, hvor 0.602 er brøken og $+24$ er eksponenten. Det, der nu karakteriserer sådan et tal, er antallet af decimaler i brøken, og området for eksponenten. Det første bestemmer tallets præcision, det sidste dets mulige område både i den forstand, hvor stort og hvor lille (forskelligt fra nul), tallet må blive. Igen er der KINDs, der er numre for de forskellige muligheder; og ligesom med typen `integer` er de faktiske numre for dem systemafhængige. Den forældede type `double precision` bliver derved kun en af flere KINDs inden for typen `real`, og er derfor fossilsk i Fortran 90 nu. I tidligere udgaver af Fortran var det nemlig ikke sikkert, præcis hvad denne type betød, man vidste kun, at dens præcision var noget mere end det dobbelte af den normale `real`. Det gav også udslag i den måde, man skrev konstanter på: hvor man skrev en `real` som, for eksempel, `3.0E+08`, skrev man den tilsvarende `double precision`-konstant som `3.0D+08`, hvor `D`'et indikerede den dobbelte præcision. I Fortran 90 ville man nu etablere det passende KIND-nummer (lad os sige, det var 3), og skrive konstanten som `3.0E+08_3`. Der skal beskrives, i et senere afsnit, en endnu bedre måde at skrive tallet på.

Der er yderligere to typer, der har forskellige KINDs. Der er næppe fornuft i det for typen `logical`, i hvert fald ikke med den måde, KIND er implementeret for typen. En logisk værdi har

kun to muligheder: `.false.` og `.true.` og det kunne godt lade sig lagre i form af en enkelt bit. Men de fleste systemer bruger hele ord (se kap. 14) af 16 eller 32 bits til en enkelt `logical`. I princippet har man indflydelse på dette ved at bruge KIND; og nogle systemer vil, for eksempel, tilbyde en KIND for typen `logical`, der pakker flere af dem bit-vis i ord. Men det er ikke sikkert, og slet ikke transportabelt, og der er ingen måde at specificere, hvad man vil, som der er for talområder for `integer` og `real`. For god ordens skyld skal det nævnes, at `logical` konstanter kan forsynes med KIND ved at hænge understreg og KIND-tallet efter, altså for eksempel `.true._2` osv.

Den sidste type der har KINDs, er `character`, og her er der reelle og interessante muligheder. De er dog ikke påkrævet af standarden, og mange systemer støtter dem slet ikke. For `character` betyder KIND forskellige tegnsæt, som for eksempel det normale latinske bogstavsæt, eller det kyrilliske, Hindi-sættet, de japanske stavelser eller de kinesiske tegn. Det er hovedsageligt i de lande, hvor disse sæt er relevante, at man har disse KINDs til rådighed. Vi springer dem over.

6.2 Praktisk brug af KIND

Der er mulighed for selv at bestemme KIND på en transportabel måde, dvs. sådan, at man trygt kan flytte et program fra et system til et andet. Kernen af det hele er to intrinsic functions, `SELECTED_INT_KIND(..)` og `SELECTED_REAL_KIND(..,..)`. Man ved ikke umiddelbart, for et givet system, hvilke KIND-numre er givet til hvilken slags tal (i dette afsnit er der kun tale om tal, `integer`, `real` og `complex`). Men numrene er heller ikke af interesse i sig selv; det er områder og præcision vi gerne vil styre. De to intrinsics kan fortælle programmet, hvilken KIND en given specifikation har. Men først skal der opbygges nogle vigtige elementer til metoden.

SELECTED_INT_KIND

Denne intrinsic er en function, og har kun et enkelt argument, et heltal som angiver maksimumantallet af ciffre i heltalstypen. En sætning som

```
n = SELECTED_INT_KIND (4)
```

vil altså give `n` den KIND-værdi, der svarer til heltalstypen med ca. 4 cifre. Der er en vigtig ting her. Eftersom forskellige systemer tilbyder kun et begrænset sæt KINDs for hver type kan man ikke være sikker på, om netop det antal cifre, man ønsker sig, faktisk er repræsenteret. Reaktionen er i så fald, at systemet vælger den type, den har, som er den næstbedre. Hvis der for eksempel skulle være heltal med 5 og 10 cifre, og vi har "bedt om" 4, så vil variablen `n` få den KIND-værdi, der svarer til 5 cifre. Det interessante fra synspunktet af transportabilitet er, at der er garanti for mindst de 4, lige meget hvilket system programmet kører i. Præcis hvordan man nu udnytter denne intrinsic, følger i et senere underafsnit.

Det er muligt i sådan en sætning at specificere et antal cifre, som systemet slet ikke har, som for eksempel

```
n = SELECTED_INT_KIND (99)
```

hvilket næppe er til rådighed. I så fald får `n` værdien -1, og det kan programmet teste for og reagere passende på.

SELECTED_REAL_KIND

Her er der to ting vi gerne ville styre, præcision og eksponentområdet. Functionen har derfor mulighed for to argumenter. I det følgende eksempel specificeres der en KIND der garanterer 5 decimalers præcision og et område fra 10^{-30} til 10^{+30}:

```
m = SELECTED_REAL_KIND (5,30)
```

hvor 30 betyder $10^{\pm 30}$. Som før nævnt er resultatet `m` det KIND-tal, der lige tilfredsstiller specifikationen. Hvis der altså ikke er 5 decimalers præcision, eller området ± 30, til rådighed, bliver `m` til den KIND, der er den næstbedre; i dette tilfælde måske henholdsvis 6 decimaler og ± 38.

Ofte er man mindre fikseret på eksponenten, der som regel er tilstrækkelig som den er per default. Den kan godt udelades, og

```
m = SELECTED_REAL_KIND (5)
```

er nok. Standarden tillader også at udelade det første af de to argumenter, men det er svært at forestille sig, at man vil det. Hvis man absolut vil det, kan man bruge "keyword"-kaldet som beskrevet sidst i kapitel 4. De to keywords er her `P` (for præcision, den første) og `R` (eksponentens område). Vil man kun specificere den anden af de to, kalder man proceduren med for eksempel

```
m = SELECTED_REAL_KIND (R=30)
```

Igen er det i princippet muligt at specificere en umulig KIND. Hvis `p` er umulig, men `r` er i orden, er resultatet -1. Hvis det er omvendt, er det -2. Hvis begge er umulige, er resultatet -3.

Erklæringer med KIND-specifikation

Indtil nu har erklæringssætninger (kapitel 2) ikke vist muligheder for at specificere KIND; det skal der nu rettes op på. For alle typer er der en yderlig parameter i erklæringen, der specificerer KIND. Eksempler er:

```
integer(KIND=0) :: i, j, k
real(KIND=1)    :: x, y(-10:10)
complex(KIND=2) :: z
```

Igen blev der brugt konstanter; det skal der snart laves om på. I den tredje sætning er en `complex` variabel erklæret med KIND nummer 2; det bevirker, at begge elementer af det komplekse tal får den KIND af `real`, hvad ellers det må være.

Brug af PARAMETER

De to intrinsics kan bruges i sætninger, hvori **parameters**, dvs. konstanter med navne, erklæres. Derefter står parameteren, i enheden hvor erklæringen er kendt, som KIND-tallet selv. Man opfinder passende navne for de KINDs man vil have. Ofte er de to slags heltal kaldt `short` og `long`, stående (eksempelvis) for 16- og 32-bits heltal. Forfatteren af denne bog har som vane at kalde `real` variabler med dobbelt præcision for `dbl`, og dem med enkelt, `sgl` – det sidste er dog noget overflødigt, eftersom det er normalt defaulted af systemet og behøver ingen KIND-bestemmelse. Det hele kan eksempelvis se sådan ud:

```
integer, parameter :: &
    short = SELECTED_INT_KIND(5),   &
    long  = SELECTED_INT_KIND(10),  &
    sgl   = SELECTED_REAL_KIND(6),  &
    dbl   = SELECTED_REAL_KIND(14), &
    ext   = SELECTED_REAL_KIND(30)
```

Det ses, at de to functions bliver kaldt i selve erklæringssætningerne. Typen der har KIND-tallet `ext` er den såkaldt "extended precision" type, som nogle maskiner tilbyder, også kaldt "quad precision", eftersom den har den firedobbelte af den præcision som `real` har. Det skal også bemærkes, at alle fem parametre er af typen `integer`, og repræsenterer (står for) KIND-talkonstanter. Fidusen er nu, at de kan bruges alle steder hvor et tal styrer en KIND, i erklæringerne.

Elementerne sættes sammen

Nu har vi samlet alle fornødne elementer til at skrive transportable programmer. Her er et eksempel, der bruger det hele:

```
integer, parameter :: &
    short = SELECTED_INT_KIND(5),   &
    long  = SELECTED_INT_KIND(10),  &
    sgl   = SELECTED_REAL_KIND(6),  &
    dbl   = SELECTED_REAL_KIND(14)

integer(KIND=short) :: i, j, k
integer(KIND=long)  :: L
real(KIND=sgl)      :: x, y, , pi_sgl
real(KIND=dbl)      :: arr(100), pi_dbl

pi_sgl = 4 * ATAN(1.0)
pi_dbl = 4 * ATAN(1.0_dbl)
```

Bemærk, at efter de fire linjer, hvor alle KIND-værdier bliver fastlagt, bruges de som de konstanter de er, både i efterfølgende erklæringer, og til at specificere KIND for en konstant. I den sætning, hvor `ATAN` bliver kaldt med argument `1.0`, bliver den af default KIND, normalt enkelt præcision, man kunne have tvunget enkelt præcision ved at angive tallet som `1.0_sgl`, men det er overflødigt. I næste sætning er argumentet `1.0_dbl`, dvs. 1-tallet med dobbelt præcision. Eftersom `ATAN` er en generisk function (kap. 4), retter resultatet sig efter argumentets type (og KIND), og regner med samme præcision.

Det gode ved denne måde at erklære på er, at man nemt kan ændre KIND for en hel stribe variabler i programmet ved bare at ændre et enkelt tal i sætningerne der bruger `SELECTED_..._KIND`.

Nu med modules

Det foregående er endnu ikke helt optimalt. Det er fordi alle disse KIND-parametre ville være lokale til den enhed, i hvilken de står. Et program består mest af et hovedprogram samt en række underprogrammer, og de skal gerne alle bruge de samme KINDs. Det kan realiseres ved brug af en module, hvori man lægger alle KIND-definitioner. Hver procedure i programmet kan så benytte sig af modulen ved at `use` den. Den følgende kode viser det hele:

```
module KIND_DEFS
  implicit none
  integer, parameter :: &
      short = SELECTED_INT_KIND(5),   &
      long  = SELECTED_INT_KIND(10),  &
      sgl   = SELECTED_REAL_KIND(6),  &
      dbl   = SELECTED_REAL_KIND(14)
END module KIND_DEFS

program NOGET
  use KIND_DEFS
  implicit none

  integer(KIND=short) :: i, j, k
  integer(KIND=long)  :: L
  real(KIND=sgl)      :: x, y, , pi_sgl
  real(KIND=dbl)      :: arr(100), pi_dbl

  pi_sgl = 4 * ATAN(1.0)
  pi_dbl = 4 * ATAN(1.0_dbl)
  ...
  call SUB (x, y, i,  pi_dbl)
  ...
```

```
END program NOGET

subroutine SUB (a, b, n, pi)
  use KIND_DEFS
  implicit none

  integer(KIND=short) :: n
  real(KIND=sgl)      :: a, b
  real(KIND=dbl)      :: pi
  ...
END subroutine SUB
```

(Sætningen `implicit none` blev inkluderet for at vise god programmeringsskik). `Use`-sætningerne gør det muligt at lade KIND-parametrene være kendt i alle enheder.

6.3 KIND i intrinsic procedurer

Det blev, af naturlige årsager, ikke nævnt i kap. 4 at flere af de såkaldte intrinsics har KIND som et af argumenterne – altid det sidste, og valgfrit, hvilket gør det muligt ved et kald med "positional argument"-måden, at udelade dette argument. De fleste af disse indbyggede procedurer, der har KIND som argument, er konverteringsprocedurer. Her er en lille udpluksliste (Bilag A giver en fuld liste af alle intrinsics):

- **AINT (A, KIND)** trunkerer `real A` til heltal, men af typen `real` med KIND som givet.
- **ANINT (A, KIND)** afrunder `A`, ellers som **AINT**.
- **CMPLX (X, Y, KIND)** (`X` og `Y` `integer` eller `real`) sætter `X` og `Y` sammen til et komplekst tal, begge componenter `real` og den givne KIND.
- **INT (A, KIND)** trunkerer `real A` til heltal, resultatet er `integer` af KIND som givet.
- **REAL (A, KIND)** konverterer argumentet `A`, som kan være `integer`, `real` eller `complex`, til en `real` af slags KIND.

Ved at inkludere `KIND` i argumentlisten kan man altså styre resultatets KIND. Ved heltal kan man bruge det til at undgå divisionsproblemer i forbindelse med heltal. Det er velkendt, at et udtryk som `3/4` giver resultatet `0`. Men vil man bruge det til at udtrykke 0.75, kan man opnå det med et udtryk som `3/REAL(4,dbl)`, for eksempel, hvor udtrykket nu bliver til 0.75 og oven i købet med den ønskede KIND (her `dbl`, som defineret ovenfor). For `real` tal kan functionen tjene konvertering fra en KIND til en anden; for `complex` tal er resultatet den absolutte værdi af komplekstallet, af typen `real` og slags KIND.

6.4 Andre procedurer associeret med KIND

Ud over konverteringsprocedurer findes der også en stribe såkaldte "inquiry functions", der kan associeres med KIND. Når man, for at få en bestemt KIND, specificerer præcision og område uden at være sikker på, præcis hvilken man får (den næstbedre), så vil man måske finde ud af disse, samt nogle andre egenskaber. Her er en kort beskrivelse af nogle procedurer af den slags (alle er fuldt beskrevet i Bilag A):

- **KIND (X)** giver KIND-tallet for `X`, alle typer tilladt.
- **DIGITS (X)** giver antallet af binære cifre i `X`.
- **PRECISION (X)** (`X` `real`) giver antallet af decimalcifre i `X`.
- **RANGE (X)** (`X` `real`) giver ABS(maksimum eksponent) som decimaltal.
- **BIT_SIZE (I)** (`I` heltal), antal bits i `I`.

Functionen `KIND` giver, for typerne `integer` og `real`, samme resultat som de tilsvarende `SELECTED_*_KIND` functioner, men med andre argumenter. Hvor man i de to `SELECTED...` specificerer antal cifre osv., specificerer man ved `KIND` en værdi, der har disse egenskaber. Har man for eksempel brugt den ovenstående module `KIND_DEFS`, og befinder sig i hovedprogrammet `NOGET`, så vil brugen af `KIND(pi_sgl)` resultere i samme værdi som konstanten `sgl`, osv. Med konstanter som argument til `KIND` kan der være lidt forvirring. Et kald som `KIND(0.9999999999)` vil, selvom konstanten synes at have mere end enkelt præcision, give den `KIND`-værdi som konstanten med kun få 9-taller. Det er fordi systemet forkorter tallet til standard-`real`. For at indikere den højere præcision, er man derfor nødt til at tvinge den med `KIND(0.9999999999_dbl)`, hvilket kan forekomme lidt overflødigt. Det er nok sjældent, man bruger `KIND`-functionen med konstanter.

Kapitel 7
Mere om arrays

I dette kapitel skal der gøres op med yderligere nogle arrayfaciliteter, ikke nævnt i kapitel 3. I kapitel 8 kommer der så de aspekter af arrays, der har at gøre med procedurer, dvs. de måder, på hvilke arrays kan blive erklæret i procedurer, de bliver overført til.

7.1 Indeksvektorer

I kapitel 3 bliver **arrayudsnit** beskrevet. Det er således muligt at henvise til en underdel af et array, ved at specificere, for hver dimension, en start- og slutindeks, samt springinterval. For eksempel, hvis vi har med et endimensionalt array `A(10)` at gøre, kan vi henvise til et nyt array, bestående af `A`-elementerne nr. 1, 3, 5, 7 og 9, som `A(1:9:2)`. De tre tal er konstanter, og mekanismen tillader kun temmelig regulære udsnit af et array. Det kan være, at man vil udtage en række elementer med vilkårlige indekser, for eksempel (stadig ved `A`) elementerne nr. 1, 5, 6 og 10. Det kan lade sig gøre ved at bruge en **indeksvektor**, som i sig selv er et array og indeholder disse indekstal:

```
integer :: vec(4) = (/ 1, 5, 6, 10 /)
integer :: A(10)
integer :: B(4)
...
B = A(vec)
```

Arrayet `B`, som kun består af de fire elementer, kommer derved til at indeholde de nævnte elementer af `A`. Eksemplet kan udvides til flere dimensioner, og det er i orden at blande indeksvektorer med normale konstanter. Hvis vi for eksempel har et todimensionalt array Z(5,7) og vektorerne U(3) og V(4), vil følgende sætninger,

```
U = (/ 1, 3, 2 /)
V = (/ 2, 1, 1, 3 /)
```

og henvisning til `Z(3,V)` give de fire elementer af Z fra række 3, `Z(3,2), Z(3,1), Z(3,1), Z(3,3)`. Læg mærke til, at rækken af indekserne i både `U` og `V` ikke er i en bestemt rækkefølge og indeholder gentagelser, hvilket er helt i orden. En henvisning til `Z(U,2)` er ensbetydende med søjlen 2, elementerne

```
Z(1,2)
Z(3,2)
Z(2,2)
```

hvorimod `Z(U,V)` skaber et todimensionalt `3*4` array med elementerne

```
Z(1,2) Z(1,1) Z(1,1) Z(1,3)
Z(3,2) Z(3,1) Z(3,1) Z(3,3)
Z(2,2) Z(2,1) Z(2,1) Z(2,3)
```

De ovennævnte arrayudsnit, som er dannet af vektorer, hvor nogle indekser er gentaget, har navnet **many-one arrays**. Sådanne arrays må ikke stå på venstre side af en tilordningssætning eller som modtager af en `read`-sætning. En anden regel er, at et arrayudsnit specificeret af en eller flere vektorer må godt overføres til en procedure, men den må ikke forsøge at ændre (eller sætte en værdi i) arrayet; det er den samme regel som gælder konstanter. I terminologien for procedurer (se næste kapitel) betyder det, at det tilsvarende *dummy argument* i proceduren ikke må have `INTENT OUT` eller `INOUT`. Sidst er der den regel, at en pegepind (se kapitel 10) ikke må pege på et sådant arrayudsnit, dvs. et arrayudsnit specificeret af en indeksvektor må ikke være *target* for en *pointer*.

7.2 Reshape

Arraykonstanter kan bruges til at give startværdier til arrays, hvor man vil give forskellige værdier til de forskellige elementer. Hvis man vil give den samme værdi til et helt array, er det let ved brug af en simpel sætning som

```
A = 0
```

osv. Men en arraykonstant er altid en endimensional struktur, og det kan være, at man vil give et flerdimensionalt array værdier. Det kan lade sig gøre ved at "krumbøje" eller omforme en arraykonstant til at sprede den over flere dimensioner. Det gøres ved den intrinsic function `RESHAPE`. Den kaldes (eller rettere sagt, nævnes) generelt med fire argumenter, hvoraf de to sidste er valgfrie:

```
...RESHAPE (SOURCE, SHAPE, PAD, ORDER)
```

hvor alle fire skal være arraykonstanter, den første af en hvilken som helst type og de andre tre af typen `integer`. `SOURCE` er den arraykonstant, dvs. den stribe af værdier, der skal omformes til et flerdimensionalt array; `SHAPE` er en beskrivelse af de dimensioner, det nye array skal have. Normalt

er det nok med de to. Et eksempel kunne være, at vi vil fylde et 2*4 array arr med værdier, fra en konstant:

```
arr(1:2,1:4) = RESHAPE ( &
          (/ 1,2,3,4,5,6,7,8 /), (/ 2,4 /))
```

(vi kunne godt have skrevet arr i stedet for arr(1:2,1:4), hvis arr er erklæret som værende 2*4). Resultatet er:

```
1  3  5  7
2  4  6  8
```

(bemærk ordenen af tallene, som følger array-ordenen, beskrevet i kap. 3, altså søjlevis for et todimensionalt array). En regel her er, at argumentet SHAPE ikke behøver at være langt nok (i antal elementer); hvis der mangler noget, skal det tredje argument, PAD, bruges: med det kan vi selv bestemme, hvilke værdier der skal erstatte de manglende i konstanten. Hvis vi for eksempel har en vektor vec bestående af kun de seks værdier (/ 1,2,3,4,5,6 /), kunne der i stedet for det ovenstående, stå

```
arr =  RESHAPE ( &
          vec, (/ 2,4 /), PAD=(/ 77, 88 /))
```

og resultatet bliver

```
1  3  5  77
2  4  6  88
```

Der blev altså udfyldt (engelsk *padded*) med vektoren [77,88] her.

I disse eksempler blev tallene fordelt i det modtagende array efter den array-orden som er beskrevet, her søjlevis. Men den orden kan også bestemmes, ved det fjerde argument, ORDER. I de ovenstående eksempler var det underforstået, at dette argument er lig med (/ 1, 2 /), hvilket siger første dimension først, anden derefter (standardorden). Vender vi dem om:

```
arr = RESHAPE ( &
      vec, (/ 2,4 /), (/ 0,0 /), (/ 2,1 /))
```

er resultatet (PAD er nu sat til nuller)

```
1  2  3  4
5  6  0  0
```

Et sidste eksempel:

```
arr = RESHAPE (vec, (/ 2,4 /), &
           ORDER=(/ 2,1 /), PAD=(/ 0, 0 /))
```

giver samme resultat, og desuden kan det ses, at et kald med keywords tillader en ændring i rækkefølgen af argumenterne.

Til sidst skal det nævnes, at RESHAPE kan blive brugt til at initialisere arrays som led i arrayerklæringer. Et eksempel kan være

```
integer :: arr(2,4) = RESHAPE ( &
          (/ 1,2,3,4,5,6,7,8 /), (/ 2,4 /))
```

7.3 WHERE

Dette *keyword* bruges på en måde, der minder om en IF-sætning, men er noget andet. Dets opgave er, på engelsk, en *masked array assignment*, hvad der kan oversættes løst til "selektiv arraytilskrivning". Arraytilskrivning er at tilskrive et array værdier af et andet, som

```
integer :: A(10,10), B(10,10)
...
B = A
```

Her får array B, ubetinget, alle værdier i A, element for element. Men det sker, at vi ønsker en vis selektivitet i tilskrivningen. Vi vil for eksempel kun overføre de elementer af A som er positive. WHERE-sætningen har de to former

```
WHERE (<arraybetingelse>) <sætning>
```

eller, mere generelt

```
WHERE (<arraybetingelse>)
  <sætning(er)>
ELSEWHERE
  <sætning(er)>
ENDWHERE
```

Alle arraysætninger skal involvere hele arrays, og de skal altid have samme dimensioner og størrelser som det array, der ses på i betingelsen. Nogle enkelte eksempler skal gøre det klart. Vi starter med to 4*4 arrays, A og B, med A indeholdende

```
-1  77  77  77
77  -1  77  77
77  77  -1  77
77  77  77  -1
```

og B indeholdende

```
 1   1   1   1
 1   1   1   1
 1   1   1   1
 1   1   1   1
```

Efter sætningen,

```
WHERE (A<0) B = 0
```

er kørt, er B nu

```
 0   1   1   1
 1   0   1   1
 1   1   0   1
 1   1   1   0
```

og efter operationen på selve A,

```
WHERE (A<0) A=1
```

bliver `A` til

```
 1  77  77  77
77   1  77  77
77  77   1  77
77  77  77   1
```

Herefter køres `WHERE`-sætningen i sin generelle form,

```
WHERE (A<2)
  A = 1
ELSEWHERE
  A = 0
ENDWHERE
```

og det laver `A` om til

```
1   0   0   0
0   1   0   0
0   0   1   0
0   0   0   1
```

7.4 Dynamiske arrays

Hvis et givet array har forskellige størrelser fra en programkørsel til en anden, kan det være en fordel at bruge dynamiske arrays. Man meddeler oversætteren at et bestemt array skal have et vist antal dimensioner, men at selve størrelsen i hver dimension ved oversættelsens tidspunkt endnu er ukendt; de bliver sat senere, når programmet kører. Det giver altså mulighed for at sætte antallet af elementerne lig med netop det, der er brug for; og desuden kan man, efter brugen, give slip igen på den plads, et sådant array optager.

Dette klares med egenskaben (attributtet) `allocatable` i erklæringen af et array, samt sætningerne `ALLOCATE` og `DEALLOCATE`. Det er meget enkelt, dog er der nogle restriktioner. Et array er erklæret med, for eksempel, sætninger som

```
integer, allocatable, &
      dimension(:,:) :: A
integer, allocatable :: B(0:,0:)
allocatable          :: C(:,:)
integer              :: C
integer             :: nA,mA,nB,mB,nC,mC,err

...         ! (nA, mA, osv fastsaettes)

ALLOCATE (A(nA,mA), B(nB,mB), &
                    C(nC,mC), STAT=err)
if (err /= 0) STOP " Fejl i ALLOCATE!"
...         ! (Benyttelse af arrayerne)
DEALLOCATE (A, B, C)  ! Plads frigjort igen
```

Her blev arrayerne `A`, `B` og `C` erklæret på tre forskellige måder, bare for at vise, hvad der er tilladt. Sætningen `DEALLOCATE` frigør den plads, arrayerne optager, og så er der mulighed for at allokere plads igen. Bemærk også, at i tilfældet `B` er der en lavere indeksgrænse lagt (`0`); det er helt i orden. Der allokeres plads til alle tre arrays i den ene sætning; det kan selvfølgelig gøres et array ad gangen. Den sidste del (nødvendigvis med `keyword`), `STAT=err`, tillader at teste, om allokeringen gik godt. Hvis den gjorde, skulle variablen `err` være lig nul. I praksis vil man nok ikke lade sig smide ud med det samme, hvis noget går galt her som i eksemplet.

Der er nogle regler, de fleste temmelig logiske:

1. Hvis de variabler, der benyttes i en `ALLOCATE`-sætning til at angive et arrays lavere indeksgrænse og størrelse(r) (som `nA` osv. i eksemplet) senere bliver ændret, ændrer det ikke allokeringen.

2. Hvis der opstår en fejl under udførelse af en `ALLOCATE`-sætning, der mangler `STAT=`-delen, termineres programkørslen.

3. Det er tilladt at overføre et færdigallokeret array til en procedure ligesom alle andre arrays; men ikke et allokerbart array, som ikke endnu er allokeret plads.

4. Prøver et program at allokere et allokerbart array, som er allokeret i forevejen, opstår der en fejl i `STAT=`-delen, hvis den er med i sætningen. Det er muligt at teste et array for, om det er allokeret eller ej, ved at bruge den intrinsic function `ALLOCATED` (se listen nedenfor).

5. Der opstår en fejl, hvis programmet prøver at deallokere et array, der ikke er allokeret.

6. Et array, der er allokeret inden for en procedure, går i en udefineret tilstand når programmet går ud af proceduren uden at deallokere arrayet, medmindre arrayet er erklæret med `SAVE` (se næste kapitel). Sådan et array, i udefineret tilstand kan ikke senere allokeres igen som andre, ordentligt deallokerede arrays kan.

Alle regler er ret indlysende; reglen 3 kan dog virke restriktiv. Det er altså ikke tilladt at erklære et array som allokerbart, og overlade det til en procedure at allokere det ved at overføre arrayet til proceduren. Den regel kan omgås ved at erklære arrayet i en module, som så kan `use`'s af flere enheder, hvor en af disse udfører allokeringen:

```
module ALLOKARRAY
  ...
  integer, allocatable, dimension(:,:) :: A
end module ALLOKARRAY
  ...
program BLABLA
  use  ALLOKARRAY;    implicit none
  integer :: nA, mA, fejl
  ...
  call ALLOKPROC (nA, mA, fejl)
  ...
end program BLABLA

subroutine ALLOKPROC (n, m, error)
  use  ALLOKARRAY;   implicit none
  integer :: n, m
  ...
  ALLOCATE (A(n,m), STAT=error)
  ...
end subroutine ALLOKPROC
```

Der er mere at sige om sætningerne `ALLOCATE`, `DEALLOCATE` og beslægtede andre, men de ting kommer i kapitlet om pegepinde.

7.5 Array intrinsics

Der følger her en liste af de intrinsics, der opererer med arrays. For en fuldt detaljeret beskrivelse af dem se Bilag A. Disse intrinsic procedurer kan deles op i grupper som vist.

Vektor- og matrixmultiplikation (functions):

- **DOT_PRODUCT**(vector_A,vector_B)
  Prikprodukt af to endimensionale arrays
- **MATMUL**(matrix_A,matrix_B)
  matrixmultiplikation

Arrayreduktionsfunctions:

- **ALL** (MASK[,DIM])
  `.true.` hvis alle værdier er `.true.`
- **ANY** (MASK[,DIM])
  `.true.` hvis nogen af værdierne er `.true.`
- **COUNT** (MASK[,DIM])
  Antal af `.true.` elementer.
- **MAXVAL** (ARRAY[,DIM][,MASK])
  Maksimumværdi i array.
- **MINVAL** (ARRAY[,DIM][,MASK])
  Minimumværdi i array.
- **PRODUCT** (ARRAY[,DIM][,MASK])
  Produktet af alle arrayelementer.
- **SUM** (ARRAY[,DIM][,MASK])
  Summen af alle arrayelementer.

Her er der valgfrihed, hvor der er argumenter i [...]. De to valgfrie argumenter behøver en forklaring. `MASK` er i alle tilfælde et array af `logical`; det kan dreje sig om et aktuelt array, eller et, der er skabt på stedet. Functionen `ALL` er et godt eksempel for at illustrere dette. Lade os sige, at vi har et heltalsarray `A` og et logisk array L, begge med samme dimensioner og størrelser, for eksempel `3*4`. Der er blevet udført sætningen

```
WHERE (A>0)
  L = .true.
ELSEWHERE
  L = .false.
ENDWHERE
```

(det forudsættes, at `A` indeholder nogle elementer < 0). Det giver, at alle elementer i L er `.true.` hvor de tilsvarende elementer i A er > 0, og `.false.` hvor det ikke er sådan. Hvis man nu bruger `ALL(L)`, så er resultatet værdien `.false.`, eftersom *ikke* alle elementer af `L` er lig med `.true.` Samme resultat fås der med kaldet `ALL(A>0)`. Her er det *udtrykket* `A>0`, der danner et logisk array, på stedet, og det fordamper igen kort efter.

Argumentet `DIM` er lidt mere indviklet. Hvis det mangler, er det enkelt: der er i så fald tale om hele arrayet, som i det ovenstående eksempel med functionen `ALL`. Hvis vi nu kalder functionen med `ALL(A>0,1)`, hvor vi specificerer den første dimension (ud af to), så er resultatet hele fire `logicals`. Det er fordi, når en dimension (`DIM`) er nævnt i kaldet, så anvendes der functionen hen ad den angivne dimension, gennemløbende de andre dimensioner. Her betyder det altså alle fire søjler i `A(3,4)`. Functionen giver altså resultatet af functionens kik på hver søjle. Generelt sagt forholder det sig sådan, at når man, for et array erklæret med n dimensioner, specificerer `DIM` som det i'te, så opererer `ALL` på et array med alle de andre dimensioner (uden i) og for hvert element i dette array, giver svar for alle i dimensionen i. Det er ikke lige let at begribe, og det er nok nødvendigt at afprøve det ved hjælp af nogle testprogrammer. Her er et eksempel, der muligvis hjælper, eller giver et fingerpeg mod andre testprogrammer, læseren selv vil prøve:

```
program ALL_TEST
  integer :: A(3,4)
  logical :: L(3,4)
  A = 1
  do i = 1, 3;    A(i,i) = -1;          enddo
  print *, " A:"
  do i = 1, 3;    print *, A(i,1:4);    enddo
```

```
  print *
  print *, " ALL(A>0)   =", ALL (A>0)
  print *, " ALL(A>0,1) =", ALL (A>0,1)
  print *, " ALL(A>0,2) =", ALL (A>0,2)
end ALL_TEST
```

En kørsel gav følgende output:

```
  A:
         -1           1           1           1
          1          -1           1           1
          1           1          -1           1

  ALL(A>0)   = F
  ALL(A>0,1) = F F F T
  ALL(A>0,2) = F F F
```

Den første `ALL`-værdi er bare en enkelt, og den er `.false.` fordi ikke alle elementer af `A` er > 0. Ingen `DIM` blev specificeret. I andet kald blev `DIM=1` nævnt (uden *keyword*), og det svarer til de fire søjler, som giver tre gange `F`, og et `T` for den sidste (fjerde) søjle, fordi alle tal i denne søjle er > 0. I den tredje `ALL` specificeres der dimension 2, hvilket giver tre resultater, for de tre rækker, hver især.

Functionen `SUM` er meget nyttig, den sparer at skrive `DO`-løkker. Vil man for eksempel have et program til at beregne en middelværdi af et antal n arrayelementer, skal man bare skrive en sætning som

```
  middel = SUM(arr(1:n)) / n
```

eller, hvis `arr` er af typen `integer`, for at tvinge et `real` resultat,

```
  middel = SUM(arr(1:n)) / REAL(n)
```

Til sidst (med `arr` et `real` array), findes der et meget kompakt udtryk for vektornormen af `arr`: `SQRT(DOT_PRODUCT(A(1:n),A(1:n)))`.

Disse intrinsics sparer ikke kun programmeringsarbejde (at indtaste løkker); de realiseres formodentlig i maskinen af optimeret maskinkode, og gør derfor programmerne mere effektive.

Array inquiry functions:

- **ALLOCATED**(ARRAY)
  `.true.` hvis allokeret, ellers `.false.`
- **LBOUND**(ARRAY[,DIM])
  Nedre indeksgrænse for arrayet.
- **SHAPE**(SOURCE)
  Dimensionsform af `SOURCE`.
- **SIZE**(ARRAY[,DIM])
  Størrelsen af arrayet, eller del af samme.
- **UBOUND**(ARRAY[,DIM])
  Øvre indeksgrænse for arrayet.

Den første af disse behøver ingen kommentar. Parret `LBOUND` og `UBOUND` giver et heltalsresultat. Det er et enkelt heltal hvis `DIM` er specificeret; det er i så fald den henholdsvis nedre eller øvre indeksgrænse (som den er erklæret) i den specificerede dimension. Hvis `DIM` er udeladt, er resultatet en stribe af henholdsvis nedre eller øvre indeksgrænser, løbende for alle dimensioner. Således, hvis der blev erklæret et array,

```
real :: Z(-10:10,-5:5)
```

så fås de følgende resultater for kaldene:

`LBOUND(Z,1)`	-10
`LBOUND(Z,2)`	-5
`LBOUND(Z)`	-10, -5
`UBOUND(Z,1)`	10
`UBOUND(Z,2)`	5
`UBOUND(Z)`	10, 5

Functionen `SHAPE` giver formen af et array (eller af en skalar, som også er tilladt som argument). `SHAPE` er defineret som en vektor af heltal, hvert element indeholdende antallet af elementer i den tilsvarende dimension i arrayet. Her dropper indeksgrænserne altså ud. Kalder man for eksempel denne function med det ovenstående array `Z` som argument, fås vektoren (arrayet) `21,11`. Hvis argumentet er en skalar (som ingen dimensioner har), er resultatet formelt en vektor af længden nul – altså ikke noget.

`SIZE` giver det totale antal elementer i arrayet, hvis `DIM` ikke er med, ellers antallet hen ad den angivne dimension. Fortsat med det ovenstående `Z`, er `SIZE(Z)` lig med 231 ($= 21 \times 11$), mens `SIZE(Z,1)` er lig med 21, osv. Denne function vender vi tilbage til i kapitel 8.

Bemærk, at alle arrays i de ovenstående intrinsics betyder arrays, som de er erklæret eller skabt som arrayudsnit. Nævnes der for eksempel arrayet `Z`, som sådan, menes der `Z`, med dimensionerne som erklæret, dvs. `(-10:10,-5:5)`. Men det er også i orden at bruge et arrayudsnit som argument i disse functions, og de opfører sig så som om de var erklæret med de angivne dimensioner. Således er `SHAPE(Z(1:10,1:5)` lig med `10,5`, `SHAPE(Z(0,:)` lig med 11, og `SIZE(Z(0:10,0:5)` lig med tallet 66.

Arraykonstruktionsfunctions:

- **MERGE** (TSOURCE,PSOURCE,MASK)
  Vælger `TSOURCE` hvor `MASK` er `.true.`, ellers `PSOURCE`.
- **PACK** (ARRAY,MASK[,VECTOR])

Omdanner alle `ARRAY` elementer hvor `MASK = .true.` til en vektor i arrayrækkefølge.

- **SPREAD** (SOURCE,DIM,NCOPIES)
  Spreder `SOURCE` over et array med en ekstra dimension.
- **UNPACK** (VECTOR,MASK,FIELD)
  Modsat til `PACK`.

Disse fire functions er lidt komplicerede at forklare; man får heller ikke indtrykket at de er særlig nyttige. Forklaringerne og eksempler findes i Bilag A.

Array reshaping function:

- **RESHAPE**(SOURCE,SHAPE[,PAD][,ORDER])
  En- til flerdimensional.

Arraymanipulationsfunctions:

- **CSHIFT** (ARRAY,SHIFT[,DIM])
  Cirkulær shift.
- **EOSHIFT**
  (ARRAY,SHIFT[,BOUNDARY][,DIM])
  End-off shift.
- **TRANSPOSE** (MATRIX)
  Matrixtransponering (kun for todim. arrays).

De to shift-functions kan ses i Bilag A; de er næppe af stor interesse. `TRANSPOSE` kan være mere nyttig. Man skal huske, at den transponerede er resultatet af functionens virken, og skal altså gives til et andet array, der har de passende dimensioner og størrelser, medmindre resultatet skal bruges på stedet. Det første af disse kan for eksempel være

```
real :: A(3,4), B(4,3)
...
B = TRANSPOSE (A)
```

eller, mere generelt,

```
real :: A(3,4), B(4,3)
...
B(1:4,1:3) = TRANSPOSE (A(1:3,1:4))
```

og det sidste kan eksempelvis forekomme i de situationer, hvor man vil gange en ($n \times n$) matrix med dens transponerede ($A \times A^T$):

```
B(1:n,1:n) = MATMUL ( &
         A(1:n,1:n), TRANSPOSE(A(1:n,1:n)))
```

hvor den transponerede ikke bliver gemt væk, men brugt direkte.

Arrayfindefunctions:

- **MAXLOC** (ARRAY[,MASK])
  Hvor er arrayets maksimum?
- **MINLOC** (ARRAY[,MASK])
  Hvor er arrayets minimum?
- **MAXVAL** (ARRAY[,DIM][,MASK])
  Hvad er arrayets maksimum?
- **MINVAL** (ARRAY[,DIM][,MASK])
  Hvad er arrayets minimum?

LOC-functions giver et endimensionalt array af længde lig med antallet af dimensionerne af `ARRAY`; tallene indeholder "koordinaterne" af det første element i arrayet, der har maksimum- eller minimumværdien (der kan jo være flere, lig med hinanden). Hvis en maske er specificeret, gælder det kun de elementer, der slipper igennem masken. Nogle eksempler: antag at `endim` indeholder vektoren `(/ 2,6,4,6 /)`, og `todim` er et todimensionalt array med indholdet

```
 0  -5   8  -3
 3   4  -1   2
 1   5   6  -4
```

Så er værdien af `MAXLOC(endim)` lig med 2, eftersom element nummer 2 (værdi: 6) er det første, der har den største værdi; og `MINLOC(endim,endim>2)` har værdien 3 (element nr. 3 har værdien 4), fordi masken her udelukker det allerførste element. `MAXLOC(todim,todim<6)` har som resultat vektoren `[3,2]`, fordi det er element `todim(3,2)` der er det største, når kun de elementer, der er mindre end 6 er med.

Selve værdierne, der er maksimum eller minimum, fås fra de to `-VAL`-functions.

Se i øvrigt flere eksempler med alle disse i Bilag A.

Kapitel 8
Mere om procedurer

Her skal der beskrives de mere indviklede aspekter af procedurer, der blev sprunget over i kapitel 4.

8.1 Nogle definitioner og klassificeringer

Aktionerne (sætningerne) i en procedure bliver sat i gang ved et procedurekald; enten ved en `CALL`-sætning, eller (i tilfældet af en *function*) ved en henvisning til proceduren. Sammen med at aktivere en procedure, sender den kaldende enhed et antal argumenter til proceduren, modtaget af proceduren som såkaldte *dummy arguments* (som kan oversættes til noget i retning af *skyggeargumenter*). Denne betegnelse hentyder bl.a. til, at disse, inden for proceduren kan have forskellige navne fra dem, de har i den kaldende enhed. Mere vigtigt er det, at dummyargumenter, som også skal erklæres inden for en procedure, alligevel ingen rigtig eksistens har (der bliver ikke plads reserveret til dem, for eksempel), indtil en procedure bliver kaldt, hvorefter dummyargumentet svarer til det, der er overført ved kaldet.

Procedurer kan klassificeres på forskellige måder. I kapitel 4 har vi allerede set de to klasser procedurer, subroutines (underprogrammer), som skal aktiveres ved en `CALL`-sætning, og functions, som bare skal henvises til i udtryk. De er bl.a. forskellige fra hinanden, idet de har – anvendt hensigtsmæssigt – forskellige funktioner. Underprogrammer bør udføre et antal sætninger eller opgaver; det kan være, at opgaven består i at ændre nogle af de overførte argumenter. Underprogrammets navn tjener udelukkende til at identificere underprogrammet og at gøre det muligt at kalde det. Functions bør bruges til at skabe en enkelt værdi (som dog kan være et array), og denne værdi gives til sidst (som regel) til functionens navn, som i dette henseende opfører sig som en variabel. Således er den sidste sætning i functionen `COT`, i kapitel 4,

```
COT = 0
```

En function burde aldrig ændre værdierne af de indkommende argumenter. Dette er helt lovligt, men er dårlig programmeringspraksis (og kan forhindres ved brug af `INTENT`, se nedenfor).

En yderlig klassificering af functions er (for intrinsic functions) at de kan være **elemental**, hvilket betyder at de kan operere på arrays og returnere arrays af værdier. De fleste matematiske intrinsics er elementals.

En anden måde at klassificere procedurer på er klasserne intrinsic, intern, ekstern, modul- og dummyprocedurerne, samt en statement function. **Intrinsics** blev beskrevet i kapitel 4. En **intern** procedure er en, der er indeholdt i en programenhed ved hjælp af en `CONTAINS`-sætning (se kapitel 4, afsnit 4.8) og er lokal til den enhed. Det vil sige, at en intern procedure udelukkende kan kaldes fra den enhed, den er intern i (undtagen en, der sidder i en module). En **ekstern** procedure er en selvstændig enhed der kan benyttes af alle andre enheder der er i stand til at kalde en procedure. Den kan være færdigoversat separat og for eksempel ligge i et procedurebibliotek, og også være skrevet i et helt andet sprog end Fortran. En **modulprocedure** er en (intern) procedure defineret i en module. Den ligner en intern procedure, med den forskel, at eftersom en module kan være tilgængelig til enhver programenhed der vælger adgang til modulen (ved en `USE`-sætning), så kan en procedure intern til en module være lige så tilgængelig. På denne måde er det muligt at begrænse tilgængeligheden af en procedure, hvilket ikke er muligt med en normal, ekstern procedure. En **dummy** procedure er en procedure, som er overført til en anden som argument. I den modtagende procedure har den så (sandsynligvis) et andet navn og bliver til en dummy. Der bliver givet mere information om disse senere i kapitlet. Til sidst er der klassen af **statement functions**, som er blevet beskrevet tilstrækkeligt i kapitel 4.

8.2 SAVE

Eftersom procedurer bliver kaldt og derefter bliver forladt igen, kan det ske, at lokale variabler interne til en procedure mister deres værdier – eller rettere sagt, mister deres ret til plads i maskinens hukommelse, og derfor bliver udefinerede. Det samme kan ske med en module, hvori der er erklæret variabler. Meningen med dette er at gøre disse variabler universelt tilgængelige, d-vs. tilgængelige for alle de programenheder, der knytter sig til modulen ved en `USE`-sætning. I de

sjældne tilfælde, hvor der er en USE i en procedure, men ikke i den enhed, der kalder proceduren, kan det ske, at variabler interne til modulen bliver udefinerede.

Der er imidlertid tit behov for at bevare sådanne variabler fra for eksempel det ene kald af en procedure til det næste; eller at bevare de universelle variabler i en module, selv når modulen ikke er i brug. Dette kan sikres med SAVE, som kan være enten et attribut (en egenskab) lagt til en erklæring eller en sætning i sig selv. Et eksempel kan illustrere nyttigheden af SAVE. Ideen er, at functionen RUNNING_AV skal returnere værdien af det nuværende gennemsnit af alle argumenter i alle kald af functionen, op til det sidste. Det giver en løbende gennemsnitsværdi, som ofte er af interesse i statistikken. Det n-te gennemsnit $u(n)$ er matematisk defineret som

$$u(n) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

hvor x er de tal, hvis gennemsnit skal beregnes. Her er en function der skal gøre dette:

```
real function RUNNING_AV (x)
  implicit none
  real :: x

  integer :: n=0
  real    :: sum=0.0
  save    :: n, sum

  sum = sum + x;   n = n + 1
  RUNNING_AV = sum / n
end function RUNNING_AV
```

Der ses en SAVE-sætning (det skal vise sig, at den faktisk er overflødig). Den kan have forskellige former. Listen af variablerne, for eksempel, kan udelades:

```
  save
```

i hvilket fald alle variabler lokale til proceduren bliver bevaret (her: n og sum). Man kan også inkludere SAVE-attributtet i erklæringerne, i stedet for at bruge en SAVE-sætning:

```
real function RUNNING_AV (x)
  ...
  integer, save :: n=0
  real, save    :: sum=0.0
  ...
```

Der er nogle regler omkring SAVE:

1. SAVE må bruges i et hovedprogram, men har ingen effekt der, eftersom alt er bevaret i et hovedprogram i forvejen.

2. Hvis en SAVE-sætning uden variabelliste bliver brugt i en programenhed, må der ikke være yderligere SAVE i samme enhed, hverken som sætning eller som attribut.

3. Det er ikke tilladt at erklære SAVE-attributtet for et argument overført til en procedure (altså et dummyargument).

4. En variabel der er initialiseret i en enhed har attributtet SAVE per default.

Den sidste regel siger, at det var unødvendigt at bruge save i eksemplet på sidste side, fordi de to variabler (n og sum) begge er initialiserede; men det gør ikke noget.

8.3 RESULT

I kapitel 4 blev functions beskrevet. En function har et navn, og den ligner en variabel, eftersom symbolet (navnet) bliver tilskrevet en værdi i functionens krop. Det er imidlertid muligt at give værdien til et andet symbol, som står for det sted, hvor værdien skal havne. Eksemplet i kapitel 4 kunne altså se sådan ud:

```
real function COT (arg) result (value)
! Function til at beregne cotangens af arg.
  real :: arg
  if (ABS(arg) > 1.0E-10) then
    value = 1.0 / TAN (arg)
  else
    print *, "Nul argument for COT!"
    value = 0
  endif
end function COT
```

Functionen kaldes (bedre: henvises til) på samme måde som den udgave vist i kapitel 4. Bemærk at den ny variabel value ikke er erklæret i selve functionen; den er allerede erklæret som en real i functionens hoved. **Result**-delen kan forekomme noget overflødig; men i rekursive functions kan det blive nødvendigt at bruge **result** (se et senere afsnit om rekursion).

8.4 INTENT

Når variabler bliver overført til en procedure, uden at der siges mere, har proceduren fri adgang til disse variabler og kan for eksempel ændre dem som den har lyst. Det er dog ofte programmørens intention at nogle variabler kun skal tjene som input til proceduren og ikke skal røres ved, andre skal komme ud af proceduren som resultat af dens virken, og endnu andre skal komme

ind og blive ændret. Programmøren kan beskytte sig selv mod forkerte handlinger ved brug af `INTENT`, som specificerer hvad der er ment. Der er alle de tre ovennævnte muligheder: man kan sætte `intent` som `in`, `out` eller `inout`. Som det er med erklæring af arrays, kan man enten angive `intent` separat eller på samme linje som typen. Her er et lidt kunstigt eksempel, der bruger alle tre slags `intent`. Proceduren skal tage argumentet `arg` (og ikke røre ved dets værdi), returnere argumentets kubikrod, og lægge roden til en i forvejen eksisterende løbende sum:

```
subroutine STUFF (arg, rod, loebesum)
  implicit none
  real, intent(in)  :: arg
  real, intent(out) :: rod
  real              :: loebesum
  intent(inout)     :: loebesum

  rod = SIGN ((ABS(arg))**(1.0/3.0), arg)
  loebesum = loebesum + rod
end subroutine STUFF
```

Bemærk at variablen `loebesum` figurerer i to erklæringssætninger; `intent` er angivet separat for den, mens det er inkluderet i typeerklæringerne for de andre to. En lille sidebemærkning om proceduren: Grunden til brug af `SIGN` og `ABS` er, at maskinen vil nægte at tage kubikroden af `arg` hvis `arg` < 0 – selvom den er matematisk defineret. Vi evaluerer derfor kubikroden af det positive `ABS(arg)`, og lægger (ved brug af `SIGN`) fortegnet af `arg` til resultatet.

8.5 Explicit interface

Når en enhed kalder en procedure uden at der er en **explicit interface**, siges der at være en **implicit interface** mellem den kaldende enhed og proceduren. Så er der en vis tillid til, at de argumenter der overføres til proceduren, bliver modtaget som det er tænkt. Dette kan gå galt, uden at oversætteren bemærker noget. Følgende procedure,

```
subroutine SUBROUT (arg)
  real :: arg
  ...
end subroutine SUBROUT
```

kan for eksempel kaldes fra denne enhed

```
  integer :: argu
  ...
  call SUBROUT (argu)
  ...
```

og det går ikke godt. Den slags fejl giver mærkelige resultater (se også kap. 14). Den kaldende enhed og proceduren har forskellige opfattelser af, hvad argumentet er for noget, hvilken type det har, og de ved ikke om hinandens opfattelser. Ved at bruge en **explicit interface** kan der oprettes et signalement i den kaldende enhed af, hvad proceduren mener, hvordan den erklærer det overførte argument. Det gøres i en såkaldt **interface blok** enten i den kaldende enhed, eller – mere sædvanligt – i en module som enheden benytter sig af. Er blokken indeholdt i selve enheden, skal den ligge blandt (normalt: lige efter) erklæringerne. Den indeholder en præcis afspejling af de erklærende dele af proceduren, samt hoved og fod. I ovenstående eksempel:

```
  integer :: argu
  interface
    subroutine SUBROUT (arg)
    real :: arg
    end subroutine SUBROUT
  end interface
  ...
  call SUBROUT (argu)
  ...
```

Hvis dette nu skal oversættes, giver oversætteren en fejlbesked, at SUBROUT bliver kaldt forkert, at der er uoverensstemmelse mellem de to typer. Det er mere normalt at lægge interfaceblokken i en module, hvor der givetvis ligger flere:

```
module INTERFACE_BLOK
  implicit none
  real :: arg, ...
  interface
    subroutine SUBROUT (arg)
    real :: arg
    end subroutine SUBROUT
    subroutine SUBROUT2 (arg)
    real :: arg
    end subroutine SUBROUT2
  end interface
end module INTERFACE_BLOK


  ...
  use INTERFACE_BLOK
  real :: argu
  ...
  call SUBROUT (argu)
  ...
```

- nu med `argu` erklæret korrekt.

Der er mange situationer, hvor brug af en sådan interface giver større sikkerhed mod fejl. Desuden er der faciliteter i Fortran, der forudsætter brugen af interfaces. Nogle af disse bliver nævnt lidt senere i dette kapitel.

Der er et par aspekter ved interfaces der skal nævnes. Det første af dem er, at når et program

benytter sig af en interface-module, hvori en eller flere functions er definerede, så må disse functions ikke erklæres for anden gang i programmet. Lad os sige, vi har functionen COT, erklæret som typen real, og kaldt fra et program. Normalt, hvis man bruger implicit none (som man burde), skal man så erklære COT:

```
program <noget>
  implicit none
  real :: COT
  ...
```

Men hvis vi nu beslutter at bruge en interface og lægger den i en module, ser det sådan ud:

```
module INTERF
  implicit none
  interface
    real function COT (arg)
      real :: arg
    end function COT
  end interface
end module INTERF

program <noget>
  use INTERF;     implicit none
  ...
```

Ved at bruge use-sætningen, og derved interfacen, har programmet allerede kendskab til typen af functionen COT og behøver ikke (*må ikke*) erklære functionen igen.

Det andet aspekt er lidt mere indviklet. Som beskrevet, er det muligt at definere sig parameters som står for visse KINDs (mest reals), og det er bedst at lægge dem i en module. Det kan være, at man så skriver nogle procedurer der modtager argumenter (eller, i tilfældet af functions, har resultat) af disse KINDs af typer. Vil man oprette interfaces til disse, kommer der en mærkelig regel i spil. Reglen er, at en interfaceblok er en lukket enhed, som ikke kender noget til den enhed, den er indeholdt i. Stadig med vores eksempel COT, men nu med en vis KIND af real, ville det være naturligt at have definitionen af KIND-parameteren samt interface til COT i samme module:

```
module ALT          ! **** Virker ikke !
  implicit none
  integer,parameter :: dbl &
                     = SELECTED_REAL_KIND(14)
  interface
    function COT (arg)
      real(kind=dbl) :: COT, arg
    end function COT
  end interface
end module ALT
```

Dette virker ikke. Årsagen er, at interfaceblokken ikke kender til parameteren dbl, og derfor ikke kan finde ud af denne KIND. Derfor er det nødvendigt at lægge de to definitioner i to forskellige moduler, og for COT at use den anden. Her er så det hele, som virker:

```
module PARAM
  implicit none
  integer,parameter :: dbl &
                       = SELECTED_REAL_KIND(14)
end module PARAM

module INTERF
  interface
    function COT (arg)
      use PARAM
      real(kind=dbl) :: COT, arg
    end function COT
  end interface
end module INTERF

program <noget>
  use PARAM; use INTERF;   implicit none
  real(kind=dbl) :: x
  ...
  x = ...
  ... = COT (x)
  ...
end
```

Det bemærkes, at der er en use PARAM-sætning i definitionen af COT i selve interfacen; og hvis der var flere procedurers interfaces der, skulle de alle have denne sætning; og at der i hovedprogrammet ikke længere er en sætning, der erklærer typen af COT, eftersom den er kendt fra sætningen use INTERF. Typen af variablen x skal dog erklæres, fordi den ikke er ens med skyggeargumentet arg i COT.

En tredje komplikation dukker op, når man lægger procedurer i en module. De er i så fald allerede anset som havende en explicit interface, så der må ikke skrives en til. En programenhed, der vil kalde disse procedurer, skal skaffe sig adgang med en use-sætning, og det vil i sig selv virke som interface til procedurerne.

8.6 OPTIONAL, brug af keywords

Det hænder, at en given procedure ikke altid skal have alle argumenter overført; de der sommetider er udeladt skal have attributtet OPTIONAL, hvilket betyder valgfrit. Denne facilitet er koblet naturligt sammen med en anden måde at kalde procedurer

på, ved at bruge **keywords** i kaldet. For at dette kan virke, skal der være en explicit interface. Her er et eksempel på en procedure der benytter faciliteten:

```
subroutine SUB (ko, el, em, en)
  integer, optional :: ko, el, em, en
  if (PRESENT(ko)) print '(" ko  =", i6)', ko
  if (PRESENT(el)) print '(" el  =", i6)', el
  if (PRESENT(em)) print '(" em  =", i6)', em
  if (PRESENT(en)) print '(" en  =", i6)', en
end subroutine SUB
```

Ordet `optional` kan lægges sammen med typen, eller også specificeres i en særskilt sætning. Proceduren benytter så den intrinsic function `PRESENT` for at finde ud af, om det pågældende argument er til stede ved kaldet, i hvilket tilfælde det bliver brugt. Proceduren kan kaldes fra en enhed der indeholder en interfaceblok (eller benytter en module med sådan en), og forskellige kald er mulige, for eksempel:

```
  ...
  integer :: k, l, m, n
  interface
    subroutine SUB (ko, el, em, en)
      integer, optional :: ko, el, em, en
    end subroutine SUB
  end interface
  ...
  call SUB (k, l, m, n)     ! Alle overfoert
  call SUB (k, l)           ! Kun k og l
  call SUB (k, l, en=n)     ! m udeladt
  call SUB (em=m)           ! Kun m overfoert
```

Det er tydeligt, at de kan overføres helt normalt, hvis de angives i den rigtige rækkefølge; hvis man derimod udelader noget, der efterlader et tomrum, skal de efterfølgende angives med keyword.

8.7 Overførsel af arrays til procedurer

Der er en række aspekter i at overføre arrays til en procedure, og de blev kun meget kort henvist til i kapitel 4. Her skal emnet udtømmes. Der er flere mulige måder at overføre arrays på. Her antages der, at læseren har brug for gode råd i stedet for bare en menu af alle muligheder, mange af dem ikke tidssvarende længere. Fortran slæber mange fossiler med sig, som ikke alle kan anbefales.

Man skal gøre sig klart, hvad der sker, når et array overføres til en procedure. Det der bliver overført, har visse egenskaber eller attributter, såsom type, antal dimensioner samt størrelsen i de forskellige dimensioner. Desuden kan der være nedre og øvre grænser på indekser for alle dimensioner, og – for at gøre tingene værre – forskel mellem de fysiske (erklærede) længder og de aktuelle. Den kaldende enhed har kendskab til alle disse; og proceduren har muligvis brug for også at kende (nogle af) egenskaberne. Det der bliver overført, er bare en adresse, dvs. adressen på det første element af et array (eller – se afsnit 4.4 – det første af en del af et array). Det kan i sig selv ikke give et udsagn om resten af attributterne. En procedure skal for eksempel operere på alle elementer i et array; for at kunne gøre dette er den nødt til at "vide", hvor mange der er i hver dimension, osv. Der er forskellige måder, hvorpå man kan overføre disse supplerende informationer til en procedure.

Præcis hvordan man takler alt dette, er ofte afhængigt af, hvad man programmerer. Der er tre situationer angående en procedure:

- Proceduren er en del af et éngangsprogram og forventes ikke at blive brugt i andre programmer;
- proceduren bliver muligvis genbrugt i andre programmer fremover;
- proceduren skal bruges i andre programmer, herunder programmer skrevet af andre.

Simpel arrayoverførsel

I det første tilfælde kan man tillade sig hvad man kalder "quick & dirty" programmering. Det medfører større risiko for fejl, men det kan være acceptabel hvis programmet er af mindre omfang og let overskueligt. Et eksempel kan være

```
program NOGET
  integer :: arr(-5:5,-10:10)
  ...
  call SUB (arr)
  ...
end program NOGET

subroutine SUB (arr)
  integer :: arr(-5:5,-10:10)
  ...
end subroutine SUB
```

Det binder proceduren `SUB` til hovedprogrammet, idet proceduren har indirekte kendskab til tal i det kaldende program (-5, 5, osv.); de er ikke overført som argumenter. Alt dette gør proceduren ufleksibel. Hvis man, for eksempel, på et senere tidspunkt fortryder arrayets attributter og vil ændre dem, er man også nødt til at ændre dem i

proceduren – og hvis der er et større antal af procedurer, i alle dem også. Men ved en lille opgave kan det være i orden.

En lidt bedre løsning er at overføre al fornøden information:

```
program NOGET
  integer :: arr(-5:5,-10:10)
  ...
  call SUB (arr, -5, 5, -10, 10)
  ...
end program NOGET

subroutine SUB (arr, lig1, oig1, lig2, oig2)
  integer :: lig1, oig1, lig2, oig2
  integer :: arr(lig1:oig1,lig2:oig2)
  ...
end subroutine SUB
```

hvor argumenterne `lig1` osv. står for nedre indeksgrænse 1, osv. Nu kan det lade sig gøre at kalde proceduren med forskelligt erklærede arrays, og sågar at bruge proceduren i andre programmer. Det virker dog lidt klodset, og der er stadig væk nogle faldgruber.

Hvis man er i gang med et "quick & dirty" program, kan man lette arbejdet yderligere ved at bruge en module med arrayet erklæret indeni. Det sparer megen overførsel:

```
module INDEKSER_OG_MERE
  implicit none
  integer, parameter :: lig1 = -5,  oig1 = 5
  integer, parameter :: lig2 = -10, oig2 = 10
  integer :: arr(-lig1:oig1,lig2:oig2)
end module INDEKSER_OG_MERE

program NOGET
  use INDEKSER_OG_MERE
  ...
  call SUB
  ...
end program NOGET

subroutine SUB
  use INDEKSER_OG_MERE
  ...
end subroutine SUB
```

Generelt, når der er tale om en procedure til mere udvidet brug, burde man programmere lidt mere stramt. Der er to forhold, der kan lette arbejdet: brug af arrayudsnit og erkendelsen af, at eksotiske indeksgrænser er mere eller mindre en behagelighed, men ikke strengt nødvendigt. Man kan altså skrive procedurer, der modtager arrays, muligvis erklæret med sådanne eksotiske indeksgrænser, men anser dem gående fra 1 til noget, eller givetvis (som ofte er logisk) fra 0 til noget. Placeringen er i samme rækkefølge som i det kaldende program. Brug af arrayudsnit gør, at de fysiske dimensioner og længder er de samme som de aktuelle, og så kan man spare sig at overføre dem allesammen. Eksemplet vil i så fald blive til

```
program NOGET
  integer :: arr(-5:5,-10:10)
  ... ! (indekser -n:n,-m:m... bestemmes)
  call SUB (arr(-n:n,-m:m), 2*n+1,2*m+1)
  ...
end program NOGET

subroutine SUB (arr, n, m)
  integer :: n, m
  integer :: arr(n,m)
  ...
end subroutine SUB
```

Hvis det drejer sig om en procedure man tilbyder andre, så følger der en vejledning med, som bl.a. specificerer at kaldet skal foregå på den måde.

Arrayoverførsel med explicit interface

Det ovenstående, som nævnt, tillader programmering der kan have faldgruber. Oversætteren checker ikke efter, hverken om dimensionerne eller typerne af et overført array er de samme i den kaldende enhed og proceduren, hvilket kan være farligt. Her er et eksempel på, hvad man må, selv om det ikke nødvendigvis er noget godt:

```
program ASSUME
  implicit none
  integer :: A(2,2)
  A = 1
  print *, " A, main:"
  print *, A
  call SUB (A,4)
end program ASSUME

subroutine SUB (arr,n)
  implicit none
  real,dimension(*) :: arr
  integer           :: i, n

  print *, " A, SUB :"
  do i = 1, n
    print *, Arr(i)
  enddo
end subroutine SUB
```

Hovedprogrammet erklærer arrayet `A` som todimensionalt og af typen `integer`, hvorimod proceduren `SUB` modtager arrayet som endimensionalt og af typen `real`. Det første er hvad der kaldes **assumed-size** overførsel og er helt legalt; det

brugtes i tidligere versioner af Fortran til at overføre "arbejdsplads", og kan nu gøres bedre (se senere, under hjælpearrays). Det andet, uenighed om typerne de to enheder imellem, er oftest en programmeringsfejl, og fører til nonsens. Problemet er blot, at oversætteren ikke brokker sig, siden det er syntaktisk helt korrekt. Den maskine, som forfatteren bruger, giver følgende output når dette program kører:

```
A, main:
        1           1           1           1
A, SUB :
1.4012985E-45
1.4012985E-45
1.4012985E-45
1.4012985E-45
```

og uoverensstemmelsen mellem typerne er tydelig.

Arbejdsplads (hjælpearrays), assumed-shape

Som nævnt ovenfor, er der ofte brug for hjælpearrays i en procedure, for at proceduren kan udføre sin opgave. Det kan være i form af et array, selv om ingen arrays er overført til proceduren; eller et array, hvis dimensioner og størrelser er relateret til et overført array.

Et eksempel opstår, når der skal findes den største egenværdi af en matrix der er fyldt i visse steder med nogle få parametre. Matricens størrelse kan være stor i forhold til antallet af forskellige parametre, men det man behøver at vide, er kun den største egenværdi. Den kan beregnes i en procedure, som modtager de få parametre, samt størrelsen af matricen. Matricen selv kan altså være en intern datastruktur i proceduren, det er kun dens størrelse, som er overført til proceduren. Man kan bruge en lokal matrix ved at benytte sig af `allocate` osv. Her er en skitse af sådan en procedure, hvor `p1...` er parametrene, `n` er størrelsen af den ønskede matrix ($n \times n$), og `maxeig` er den (komplekse) største egenværdi, beregnet af proceduren:

```
subroutine EGENVAERDI(p1, p2, ..., n, maxeig)
  implicit none

  integer, intent(in)  :: n
  real, intent(in)     :: p1, p2, ...
  complex, intent(out) :: maxeig

  integer              :: ...
  real, allocatable    :: mat(:,:)

  ...
  allocate (mat(n,n))
  ...        ! Beregn egenvaerdierne osv
  deallocate (mat)
end subroutine EGENVAERDI
```

Arrayet `mat` er oprettet på stedet, brugt og nedlagt igen. Den hukommelsesplads, det optager, er altså frigivet igen efter programmet kommer ud af proceduren.

Brug af assumed-shape og assumed-size

De to måder at overføre arrays til procedurer på er blevet nævnt i det ovenstående i forbigående. Her skal der gives nogle få råd om, hvornår de er brugbare. Faktisk ligner de hinanden til en vis grad. Man overfører et array til en procedure uden at specificere fuldt, hvad arrayets attributter er. I tilfældet **assumed-shape** kan man slippe afsted med udelukkende at overføre selve arraynavnet. I proceduren bliver arrayet modtaget udelukkende med dets antal dimensioner, dvs. dets form (eng.: **shape**). Eftersom man er tvunget til også at stille en explicit interface til rådighed, kan proceduren selv finde ud af længderne i de enkelte dimensioner; dog ikke de indeksgrænser arrayet er erklæret med i den kaldende enhed, hvilket sommetider ville være ønskeligt. Her er nogle elementer af hvad der er tale om her:

```
module STUFF
  interface
    subroutine SUB (arr)
      implicit none
      integer,dimension(:,:) :: arr
    end subroutine SUB
  end interface
end module STUFF

program ASSUME
  use STUFF;  implicit none
  integer :: A(-1:1,-2:2), i, j
  ...
  call SUB (A)
end program ASSUME

subroutine SUB (arr)
  implicit none
  integer,dimension(:,:) :: arr
  integer         :: lb1, ub1, lb2, ub2, n, m
  n = SIZE(arr,1);   m = SIZE(arr,2)
  print *, " Passed array:"
  print *, " L along dim 1 and 2:", n, m
  lb1 = LBOUND (arr, 1); ub1 = UBOUND (arr,1)
  lb2 = LBOUND (arr, 2); ub2 = UBOUND (arr,2)
  print *, " L & U bounds, dim 1:", lb1, ub1
  print *, " L & U bounds, dim 2:", lb2, ub2
end subroutine SUB
```

Resultatet af at køre dette er

```
  Passed array:
  L along dim 1 and 2:          3          5
  L & U bounds, dim 1:          1          3
  L & U bounds, dim 2:          1          5
```

Med andre ord: De erklærede indeksgrænser, `-1, 1, -2, 2`, er gået tabt. Faktisk kan man spørge sig, hvad de intrinsics `LBOUND` og `UBOUND` kan bruges til, hvis de kun giver de rigtige svar inden for den enhed, hvori det pågældende array er erklæret; man har næppe glemt, hvad man selv har programmeret. På den anden side kan man også anse sådan mere eller mindre eksotiske indeksgrænser som unødvendig luksus. I det ovenstående eksempel kan proceduren godt operere med det overførte array, ved at bruge indekserne `1:n` og `1:m`.

Sammenfattende kan man altså sige, at assumed-shape overførsel sparer at også skulle overføre længder; dog er man tvunget til at bruge en interface, hvilket kan virke lidt tungt, men anses for at være noget godt, da interfaces sikrer ordentlig overførsel af argumenter til procedurer og derved forhindrer visse fejl.

Der er imidlertid ikke så meget godt at sige om **assumed-size**. I de situationer, hvor man kan bruge det, kan man lige så godt klare sig uden; den er fossilsk. I eksemplet

```
  integer :: vec(100)
  ...
  call VECSUB (vec, 100)
  ...
subroutine VECSUB (v, n)
  integer :: n, V(*)
  ...
end subroutine VECSUB
```

var det nødvendigt at overføre arrayets længde (100); derfor kunne proceduren lige så godt have modtaget arrayet med erklæringen

```
subroutine VECSUB (v, n)
  integer :: n, V(n)
  ...
end subroutine VECSUB
```

Desuden er brugen af assumed size forbundet med nogle restriktioner: hvis `V` er erklæret som `V(*)`, er det ikke tilladt at nævne hele arrayet i helarrayudtryk, og heller ikke at nævne arrayudsnit af sådanne arrays.

Der er en anden anvendelse af assumed-size, selvom det i beskrivelsen af Fortran 90 ikke kaldes ved navnet: ved overførsel af tegnstrenge. En tegnstreng er formelt en skalar, men opfører sig på visse måder som et array. Den har en længde, for eksempel. Denne længde kan, i modsætning til arrays, overføres automatisk til en procedure, som det allerede blev vist i kapitel 4, afsnit 4.6, hvor proceduren `MATOUT` indeholder bl.a.

```
subroutine MATOUT (streng, ...)

  character(LEN=*) :: streng
  ...
  L = LEN_TRIM (streng)
```

som ligner assumed-size lidt. Formelt er det dog noget andet.

8.8 PUBLIC og PRIVATE

Indtil nu er modules blevet nævnt som en god måde at gøre mange ting tilgængelige for andre enheder på; enhederne deler derved de informationer i modulerne som de benytter ved `USE`. Det kan være parameters, datastrukturer (arrays m.m.), interfaces, og procedurer definerede under `CONTAINS`-afdelingen af en given module. Det er imidlertid tænkeligt, at man gerne vil beholde visse oplysninger til modulen, og altså ikke gøre dem offentlige til andre enheder, selv om modulen indeholder offentlige ting. Et typisk eksempel er variabler – eller functions – delt mellem procedurer interne til en modul, men som andre enheder ikke behøver at kende. Dette kan styres med sætningerne `PUBLIC` og `PRIVATE`. Default her er det første, men man kan udtrykkeligt slå det fast med sætningen

```
  PUBLIC
```

og udvælge visse variabler som det modsatte, som

```
  PRIVATE :: arr, n
```

for eksempel. Modsat kan alt blive privat og visse ting offentlige. Hvis en generisk function i en module er gjort privat, betyder det ikke automatisk, at alle dens instanser også er private, hvis ikke de er erklæret som sådan, hver for sig (se kapitel 9 for mere om generiske functions).

8.9 Array-valued functions

En function kan returnere arrays, ligesom elemental intrinsics, og er i så fald (på engelsk) kaldt **array-valued**, hvilket ikke kan oversættes ordentligt. Det kræver en interface. De eksempler, man træffer i lærebøgerne, er dog lidt overflødige, eftersom man kan lave de fleste af disse med helarrayoperationer. Vi giver et eksempel her, som ikke er særlig spændende, men i hvert fald ikke kan lade sig gøre med en enkelt operation. Vi vil få et array, som indeholder de samme elementer som et bestående array, men i omvendt rækkefølge. Interfacen til proceduren er:

```
  interface
    function VEND (arr)
      implicit none
      integer,intent(in) :: arr(:)
      integer            :: VEND(SIZE(arr))
    end function VEND
  end interface
```

og selve functionen er:

```
function VEND (arr)
  implicit none
  integer,intent(in) :: arr(:)
  integer            :: VEND(SIZE(arr))
  integer :: n, i
  n = SIZE(arr)
  do i = 1, n
    VEND(n-i+1) = arr(i)
  enddo
end function VEND
```

Det kan her lade sig gøre at bruge `assumed-shape` overførsel for argumentet, og at bruge dens `SIZE` til at erklære functionens arraystørrelse, fordi der er en interface til stede.

Man kan også skrive en array-valued function, som ikke modtager et array som argument. Et eksempel kan være en function, der returnerer enhedsmatricen af størrelsen $n \times n$, med blot `n` som argument:

```
function UNIT (n)
  implicit none
  integer,intent(in) :: n
  integer            :: UNIT(n,n)
  integer :: i
  UNIT(1:n,1:n) = 0
  do i = 1, n;  UNIT(i,i) = 1;  enddo
end function UNIT
```

Igen skal der være en interface til den. Det todimensionale array, som functionen skal have som værdi, bliver skabt på stedet. Når functionen kaldes, skal der være et array, der modtager værdien, for eksempel

```
  integer :: A(5,5) ...
  ...
  A = UNIT(5)
```

8.10 Character-valued functions

Analog med array-valued functions kan man lave en function, hvis værdi består af en tegnstreng. Det ligner array–valued functions til en vis grad. Et enkelt eksempel illustrerer, hvordan man gør. Følgende function laver en tegnstreng om til en, hvor alle små bogstaver er erstattet med store:

```
function STORE_BOGSTAVER (str)
  implicit none
  character(len=*),intent(in) :: str
  character(len=LEN(str))  :: STORE_BOGSTAVER

  character(len=1) :: tegn
  integer          :: L, i, afstand

  L = LEN(str); afstand=ICHAR("A")-ICHAR("a")
  do i = 1, L
    tegn = str(i:i)
    if (tegn>="a" .and. tegn<="z") then
      STORE_BOGSTAVER(i:i) = &
                 CHAR (ICHAR(tegn) + afstand)
    else
      STORE_BOGSTAVER(i:i) = tegn
    endif
  enddo
end function STORE_BOGSTAVER
```

Functionen bliver en tegnstreng med samme længde som argumentet. Hvis functionen bliver kaldt med, for eksempel,

```
  print '(a)', STORE_BOGSTAVER("hallo hallo")
```

så er resultatet

```
HALLO HALLO
```

8.11 Dummyprocedurer

Bortset fra variabler (eller konstanter) kan man også overføre procedurer til procedurer. Der får de "dummy"-navne, ligesom variabler osv. gør. Denne facilitet er nok mest brugt med functions. Proceduren, der modtager en procedure, skal erklære den på sådan en måde, at det er klart, at den **er** en dummyprocedure; og det skal den kaldende enhed også. Den modtagende enhed, hvortil en procedure bliver overført, bruger `EXTERNAL`-sætningen til formålet; mens den kaldende skal erklære det der overføres enten som `EXTERNAL` – hvis det er en ekstern procedure – eller `INTRINSIC` – hvis det er en af de intrinsic functions. Dette bliver nyttigt, når man for eksempel vil skrive en procedure, der kan minimere en given function, eller integrere den. Her er et eksempel på en integreringsprocedure, der bliver kaldt med to forskellige functions der skal integreres, den ene en ekstern, den anden en intrinsic:

```
program DUMMYPROC
  implicit none
  intrinsic    SIN
  external     MYFUNC
  integer   :: n
  real      :: MYFUNC, a, b, INTEG, pi
```

```
  pi = ACOS (-1.0)
  a = 0;  b = pi/2;  n = 100
  print '(" Integ. of SIN from 0..pi/2   =",&
        &      f10.3)', INTEG(SIN, a, b, n)
  a = -1;  b = 1
  print '(" Integ. of MYFUNC from -1..+1 =",
               f10.3)', INTEG(MYFUNC, a,b,n)
end program DUMMYPROC

real function INTEG (FN, a, b, n)
! To integrate the passed function FN between
! the limits [a,b], using the trapezoidal
! rule and n intervals.

  implicit none

  real, external      :: FN
  integer, intent(in) :: n
  real, intent(in)    :: a, b

  integer             :: i
  real                :: x, delta_x, sum

  delta_x = (b - a) / n
  sum = (FN(a) + FN(b)) / 2
  do i = 1, n-1
    x = a + i*delta_x
    sum = sum + FN(x)
  enddo
  INTEG = sum * delta_x
end function INTEG

real function MYFUNC (x)
  implicit none
  real, intent(in) :: x

  MYFUNC = EXP(x) + EXP(-x)
end function MYFUNC
```

Hovedprogrammet kalder integrationsproceduren to gange; første gang med den intrinsic function `SIN`, erklæret som `intrinsic`, og anden gang med den eksterne function `MYFUNC`, erklæret som `external`. I selve integrationsproceduren `INTEG` (en function), hedder alle functions, der er overført der, `FN`, og dette symbol er erklæret som `external`. Programmet giver i øvrigt de rigtige resultater:

```
 Integ. of SIN from 0..pi/2   =     1.000
 Integ. of MYFUNC from -1..+1 =     4.701
```

Man kunne stramme yderligere op på programmet ved at inkludere en interface til integrationsproceduren, eller en module med en sådan interface. Men eksemplet viser de vigtige træk i overførsel af procedurer til andre procedurer.

8.12 Rekursion

Rekursion betyder at en procedure kalder – direkte eller indirekte – sig selv. (Et indirekte kald af sig selv er givet, når en procedure A kalder en anden B, som kalder ..., som kalder A.) At kalde sig selv kan umiddelbart lyde mærkeligt, men tillader meget simpel algoritmedesign i visse programmeringssituationer. De fleste lærebøger forklarer rekursion med hjælp af et eksempel, hvor rekursion faktisk ikke er fordelagtig, eftersom man normalt ville programmere problemet uden rekursion. Det drejer sig om fakultetsfunktionen $n!$. Den er meget let at programmere med en `do`-løkke:

```
integer function FAKULTET (n)
  implicit none
  integer :: n
  integer :: i, prod

  if (n == 0) then
    FAKULTET = 1
  else
    prod = 1
    do i = 1, n
      prod = prod * i
    enddo
    FAKULTET = prod
  endif
end function FAKULTET
```

hvor vi har taget højde for, at, per definition, $0! = 1$. Som sagt, nævner lærebøger, at fakultetsfunktionen $n!$ kan omdefineres rekursivt som $n \times (n-1)!$, og $(n-1)!$ videre som $(n-1) \times (n-2)!$ osv., ned til $0!$. Det kan implementeres med en rekursiv function:

```
recursive integer function RECFAK (n) &
                           RESULT (value)
  implicit none
  integer :: n

  if (n == 0) then
    value = 1
  else
    value = n * RECFAK(n-1)
  endif

end function RECFAK
```

Bemærk, at functionen kalder (eller refererer til) sig selv. Det fører til en kæde af kald, indtil argumentet i kaldet (`n-1`) er blevet lig med nul, hvor functionen ikke længere kalder sig selv, men giver et egentligt resultat (1). Herefter følger en baglæns kæde af tilbagevenden i kæden af kald, og en multiplikation med alle tal gemt undervejs.

Der er en god grund til, at RESULT blev brugt i eksemplet. Selvom nogle oversættere nok ville tillade at man ikke bruger RESULT, hvor altså de to relevante sætninger ville være

```
if (n == 0) then
  RECFAK = 1                                !***
else
  RECFAK = n * RECFAK(n-1)                  !***
endif
```

- så er det imod reglerne. Grunden er, at enhver reference til functionens navn er et potentielt kald af functionen, selv om det måske er på den venstre side af en tilordningssætning. Sådan står det i reglerne. I praksis er denne regel håndhævet lidt mere tolerant, og RESULT er kun påkrævet hvis den rekursive function er array-valued, i hvilket tilfælde der ville stå, på den venstre side, noget i retning af

```
RECFUNC(i) = ...
```

og det ville enhver oversætter have svært ved at skelne fra et functionskald. Bundlinjen her er: brug RESULT med rekursive functions.

Functionen RECFAK er en anelse kortere end den første udgave, hvor vi brugte en løkke; og det er karakteristisk for rekursive procedurer, at de virker meget kompakte og overskuelige. En ting skal der altid være: en udvej, der standser kæden af selvkald; i dette eksempel når argumentet (n) er lig med nul.

Sådan en procedure *s*kal have en interface i den kaldende enhed, (eller i en module).

```
implicit none
interface
  recursive integer function RECFAK (n) &
                             RESULT (value)
    integer :: n
  end function RECFAK
end interface
```

Det ville være bedst at lægge interfacen i en module. Om der er en module, eller om interfacen ligger i den kaldende enhed, skal functionen ikke erklæres yderligere i enheden, eftersom aktivering af interfacen allerede har etableret attributterne.

Det bliver ofte sagt, at en rekursiv procedure er mindre effektiv end en tilsvarende ikke rekursiv (den hedder **iterativ**); dvs. den tager længere tid at udføre. Grunden er, at der selvfølgelig foregår en hel del bag kulisserne, når rekursion udføres. Kort sagt, skal en del variabelværdier gemmes væk (i en stak), for at blive hentet igen senere. I eksemplet med fakultetsfunktionen er det alle n-værdier. I mere indviklede procedurer kan der være en hel del der skal stakkes. Derfor er der opskrifter på at konvertere fra en rekursiv algoritme til en iterativ. Det er nemlig ofte meget nemt at formulere en algoritme rekursivt; et eksempel gives lige herefter. Men der er gode argumenter for rekursion. Regnemaskiner er meget hurtige om stunder, og det er vigtigere at skrive et klart program, der let kan verificeres, end et effektivt program, der udføres i den korteste tid. Ofte drejer det sig kun om sekunders forskel, og det kan være lige meget.

Et mere forsvarligt eksempel hentes fra sorteringsproblemet. Sortering er et klassisk problem inden for programmering, og man søger konstant efter bedre algoritmer. Vil man sortere mange poster (f.eks. 1 million), så kan det betale sig at anvende en effektiv sorteringsalgoritme. Det er blevet teoretisk fastlagt, at det bedste man kan opnå ved sortering af n poster, er en algoritme der udfører $nlog_2n$ sammenligninger. De mest simple algoritmer – de, man umiddelbart vil finde på – bruger derimod n^2, hvilket kan være betydeligt flere (hvis der er f.eks. 1000 poster, drejer det sig om henholdsvis 10^4 og 10^6). En af de bedre algoritmer for sortering, eller en variant af den, er at dele opgaven i to. Algoritmen er altså:

1. del de n poster i to ca. lige lange dele;
2. sorter de to dele særskilt;
3. flet dem sammen i den rigtige sorteringsorden.

Denne algoritme skubber problemet foran sig selv – typisk for rekursion: den beskriver ikke, hvordan den egentlige sortering, nu af hver enkelt af de to halvdele, skal foregå. Men det er i orden, fordi samme procedure gælder de to; samme algoritme anvendes for dem. Det betyder, at algoritmen bliver ved med at dele i to, indtil rækken af poster, der skal sorteres, er nede ved en enkelt, hvor der ikke er noget at sortere mere; nu kan den "række" flettes sammen med den anden halvdel, der også er blevet sorteret. Det viser sig, at hele arbejdet bliver gjort af fletningsalgoritmen. Her er en procedure for sortering af n heltal, der benytter denne tankegang (sammen med en fletningsprocedure):

```
recursive subroutine REC_SORT (X, n)
! Sorterer X i stigende raekkefoelge, ved at
! dele den i to halvdele, sortere disse og at
! lade FLET blande dem sammen.

  implicit none

  integer, intent(in)    :: n
  integer, intent(inout) :: X(n)
```

```
  integer :: idel

  If (n > 1) then
   idel = n / 2    ! Del i ca. lige halvdele
   call REC_SORT(X(1:idel),idel) ! Sorter dem
   call REC_SORT(X(idel+1:n),n-idel)  ! ...
   call FLET (X, n, idel)   ! Flet dem sammen
  endif

end subroutine REC_SORT

subroutine FLET (X, n, skel)
! Fletter de to halvdele af X, delt ved SKEL,
! ind i hinanden, i sorteringsorden.

  implicit none

  integer,intent(in)    :: n, skel
  integer,intent(inout) :: X(n)

  integer               :: i1, i2, i, j
  integer,allocatable   :: Y(:)

  allocate (Y(n))
  i1 = 1;  i2 = skel+1  ! Indekser til delene
  do i = 1, n
   if (X(i1) < X(i2)) then ! Vaelg den mindre
     Y(i) = X(i1);  i1 = i1 + 1
     if (i1 > skel) then ! Er 1.del brugt op?
       do j = i+1, n     ! I saa fald, brug
                         ! .. resten af 2.del.
         Y(j) = X(i2);  i2 = i2 + 1
       enddo
       exit
     endif
    else
     Y(i) = X(i2);  i2 = i2 + 1
     if (i2 > n) then    ! Er 2.del brugt op?
       do j = i+1, n     ! Brug resten af 1.
         Y(j) = X(i1);  i1 = i1 + 1
       enddo
       exit
     endif
   endif
  enddo
  X(1:n) = Y(1:n)       ! Y overskriver nu X.
  deallocate (Y)
end subroutine FLET
```

Subroutinen `REC_SORT` er faktisk nem at gennemskue, hvilket er typisk for rekursive procedurer. Den gør ikke noget, hvis antallet af tal, der skal sorteres, er lig med 1; ellers deler den og kalder sig selv og, til sidst, proceduren `FLET`. Denne procedure er lidt mere indviklet. Den tager fra den ene (sorterede) halvdel af arrayet X, eller den anden, alt efter sorteringsorden, og den skal sikre, hvis den ene eller anden halvdel er brugt op, at alt går godt derefter (dvs. den i så fald bare bruger resten af den halvdel, der er tilbage). Bemærk, forresten, brug af det allokerbare array `Y`, som modtager de sammenflettede `X`-værdier. De bliver til sidst kopieret tilbage i `X`, hvorefter `Y` bliver nedlagt igen.

Hvis man vil programmere denne del-og-sorter-algoritme uden rekursion, kommer man sikkert hurtigt i vanskeligheder. Rekursion eliminerer dem, og opgaven består kun af fletningsalgoritmen, en nemmere programmeringsopgave.

Også denne procedure skal have en interface i den kaldende enhed. I dette eksempel var den lagt i en module:

```
module DEF
  implicit none
  ...
  interface
    recursive subroutine REC_SORT (X, n)
      integer, intent(in)    :: n
      integer, intent(inout) :: X(n)
    end subroutine REC_SORT
  end interface
end module DEF
```

Sammenfattende kan det siges, at det i visse situationer er lettest at formulere en rekursiv algoritme, og det resulterende program (eller underprogram) bliver som regel let at overskue. Det er muligt at det medfører en mindre effektivitet, sammenlignet med en iterativ algoritme. Skulle det være tilfældet, og skulle effektiviteten være af overordentlig vigtighed, kan man altid omskrive en rekursiv procedure til en iterativ ved hjælp af visse regler. Disse går ud over denne bogs ramme, men kan findes i et antal af lærebøger med ordene “data structures” i deres titler.

Kapitel 9

Generiske procedurer og definerede operatorer

Det engelske fagudtryk **overloading**, anvendt i sammenhæng med Fortran, betyder noget i retning af **udvidelse**. Det hentyder til, at en procedures aktionsradius, eller en operator, er udvidet til mere end er normalt. Der er to slags overloading: **generiske procedurer**, og **definerede operatorer**.

9.1 Generiske procedurer

Ret mange intrinsic functions er **generic** (engelsk for generisk), hvilket betyder, at de er en slags overskrifter over en række individuelle functions. For eksempel dækker den generiske function `SQRT` over selve `SQRT` (for enkeltpræcision), `DSQRT` (dobbeltpræcision) og `CSQRT` (enkeltpræcision-kompleks). Hvilken af de underliggende der bliver anvendt, er afhængigt af, hvilken type argumentet (eller argumenterne) har.

Denne facilitet er ikke forbeholdt de indbyggede procedurer; man kan selv skrive generiske procedurer. Det gøres ved hjælp af navngivne interfaces. Et eksempel vil gøre det klart. Opgaven er at bytte to arrays, som kan (begge samtidigt) være af to forskellige `KIND`s inden for typen `real`, dvs. de to: mindst 6 decimalcifre, og mindst 14. Her er et testprogram der skaber en generisk subroutine til formålet:

```
module PARAMS
! Supplerer nogle navngivne konstanter
! (parameters), bl.a. nogle der staar for
! KINDs, SGL og DBL.
  implicit none
  integer,parameter :: &
              sgl = SELECTED_REAL_KIND(6),  &
              dbl = SELECTED_REAL_KIND(14), &
              maxdim = 5
end module PARAMS
module GENERISK
! Etablerer den generiske procedure
! (interface) ARRAYBYT, som daekker over de
! aktuelle procedurer SGL_BYT og DBL_BYT.
  use PARAMS;  implicit none
  interface ARRAYBYT
    module procedure SGL_BYT
    module procedure DBL_BYT
  end interface

  contains
  subroutine SGL_BYT (A, B)
   use PARAMS;  implicit none
   real(kind=sgl),dimension(maxdim) :: A, B
   real(kind=sgl),allocatable       :: gem(:)

   allocate (gem(maxdim))
   gem = A;  A = B;  B = gem
   deallocate (gem)
  end subroutine SGL_BYT

  subroutine DBL_BYT (A, B)
   use PARAMS;  implicit none
   real(kind=dbl),dimension(maxdim) :: A, B
   real(kind=dbl),allocatable       :: gem(:)

   allocate (gem(maxdim))
   gem = A;  A = B;  B = gem
   deallocate (gem)
  end subroutine DBL_BYT
end module GENERISK

program BYT_DEM
! Demoprogram, viser brug af generiske procs.
  use PARAMS;  use GENERISK;  implicit none
  real(kind=sgl),dimension(maxdim) :: S1, S2
  real(kind=dbl),dimension(maxdim) :: D1, D2

  S1 = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)  ! arrs,
  S2 = (/ 6.0, 7.0, 8.0, 9.0, 10.0 /) ! skal
  D1 = S1;  D2 = S2               ! byttes.
  call ARRAYBYT (S1, S2)
  call ARRAYBYT (D1, D2)
  print '(" S1 efter bytning:", 10f5.1)', S1
  print '(" S2 efter bytning:", 10f5.1)', S2
  print *
  print '(" D1 efter bytning:", 10f5.1)', D1
  print '(" D2 efter bytning:", 10f5.1)', D2
end program BYT_DEM
```

Her kaldes der ARRBYT, som egentligt er en interface; den delegerer opgaven videre til den procedure, der kan modtage argumenterne, dvs. med de `KIND`s der svarer til dem, proceduren er kaldt

med. En udskrift af dette program giver:

```
 S1 efter bytning:  6.0  7.0  8.0  9.0 10.0
 S2 efter bytning:  1.0  2.0  3.0  4.0  5.0

 D1 efter bytning:  6.0  7.0  8.0  9.0 10.0
 D2 efter bytning:  1.0  2.0  3.0  4.0  5.0
```

hvilket viser, at programmet, og metoden, virker.

Man må også godt kalde de bagliggende procedurer direkte; men i så fald skal man passe på at overføre argumenter af den rigtige type.

Der er en ting, der falder i øjnene i det ovenstående eksempel: det såkaldte interface i modulen `GENERISK` indeholder ikke mere end de to sætninger

```
module procedure SGL_BYT
module procedure DBL_BYT
```

hvor man muligvis ville forvente rigtige interfaces til procedurerne. Det skyldes reglen om, at man ikke må skrive en interface for en procedure, der er `contain`'ed i en module, eftersom det i sig selv fungerer som interface. Men her er der absolut brug for en interface for at kunne have den generiske mekanisme. Derfor henvises der i selve interfacen til de to procedurer på den måde, som bare indikerer, at procedurerne ligger et andet sted i samme module.

9.2 Definerede operatorer

I kapitel 2 er der beskrevet en række operatorer til at operere med de forskellige grundtyper. Der er **unitære** operatorer (f.eks. `-` eller `+`, som i `+x`), og **binære** operatorer, (f.eks. `*`, som i `a*b`). Plus- eller minustegnet kan være både unitært og binært. Der er desuden de logiske operatorer.

Især når man har med strukturer at gøre (se næste kapitel), kan der være behov for yderligere operatorer, hvis funktion programmøren selv definerer. Her er vi nødt til at holde os ved grundtyperne, men også her kan der være grund til at ville definere sig nye operatorer. Fremgangsmåden er at vælge et tegn eller en tegnstreng, der skal repræsentere den nye operator, og så at definere dens funktion i en interface; det er bedst at gøre alt i en module. Valget af tegnene er dog begrænset: man skal enten bruge en eksisterende operator (`+`, `-`, `*`, `/`, `**`, `//`, `.and.`, osv.), eller lave en, der består af 1-31 tegn i en streng, indrammet af punktummer: .<*str*>. Desuden skal man bruge et i forvejen unitært operatortegn for at betegne en ny unitær operator.

På denne måde udvider man altså funktionen af en given operator (hvis man vælger en der findes i forvejen), og det kaldes **overloading**. Bemærk, at operatorens oprindelige betydning ikke går tabt. I det følgende eksempel bliver minustegnet overloaded til også at stå for den logiske operation `.not.`; det forhindrer ikke tegnets brug i den vante sammenhæng.

Et ret simpelt eksempel viser vejen. Vi har at gøre med et program, hvori der skal opereres meget med typen `logical`, og vi er trætte af at indtaste operatorerne `.not.`, `.and.` og `.or.`. De skal erstattes med ny operatortegn, henholdsvis `-`, `*` og `+`. Læg mærke til, at den unitære operator `-` er (eller kan være) unitær i forvejen i sin uudvidede eksistens, så den kan bruges til at stå for den unitære operator `.not.` Man skriver en module, der bruger interne functions, der ved hjælp af nogle interfaces bliver generiske, med operatorerne som overflade. Her er modulen:

```
module DEFS
  interface operator (-)
    module procedure NOT
  end interface
  interface operator (*)
    module procedure AND
  end interface
  interface operator (+)
    module procedure OR
  end interface
  contains
    logical function NOT (logvar) &
                     RESULT (NEGATE)
      logical, intent(in) :: logvar
      NEGATE = .not. logvar
    end function NOT
    logical function AND (logvar1, logvar2) &
                     RESULT (BOTH)
      logical, intent(in) :: logvar1, logvar2
      BOTH = logvar1 .and. logvar2
    end function AND
    logical function OR (logvar1, logvar2) &
                     RESULT (EITHER_OR)
      logical, intent(in) :: logvar1, logvar2
      EITHER_OR = logvar1 .or. logvar2
    end function OR
end module DEFS
```

Modulen indeholder nogle functions (`NOT`, osv.), som også kan kaldes direkte, men de kan alle kaldes ved at bruge de for dem stående operatorer, definerede ved de viste interfaces. Bemærk, at disse interfaces alle har keywords `module procedure` <...>; det er påkrævet, sammen med at argumenterne til de interne functions alle skal være erklærede med `intent(in)`. Her er et lille testprogram, der afprøver alt dette. Det benytter to logiske variabler, med værdierne `.true.` og `.false.`

Programmet benytter både de nye operatorer og direkte kald af functionerne.

```
program DEFOP
! Test af logiske definerede operatorer:
! - som skal staa for .not. (en unitaer op)
! *  "   "     "   "  .and.
! +  "   "     "   "  .or.

  use DEFS;   implicit none
  logical :: t=.true., f=.false.

  print '(" - .true.         =", L10)', - t
  print '(" .true. * .false. =", L10)', t * f
  print '(" .true. + .false. =", L10)', t + f
  print '(/" Direkte kald af module procs:")'
  print '(" NOT(.true.)          =", L10)', &
                                    NOT(t)
  print '(" AND(.true., .false.) =", L10)', &
                                    AND(t, f)
  print '(" OR (.true., .false.) =", L10)', &
                                    OR(t, f)
end program DEFOP
```

Resultatet af en kørsel af programmet er:

```
 - .true.         =         F
 .true. * .false. =         F
 .true. + .false. =         T

 Direkte kald af module procs:
 NOT(.true.)          =         F
 AND(.true., .false.) =         F
 OR (.true., .false.) =         T
```

og resultaterne af de to forskellige slags kald er identiske.

Der er yderligere en operatorudvidelse, der bruger lighedstegnet, men det skal vente indtil strukturer er kendt, hvilket de bliver i kapitel 10.

Kapitel 10
Derived types og strukturer

10.1 Indledning

Som beskrevet i de indledende kapitler, er der fire grundtyper i Fortran: `integer`, `real`, `logical` og `character`, samt kombinationer deraf såsom `complex` og arrays af disse. Arrays er samlinger af skalarer og har den egenskab, at alle arrayelementer er af samme type. Det hænder imidlertid, at man gerne vil samle forskellige typer i klynger under et navn, og dertil er der de såkaldte **derived types**, som kan oversættes til **samletyper** (men vi holder os mest til **derived types**). Variabler erklæret som værende disse typer kaldes **strukturer**.

En struktur bliver erklæret med en `type`-sætning. Det følgende eksempel illustrerer fremgangsmåden. Vi vil etablere en oversigt (database) over vores privatbibliotek, som er blevet uoverskueligt stort. Vi har nummereret alle hylderne. Nu opretter vi typen `bog`, der samler den slags information, man måske vil have om sådan en:

```
type bog
  character(len=40)  :: forfatterfornavn
  character(len=40)  :: forfatterefternavn
  character(len=100) :: titel
  integer            :: pub_aar
  character(len=40)  :: forlag
  integer            :: sider
  integer            :: hyldenummer
endtype bog
```

Der samles tegnstrenge af forskellige længder og nogle heltal. Har man denne erklæring af typen, kan man så erklære variabler med den type, altså som strukturer. En enkelt af dem, lade os kalde den `MUNK`, erklæres således med sætningen

```
  type(bog) :: munk
```

og så kan man operere med denne variabel i programmet. Hver gang der bliver refereret til navnet `munk`, refereres der til hele bunken af dens komponentdele. Man kan også referere til disse; det foregår ved at nævne *<variabelnavn>*%*<komponent>*, for eksempel `munk%forfatterfornavn` eller `munk%sider` osv. Eksempler på sådanne referencer er:

```
munk%forfatterfornavn = "Thorkild"
print *, " Antal sider =", munk%sider
```

hvor komponentdelene bliver anvendt forskelligt.

Nu er det ikke særlig smart at oprette individuelle variabler for alle ens bøger; det er bedre at bruge et array. Så bliver erklæringen til

```
  type(bog),dimension(1000) :: boeger
```

og hvis man nu ville have fat i en bestemt bogs forfatterefternavn, for eksempel bog nr. 17, så refererer man til `boeger(17)%forfatterefternavn`. Her ser man, at man kan have arrays af en derived type. Modsat er det også muligt at have et array som komponent af en derived type. Man kunne tænke sig at lægge de to navne sammen under et array:

```
type bog
  character(len=40)  :: forfatternavn(2)
  character(len=100) :: titel
  integer            :: pub_aar
  character(len=40)  :: forlag
  integer            :: sider
  integer            :: hyldenummer
endtype bog
```

hvor der dannes et lille array med to elementer.

10.2 Input-Output

Når der er tale om de enkelte komponenter af en struktur, er der ikke forskel fra normale variabler – de *er* normale typer (medmindre man har en struktur som komponent af en anden). De kan altså indlæses og udskrives ganske normalt som vist ovenfor, hvor der er en `print`-sætning. Eneste forskel er, at hvis man refererer til hele strukturen, menes der alle komponenter, i den erklærede rækkefølge. Hvis man for eksempel programmerer

```
print *, munk
```

så får man en udskrift af hele indholdet på skærmen i en form bestemt af systemet. Men man kan selv bestemme formatet efter komponenterne.

På samme måde kan strukturer indlæses som data. Her er det bedst, som næsten altid, at bruge frit format og passe på at have anførselstegnene med. Inputdata for variablen `munk`, læst ind med sætningen

```
read *, munk
```

ville for eksempel se ud som

```
"Thorkild", "Hansen"
"Jens Munk"
1982
"Gyldendal"
498
5
```

10.3 SEQUENCE

Specifikationen `sequence` burde nævnes, selvom det er tvivlsomt, hvor nyttig den er, når man bruger modules. Hvis man lægger den til, f.eks.

```
type bog
  sequence
  character(len=40)  :: forfatternavn(2)
  character(len=100) :: titel
  integer            :: pub_aar
  character(len=40)  :: forlag
  integer            :: sider
  integer            :: hyldenummer
endtype bog
```

så beordrer man oversætteren til at anordne de enkelte komponenter af typen i den nævnte rækkefølge. Det er ellers ikke garanteret; i princippet kan systemet vælge en anden orden, der måske giver en mere effektiv pakning af komponenterne. Hvis det sker, så er der tilmed ingen garanti for, at systemet vil vælge den samme pakning i forskellige enheder, hvor typen er erklæret. Kalder en enhed så en anden og overfører sådan en type, kan den være inkompatibel med definitionen i proceduren – meget usandsynligt, men efter sigende muligt. Alt dette ungås ved at forlange `sequence`. Det undgås også ved at bruge en module til at erklære typen i.

10.4 Type constructors

Ligesom der findes konstanter for alle andre typer og for arrays, findes der konstanter for strukturer. De kaldes på engelsk **type constructors**, og det er temmelig håbløst at prøve at oversætte denne term. En type constructor består af typenavnet og indholdet i form af de enkelte konstanter, alt i parentes. Vores eksempel kunne således være

```
munk = bog("Thorkild","Hansen","Jens Munk", &
                1982, "Gyldendal", 498, 5)
```

hvor alt der er på den højre side af "=" er en type constructor, med de enkelte komponenter efter hinanden.

10.5 Strukturer og procedurer

Når der skal skrives procedurer (subroutines eller functions) der opererer med strukturer, bliver det klart, at det er bedst at bruge modules. Vi går nu over til en lidt mere håndterlig struktur, der skal bruges i det følgende. Vi lader som om typen `complex` ikke findes, og vi skal lave vores egen; den skal hedde `komplextal`. Der skal oprettes en række procedurer til at operere med den type, og vi begynder med kun en, der skriver værdierne ud sammen med en tekst. Den mest simple "quick & dirty" måde at gøre det på er følgende:

```
program DERIVED_TYPES
! Afproever derived type KOMPLEXTAL.
  implicit none

  type komplextal
    real :: rdel, idel
  end type komplextal
  type (komplextal) :: a
  print*," Indtast et komplex tal (2 reals):"
  read *, a
  call PRINT_KOMPLEXTAL ("a:", a)
end program DERIVED_TYPES

subroutine PRINT_KOMPLEXTAL (str, z)
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
  character(len=*),intent(in)  :: str
  type (komplextal),intent(in) :: z

  character(len=20) :: localstr
  integer           :: L

  L = LEN_TRIM(str); localstr =  " "
  localstr(1:L) = str(1:L)
  print '(1x,a20,f8.4," +",f8.4," * i")', &
          localstr, z
end subroutine PRINT_KOMPLEXTAL
```

For at proceduren kan modtage strukturen `z` ordentligt, skal den selv have typeerklæringen, og den skal være præcis den samme som den i hovedprogrammet. Ligesom med arraydimensionerne, er dette en dårlig ide. Hvis man vil ændre komponenterne af typen, skal man pille ved begge definitioner, på præcis samme måde.

Læseren kan se bort fra den måske lidt kryptiske måde, den indkomne tegnstreng bliver kopieret i den lokale `localstr` osv.; det var et lille trick for at sikre det ønskede format, og er afhængigt af, at `str` ikke bliver printet ud med flere end 20 tegn.

Brug af en module

Hvis typedefinitionen bliver lagt i en module, løses det ovennævnte problem med de to erklæringer:

```
module COMPLEXSTUFF
! Module til at definere den hjemmelavede
! type, komplextal.
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
end module COMPLEXSTUFF

program DERIVED_TYPES
! Afproever derived type KOMPLEXTAL.
  use COMPLEXSTUFF;    implicit none
  type (komplextal) :: a

  print*," Indtast et komplex tal (2 reals):"
  read *, a
  call PRINT_KOMPLEXTAL ("a:", a)
end program DERIVED_TYPES

subroutine PRINT_KOMPLEXTAL (str, z)
  use COMPLEXSTUFF;    implicit none
  character(len=*),intent(in)  :: str
  type (komplextal),intent(in) :: z

  character(len=20) :: localstr
  integer           :: L

  L = LEN_TRIM(str); localstr =  " "
  localstr(1:L) = str(1:L)
  print '(1x,a20,f8.4," +",f8.4," * i")', &
          localstr, z
end subroutine PRINT_KOMPLEXTAL
```

Nu kan man nøjes med at pille ved typen det ene sted, hvis man vil ændre typedefinitionen, og der er mindre risiko for fejl.

Brug af interface

Vi har lært, at det er godt at bruge interfaces til procedurer for at sikre sig imod problemer ved kaldene; så nu lægger vi en interface til programmet. Her løber vi ind i et lille problem. Det logiske ville være at lægge denne interface i samme module som typedefinitionen for `komplextal` er lagt ind i. Modulen ville se sådan ud:

```
module COMPLEXSTUFF ! Virker ikke! *********
! Module til at definere den hjemmelavede
! type, komplextal.
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
  interface
    subroutine PRINT_KOMPLEXTAL (str, z)
      character(len=*),intent(in)  :: str
      type (komplextal),intent(in) :: z   !**
    end subroutine PRINT_KOMPLEXTAL
  end interface
end module COMPLEXSTUFF
```

Den går ikke! Oversætteren brokker sig om, at i den mærkede linje ("!**") er typen `komplextal` ukendt. Grunden til denne besked, som umiddelbart virker som løgn, er, at en interface udgør sin egen lille lukkede enhed, og den kender ikke til omverdenen, ikke engang den nærmeste, indholdet af resten af modulen, herunder definitionen af den omtalte type (det ville på samme måde gælde for eksempel `kind`-definitioner!). Løsningen er at lægge typedefinitionen og interfacen i to forskellige modules, og at bruge `use` passende. Det bliver derfor til:

```
module COMPLEXSTUFF
! Module til at definere den hjemmelavede
! type, komplextal.
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
end module COMPLEXSTUFF

module INTERF
! Interface for procedurer der opererer med
! typen, komplextal.
  interface
    subroutine PRINT_KOMPLEXTAL (str, z)
      use COMPLEXSTUFF
      character(len=*),intent(in)  :: str
      type (komplextal),intent(in) :: z
    end subroutine PRINT_KOMPLEXTAL
  end interface
end module INTERF

program DERIVED_TYPES
! Afproever derived type KOMPLEXTAL.
  use COMPLEXSTUFF; use INTERF; implicit none
  ...
end program DERIVED_TYPES

subroutine PRINT_KOMPLEXTAL (str, z)
  use COMPLEXSTUFF;    implicit none
  character(len=*),intent(in)  :: str
  type (komplextal),intent(in) :: z

  character(len=20) :: localstr
  integer           :: L

  L = LEN_TRIM(str); localstr =  " "
  localstr(1:L) = str(1:L)
```

```
  print '(1x,a20,f8.4," +",f8.4," * i")', &
          localstr, z
end subroutine PRINT_KOMPLEXTAL
```

`Use`-sætningen for `COMPLEXSTUFF` findes i hovedprogram, proceduren, og også i den del af proceduren som interfacen `INTERF` gengiver. Lidt senere bliver dette programeksempel udvidet til flere procedurer, og det er sådan, at hver interface til disse skal indeholde `use`-sætningen.

10.6 Den tredje overload: defined assignment

I kapitel 9 blev de to slags overloads, generiske procedurer og definerede operatorer, beskrevet. Der er en tredje: Defineret tilordning (eller tilskrivning) (engelsk: **defined assignment**); og det er det rigtige sted her at forklare den.

Det er tit sådan, at en defineret type kan gives en værdi fra værdien af en intrinsic type. Vores type `komplextal` er et eksempel; den kan få de to komponenter af et tal af typen `complex`, og omvendt. Men det er syntaktisk forkert bare at skrive en tilordningssætning af formen

<komplextal> = <complex værdi>

eller omvendt, fordi de to størrelser har forskellige typer og derfor er inkompatible med hinanden. Det kan dog gøres korrekt, med nogle modules og interfaces. Mekanismen er at udvide (overload) lighedstegnet med de passende procedurer. Der skal være en interface, og den skal referere til de to procedurer, som **skal** være subroutines. Alt dette bliver illustreret med vores type `komplextal`, som vi nu vil kunne overføre til normale `complex` og omvendt. Dertil skriver vi de to subroutines `KOMPLEXTAL_COMPLEX` og `COMPLEX_KOMPLEXTAL`. De bliver begge indlagt i selve modulen, hvor vi nu også lægger printproceduren:

```
module COMPLEXSTUFF
! Module til at definere den hjemmelavede
! type, komplextal.
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
end module COMPLEXSTUFF

module INTERF
! Interface module for definitionen af "="
! som skal muliggoere tilskrivningerne c = k
! og k = c, hvor c er af den intrinsic type
! COMPLEX, og k er af vores defineret type,
! KOMPLEXTAL.
  interface assignment (=)
    module procedure KOMPLEXTAL_COMPLEX
    module procedure COMPLEX_KOMPLEXTAL
  end interface

  contains
  subroutine PRINT_KOMPLEXTAL (str, z)
    use COMPLEXSTUFF;    implicit none
    character(len=*),intent(in)  :: str
    type (komplextal),intent(in) :: z

    character(len=20) :: localstr
    integer           :: L

    L = LEN_TRIM(str); localstr =  " "
    localstr(1:L) = str(1:L)
    print '(1x,a20,f8.4," +",f8.4," * i")', &
            localstr, z
  end subroutine PRINT_KOMPLEXTAL

  subroutine KOMPLEXTAL_COMPLEX (ktal, ctal)
  ! Definerer tilskrivningen
  ! komplextal = complex som assignment.
    use COMPLEXSTUFF; implicit none
    type (komplextal),intent(out) :: ktal
    complex,intent(in)            :: ctal

    ktal%rdel = REAL(ctal)
    ktal%idel = AIMAG(ctal)
  end subroutine KOMPLEXTAL_COMPLEX

  subroutine COMPLEX_KOMPLEXTAL (ctal, ktal)
  ! Definerer tilskrivningen
  ! komplextal = complex som assignment.
    use COMPLEXSTUFF; implicit none
    complex,intent(out)           :: ctal
    type (komplextal),intent(in)  :: ktal

    ctal = CMPLX (ktal%rdel, ktal%idel)
  end subroutine COMPLEX_KOMPLEXTAL
end module INTERF
```

De afgørende sætninger her er

```
  interface assignment (=)
    module procedure KOMPLEXTAL_COMPLEX
    module procedure COMPLEX_KOMPLEXTAL
  end interface
```

og, at ligesom med definerede operatorer, skal interfacen bare henvise til de relevante module procedurer som sådan. Det ovenstående blev afprøvet med et lille testprogram:

```
program DERIVED_TYPES
! Afproever derived type KOMPLEXTAL.
```

```
  use COMPLEXSTUFF; use INTERF; implicit none

  complex           :: x, y
  type (komplextal) :: a, b

  a = KOMPLEXTAL (1.0, 1.0) ! Type constr.
  call PRINT_KOMPLEXTAL ("a = (1,1):", a)
  print *
  print '(" Tilskrivning fra complex", &
         &         "til komplextal:")'
  x = (77.0, 99.0)
  print '(" Complextal er    :", 2f10.4)', x
  b = x   ! Defined assignment, type k = c
  call PRINT_KOMPLEXTAL ("Overfoert:", b)
  print *
  print '(" Tilskrivning fra komplextal", &
         &              "til complex:")'
  y = a   ! Defined assignment, type c = k
  print '(" Complex resultat (1+1i):", &
         &                       2f10.4)', y
end program DERIVED_TYPES
```

og det gav resultatet

```
 a = (1,1):            1.0000 +  1.0000 * i

 Tilskrivning fra complex til komplextal:
 Complextal er    :   77.0000   99.0000
 Overfoert:           77.0000 + 99.0000 * i

 Tilskrivning fra komplextal til complex:
 Complex resultat (1+1i):    1.0000    1.0000
```

Det er, som programmet (modulen) er skrevet, muligt at kalde de bagvedliggende to procedurer direkte, ved kald som,

```
    call KOMPLEXTAL_COMPLEX (b, x)    ! b = x
    call COMPLEX_KOMPLEXTAL (y, a)    ! y = a
```

men det kan være uønskeligt at tillade det. Det kan forbydes, ved brug af `private`. Modulen bliver i så fald til

```
module INTERF
! Interface module for definitionen af "="
! som skal muliggoere tilskrivningerne c = k
! og k = c, hvor c er af den intrinsic type
! COMPLEX, og k er af vores defineret type,
! KOMPLEXTAL.

  private
  public :: assignment (=), PRINT_KOMPLEXTAL
  interface assignment (=)
    module procedure KOMPLEXTAL_COMPLEX
    module procedure COMPLEX_KOMPLEXTAL
  end interface
  ...
```

hvor sætningen

```
  private
```

først erklærer hele indholdet af interfacen som `private`, dvs. uvedkommende for andre enheder end interfacen selv. Det bliver så blødet lidt op ved den næste sætning,

```
  public :: assignment (=), PRINT_KOMPLEXTAL
```

som tillader adgang til selve lighedstegnet i denne sammenhæng og printproceduren, men ikke andet. Resultatet er det samme som vist ovenfor. Men hvis man prøver at bruge de direkte kald, opstår der en fejl.

10.7 Eksemplet udvidet

For at illustrere fremgangsmåden præsenteres der nu et lidt længere eksempel, som udvider det ovenstående. Der lægges nu functions til, der opererer med vores type `komplextal`. Ud over den før sete procedure til at printe dem, lægger vi nu fire functions til, for de fire aritmetiske operationer med komplekse tal. Disse functions er alle af typen `komplextal`, ligesom argumenterne overført til dem. Der bruges to modules. Hovedprogrammet læser to af disse typer ind, og skriver dem ud ved hjælp af proceduren sammen med resultaterne af de operationer, functionerne udfører. Til sidst testes disse resultater ved at lade Fortran beregne dem ved brug af de indbyggede operationer med typen `complex`. Kun de definerede operatorer er tilgængelige, samt udskrivningsproceduren, som beskrevet ovenfor, ved brug af `private` og selektiv `public`.

```
module COMPLEXSTUFF
! Module til at definere den hjemmelavede
! type, komplextal.
  implicit none
  type komplextal
    real :: rdel, idel
  end type komplextal
end module COMPLEXSTUFF

module INTERF
! Interface module for procedurer der
! opererer med typen komplextal. Alle
! procedurerne er contained her.
! Operatoren '=' er defineret; den skal goere
! det muligt at overfoere vores type til
! intrinsic complex, og omvendt, ved en normal
! tilskrivning.

  private
  public :: operator(+), operator(-), &
            operator(*), operator(/), &
            assignment(=), PRINT_KOMPLEXTAL
```

```
interface operator (+)
  module procedure KOMPLEXSUM
end interface
interface operator (-)
  module procedure KOMPLEXDIFF
end interface
interface operator (*)
  module procedure KOMPLEXPROD
end interface
interface operator (/)
  module procedure KOMPLEXDIV
end interface
interface assignment (=)
  module procedure KOMPLEXTAL_COMPLEX
  module procedure COMPLEX_KOMPLEXTAL
end interface

contains
function KOMPLEXSUM (z1, z2)
  use COMPLEXSTUFF;    implicit none
  type (komplextal) :: KOMPLEXSUM
  type (komplextal),intent(in) :: z1, z2

  KOMPLEXSUM%rdel = z1%rdel + z2%rdel
  KOMPLEXSUM%idel = z1%idel + z2%idel
end function KOMPLEXSUM

function KOMPLEXDIFF (z1, z2)
  use COMPLEXSTUFF;    implicit none
  type (komplextal) :: KOMPLEXDIFF
  type (komplextal),intent(in) :: z1, z2

  KOMPLEXDIFF%rdel = z1%rdel - z2%rdel
  KOMPLEXDIFF%idel = z1%idel - z2%idel
end function KOMPLEXDIFF

function KOMPLEXPROD (z1, z2)
  use COMPLEXSTUFF;    implicit none
  type (komplextal) :: KOMPLEXPROD
  type (komplextal),intent(in) :: z1, z2

  KOMPLEXPROD%rdel =   z1%rdel*z2%rdel &
                     - z1%idel*z2%idel
  KOMPLEXPROD%idel =   z1%rdel*z2%idel &
                     + z2%rdel*z1%idel
end function KOMPLEXPROD

function KOMPLEXDIV (z1, z2)
  use COMPLEXSTUFF;    implicit none
  type (komplextal) :: KOMPLEXDIV
  type (komplextal),intent(in) :: z1, z2

  real               :: div, reel, imag

  reel = z1%rdel*z2%rdel + z1%idel*z2%idel
  imag = z1%idel*z2%rdel - z1%rdel*z2%idel
  div =  z2%rdel**2 + z2%idel**2

  KOMPLEXDIV%rdel = reel / div
  KOMPLEXDIV%idel = imag / div
end function KOMPLEXDIV

subroutine PRINT_KOMPLEXTAL (str, z)
  use COMPLEXSTUFF;    implicit none
  character(len=*),intent(in)  :: str
  type (komplextal),intent(in) :: z

  character(len=20) :: localstr
  integer           :: L

  L = LEN_TRIM(str); localstr =  " "
  localstr(1:L) = str(1:L)
  print '(1x,a20,f8.4," +",f8.4," * i")', &
          localstr, z
end subroutine PRINT_KOMPLEXTAL

subroutine KOMPLEXTAL_COMPLEX (ktal, ctal)
! Definerer tilskrivningen
! komplextal = complex som assignment.
  use COMPLEXSTUFF; implicit none
  type (komplextal),intent(out) :: ktal
  complex,intent(in)            :: ctal

  ktal%rdel = REAL(ctal)
  ktal%idel = AIMAG(ctal)
end subroutine KOMPLEXTAL_COMPLEX

subroutine COMPLEX_KOMPLEXTAL (ctal, ktal)
! Definerer tilskrivningen
! komplextal = complex som assignment.
  use COMPLEXSTUFF; implicit none
  complex,intent(out)           :: ctal
  type (komplextal),intent(in)  :: ktal

  ctal = CMPLX (ktal%rdel, ktal%idel)
end subroutine COMPLEX_KOMPLEXTAL
end module INTERF

program DERIVED_TYPES
! Afproever derived type KOMPLEXTAL,
! sammenligner med Fortrans.

  use COMPLEXSTUFF; use INTERF; implicit none

  complex           :: x, y, z
  type (komplextal) :: a, b, c
  print*," Indtast et komplex tal (2 reals):"
  read *, a
  print *, " .. og et til:"
  read *, b
  call PRINT_KOMPLEXTAL ("a:", a)
  call PRINT_KOMPLEXTAL ("b:", b)
  print *
  c = a + b; call PRINT_KOMPLEXTAL("a+b:", c)
  c = a - b; call PRINT_KOMPLEXTAL("a-b:", c)
```

```
  c = a * b; call PRINT_KOMPLEXTAL("a*b:", c)
  c = a / b; call PRINT_KOMPLEXTAL("a/b:", c)
  print *
  print *, &
   " De fire operationer udfoert af Fortran:"
  x = a;   y = b     ! Defineret tilskrivning

  print '(" Complex a+b:", 2f10.4)', x + y
  print '(" Complex a-b:", 2f10.4)', x - y
  print '(" Complex a*b:", 2f10.4)', x * y
  print '(" Complex a/b:", 2f10.4)', x / y

end program DERIVED_TYPES
```

Resultatet af kørsel (programmets prompts – den indledende dialog – er fjernet):

```
 a:                    1.0000 +  1.0000 * i
 b:                    1.0000 + -1.0000 * i

 a+b:                  2.0000 +  0.0000 * i
 a-b:                  0.0000 +  2.0000 * i
 a*b:                  2.0000 +  0.0000 * i
 a/b:                  0.0000 +  1.0000 * i

  De fire operationer udfoert af Fortran:
 Complex a+b:    2.0000    0.0000
 Complex a-b:    0.0000    2.0000
 Complex a*b:    2.0000    0.0000
 Complex a/b:    0.0000    1.0000
```

Kapitel 11
Pegepinde og datastrukturer

Pegepinde (eng.: **pointers**) muliggør brugen af visse datastrukturer, som gør nogle opgaver lettere at programmere og giver bedre effektivitet i visse tilfælde. Først skal selve pegepindene beskrives.

11.1 Pegepinde

Som navnet antyder, er en pegepind (pointer) en slags variabel, der henviser til en anden. Pegepinden indeholder altså ikke selv en værdi, bortset fra informationen om, hvor den peger hen. Det, der peges på, er kaldt **target** eller, på dansk, **mål**. I Fortran er det sådan, at når pegepinden bliver refereret til, kan referencen betyde enten pegepinden selv, eller dens mål, alt efter sammenhængen. Først nogle enkeltheder om hvordan man opstiller og arbejder med pegepinde.

Erklæring, association

Pegepinde skal erklæres sammen med deres targets, og her ser man dualiteten ved pegepinde, som kan være både henvisninger til deres target, og target selv. De har nemlig samme type som deres target. Her er en erklæring af en pegepind samt dens target:

```
real,pointer :: p
real,target  :: x
```

Nøgleordene `pointer` og `target` skal være med. Variablen `p` tjener altså til at henvise til variablen `x`. Det er dog sådan, at en erklæring i sig selv ikke får pegepinden til at pege på noget bestemt; den er, i starten, uassocieret (eng.: **disassociated**), og skal associeres med sit target. Det er klart, hvorfor det er sådan: der kan være flere pegepinde og flere targets, og associationerne skal fastsættes. Der er to måder at associere en pegepind på. Den ene af dem er at bruge en særlig operator dertil. I vores eksempel kan man associere `p` med `x` med sætningen

```
p => x
```

hvor `=>` er operatoren. Pegepinden er nu associeret med `x`. Hvis man herefter refererer til `p`, mener man `x`. For eksempel, de to sætninger

```
print *, x
print *, p
```

har samme resultat, de skriver værdien af `x` ud. Man kan også associere en pegepind med en andens target, indirekte via den anden pegepind:

```
real,pointer :: p, q
real,target  :: x, y
...
p => x
q => p
```

Den sidste sætning bevirker, at `q` nu også peger på `x` (ikke på `p`!). Det er helt i orden at have to pegepinde til at pege på samme target. Bemærk også, at sætningen

```
q = p
```

betyder noget helt andet; lighedstegnet ændrer sammenhængen radikalt, og begge pegepinde fungerer nu som deres targets. Det vil sige, at sætningen fungerer som en tilordningssætning, der tilskriver target af `q` værdien af target af `p`. For at gøre det klarere, bruger vi følgende eksempel:

```
real,pointer :: p, q
real,target  :: x, y

x = 1;  y = 2
p => x;  q => y
print *, x, y
print *, p, q;   print *
q = p               ! 1
print *, x, y
print *, p, q;   print *
y = 3               ! 2
print *, x, y
print *, p, q;   print *
q => p              ! 3
print *, p, q
print *, p, q
```

Resultatet af en kørsel af dette er:

```
 1.000000       2.000000
 1.000000       2.000000

 1.000000       1.000000
 1.000000       1.000000

 1.000000       3.000000
 1.000000       3.000000
```

```
1.000000        1.000000
1.000000        1.000000
```

De første to `prints` giver værdierne 1 og 2, som forventet, både for variablerne og deres associerede pegepinde. Sætningen mærket med "! 1" overfører værdien af `x` til selve `y`; de er nu begge lig med 1. Pegepindene peger stadig på henholdsvis `x` og `y`, som kan ses efter "! 2", hvor `y` får en ny værdi, og den skrives ud i de to næste `prints`. Men ved "! 3" associeres `q` nu også med x, og de sidste to `prints` viser dette.

Det, at man refererer til targetvariablen når man blot nævner dens pegepind, udstrækker sig til de normale operationer. Hvis der f.eks. foreligger

```
real,pointer :: p, q, r
real,target  :: x, y, z
...
p => x;  q => y;  r => z
```

så ville sætningen

```
p = q + r
r = 77
```

være ensbetydende med

```
x = y + z
z = 77
```

Noget man nu kan gøre med pegepinde, er at bytte to af dem, så de bytter targets, uden at de to targetvariabler er ændrede. Det foregår ved hjælp af associeringer og en midlertidig pegepind:

```
real,pointer :: p, q, byt
real,target  :: x, y
...
p => x;  q => y  ! (p,q) peger paa (x,y)
byt => p         ! Byt peger nu ogsaa paa x
p => q           ! p peger nu (ogsaa) paa y
q => byt         ! q peger nu paa x
```

Alt det ovenstående kan virke lidt underligt og overflødigt; så vidt er der ikke blevet vist noget, man ikke ellers kan med mindre indviklede metoder. Men det kommer lidt senere. Først skal nogle intrinsics og nogle enkelte regler beskrives.

Pegepind-intrinsics, regler

Der er en intrinsic function samt en sætning til pegepinde. Det er

```
ASSOCIATED (pegepind[, target])
```

en logical function. Følgende regler gælder:

- Hvis `target` (som er et valgfrit argument) ikke er til stede, er resultatet `.true.` hvis pegepinden er associeret med et hvilket som helst target, ellers `.false.`
- hvis `target` er til stede, er resultatet `.true.` hvis pegepinden er associeret med dette target, ellers `.false.`
- Hvis `target` er til stede og selv er en pegepind, er resultatet `.true.` hvis pegepinden er associeret med det samme target som `target` er associeret med; de skal begge være associeret med det samme target, ellers er resultatet `.false.`
- pegepinden må ikke være i udefineret tilstand.

Den tredje regel viser, at `target` i sig selv må være en pegepind, og den skal være associeret med sit target; functionen tester, om den første pegepind er associeret med dette (sekundære) target. Angående den sidste regel: Når man erklærer en pegepind, er dens tilstand udefineret – hverken associeret eller disassocieret. De fleste systemer ville nok give resultatet `.false.` når functionen er kaldt, muligvis sammen med en advarsel om, at variablen ikke er defineret. Men i princippet burde en pegepind have en defineret tilstand, hvis den er argument til et kald af `ASSOCIATED`. Det åbner spørgsmålet om, hvordan man disassocierer en pegepind. Det gør man ved sætningen

```
NULLIFY(arg)
```

hvor `arg` er en pegepind. Hvis den var associeret, bliver den frakoblet (disassocieret); hvis dens tilstand var udefineret, bliver den defineret som disassocieret. Dette bliver vigtigt når anvendelse af pegepinde i forbindelse med visse datastrukturer bliver gennemgået.

11.2 Pegepinde, arrays og strukturer

Indtil nu er det givetvis svært at se, hvorfor pegepinde skal bruges. Det skal nu blive klart. Der er hovedsageligt to fordele ved pegepinde til targets, over selve targets. Den første af de to er, at man kan flytte rundt med pegepinde, og samtidig lade deres targets blive hvor de er. Det skal der gives et mindre eksempel på om lidt, men der er andre, mere væsentlige fordele, som også skal beskrives. Man kan for eksempel spare en hel del regnetid ved at flytte nogle pegepinde, i stedet for deres targets, hvis disse er større klumper af data, som f.eks. strukturer eller arrays. Den anden

fordel er at pegepinde gør visse dynamiske datastrukturer mulige, som f.eks. kædede lister og binære træer. Også de bliver forklaret.

Array pointers

Her introduceres **array pointers**, dvs. pegepinde til arrays. Her er det sådan, at en enkelt pegepind er koblet til (associeret med) et array. Pegepinden er i sig selv ikke et array, men kan virke som et i nogle sammenhænge. Et eksempel vil gøre det klart:

```
integer,pointer,dimension(:) :: p
integer,target,dimension(10) :: x
...
p => x
```

I erklæringen af pegepinden `p` er der en indikation af, at den kommer til at pege på et array, men ikke hvor stort det skal være. Det er kun antal af dimensionerne der vises (i dette tilfælde, en), og det skal være med. Selve arrayet, som bliver til `p`'s target (`x`) får sin faste størrelse i erklæringen (men kunne godt være allokerbart). Det er den sidste sætning, der får `p` til at pege på arrayet. Der er nogle lidt modsigende egenskaber af pegepinden. Hvis man senere i programmet refererer til `p` som sådan, refererer man derved til hele arrayet `x`. Det er også muligt at referere til et bestemt element i `x` ved, f.eks. `p(7)`. Det sidste synes at betyde, at der findes et helt array af pegepindelementer `p`, men det er ikke tilfældet. Internt i systemet er der kun en enkelt `p`, samt noget information om dens target, bl.a. targets antal dimensioner og størrelserne. Det virker muligvis lidt indviklet, men er af stor betydning for effektiviteten af visse operationer, som nu skal belyses.

Lad os sige, at vi har et stort 2-D array `A`, `1000*1000` og vil bytte det med et andet, `B`, lige så stort. Der er to måder at gøre det på uden pegepinde. Enten erklærer man endnu et array `TEMP`, og overlader det til systemet at gøre operationen effektivt ved sætninger som

```
TEMP = A
A = B
B = TEMP
```

eller – hvis der ikke er plads til så mange arrays, bruger man en dobbeltløkke til at bytte arrayene element for element:

```
do i = 1, 1000
  do j = 1, 1000
     temp = A(i,j)
     A(i,j) = B(i,j)
     B(i,j) = temp
  enddo
enddo
```

hvor `temp` nu er en skalar. Det er klart en del operationer, der skal udføres. Forfatteren har programmeret dette eksempel og målt den tid, regnemaskinen brugte, og det tog 0.18 sekunder, hvilket ikke synes at være så meget, men maskinen er hurtig, og det er faktisk et betydeligt forbrug.

Hvis man derimod bruger pegepinde, og bytter dem i stedet, tager hele operationen så lidt tid, at den ikke kan måles. Det foregår sådan (det er et udsnit af programmet):

```
integer,parameter :: N=1000
real,target   :: x(N,N), y(N,N)
real,pointer  :: p1(:,:), p2(:,:), swp(:,:)

p1 => x;  p2 => y
...
swp => p1;  p1 => p2;  p2 => swp
...
```

Hvor `p` og `q` før pegede henholdsvis på `X` og `Y`, peger de nu på `Y` og `X` i stedet. Operationerne involverer kun de tre pegepinde, og det går hurtigt.

Strukturpegepinde

Det er muligvis svært at forestille sig en programmeringssituation, hvor det er nødvendigt at bytte to store arrays. Men når man programmerer med strukturer, især arrays af strukturer, er det meget tit, at man vil anordne dem i en vis orden, og ændre den hen ad vejen. Strukturer kan blive ret store, og det kan blive tidskrævende at flytte rundt med dem. Her kommer pegepinde ind. Helt analogt til det ovennævnte eksempel med to store arrays, bytter vi nu pegepindene på to strukturer. Vi bruger en defineret type for en student, som her kun indeholder navn og årskortnummer, og vi holder os til kun to af dem:

```
type student_type
  character(len=40) :: fornavn
  character(len=40) :: efternavn
  integer           :: aarskortnr
endtype student_type
type(student_type),pointer :: p1,p2,p_temp
type (student_type),target :: stud1, stud2

...
p1 => stud1;  p2 => stud2
...
p_temp => p1;  p1 => p2;  p2 => p_temp
```

Igen var det nødvendigt blot at bytte to pegepinde, hvor det ellers ville have været nødvendigt at bytte alle de enkelte komponenter af de to strukturer.

Eksemplet bliver nu forbedret lidt ved at lægge typeerklæringen ind i en module:

```
module DEFS
  implicit none
  type student_type
    character(len=40) :: fornavn
    character(len=40) :: efternavn
    integer           :: aarskortnr
  endtype student_type
end module DEFS

program SWAP_TWO_STRUCTURES

  use DEFS;    implicit none

  type(student_type),pointer :: p1,p2,p_temp
  type (student_type),target :: stud1, stud2

 ...
  p1 => stud1;  p2 => stud2
  ...
  p_temp => p1;  p1 => p2;  p2 => p_temp
  ...
end program SWAP_TWO_STRUCTURES
```

Dette gør det muligt at bygge videre på eksemplet, som det nu skal ske.

Arrays af pegepinde

Man ville næppe skrive et program for at administrere et antal studerende, og give dem allesammen deres individuelle navne (`stud1`, `stud2`, osv.). Det er oplagt at konstruere et array af disse strukturer. Det giver problemet, at man ikke umiddelbart kan have et array af pegepinde – når man erklærer sådan et array, får man jo en enkelt pegepind, der peger på et array. Hvis man prøver at erklære et pegepindearray med en vis dimension (i stedet for `(:)`) så er det en fejl. Det kan man snyde sig til. Det er nemlig muligt at lægge en pegepind ind i en struktur – givetvis den eneste komponent i strukturen – og det er helt lovligt at have arrays af strukturer. På denne måde kan vi altså opstille et array af forskellige pegepinde, der peger på et target array. Vi udvider nu eksemplet i den retning. Typen `student_type` danner et array `student`, og `p` er det tilsvarende array af strukturen af typen `pegepind`, hvis eneste komponent `pind` er selve pegepinden. Opgaven her er at indlæse et antal N strukturer af `student_type` og udskrive dem i alfabetisk rækkefølge efter efternavnene med fornavnene i anden kolonne. Man kunne også tænke sig at sortere efter årskortnumrene, og at gøre det valgfrit, men vi holder det simpelt:

```
module DEFS
  implicit none
  integer,parameter :: Nmax=100, maxlen=30
  type student_type
    character(len=maxlen) :: fornavn
    character(len=maxlen) :: efternavn
    integer               :: aarskortnr
  endtype student_type
  type pegepind
    type (student_type),pointer :: pind
  endtype pegepind
end module DEFS

program SORT_STRUCTURES
! Sortering af student-strukturer ved brug af
! et pegepind-array. Det er kun pegepindene
! der bliver sorterede.

  use DEFS;    implicit none

  type(pegepind),allocatable :: p(:)
  type (student_type),target :: student(Nmax)
  integer                    :: N_stud, i

  call LAES_DEM_IND (student, N_Stud)
  print '(" Der blev", i4, &
   & " studenterrecords laest ind:")', N_stud
  ALLOCATE (p(N_stud)) ! Pegepindene oprettes
  do i = 1, N_stud       ! ... og
    p(i)%pind => student(i) ! ... associeres.
  enddo
  ! Pegepindene sorteres:
  call SORTER_DEM (p, N_stud)
  ! Udskriv stud-listen:
  call SKRIV_DEM_UD (student, N_stud)
  ! ... og den sorterede:
  call SKRIV_PTR_UD (p, N_stud)

end program SORT_STRUCTURES

subroutine LAES_DEM_IND (stud, N)
! Indlaeser, fra standardenheden, saa mange
! strukturer af typen student_type, som der
! er (max Nmax), dvs indtil end-of-input.
! De bliver sat ind i arrayet.

  use DEFS;    implicit none
  type (student_type) :: stud(Nmax)
  integer             :: N

  integer             :: ios

  N = 0
  do
    read (*,'(a)',iostat=ios), stud(N+1)%fornavn
    if (ios /= 0) exit
    N = N + 1                     ! (Tael op)
    read (*,'(a)',iostat=ios), &
```

```
              stud(N)%efternavn
    if (ios /= 0) &
        STOP " Noget galt med input-data!"
    read (*,*,iostat=ios), stud(N)%aarskortnr
    if (ios /= 0) &
        STOP " Noget galt med input-data!"
    if (N == Nmax) then
      print '(i6, " indlaest,", &
            & " hvilket er maksimum.")', Nmax
      exit
    endif
  enddo

end subroutine LAES_DEM_IND

subroutine SKRIV_DEM_UD (stud, N)
! Skriver de N studenter ud.

  use DEFS;    implicit none
  type (student_type) :: stud(Nmax)
  integer             :: N

  integer             :: i

  print '(/" Studenterliste som den staar:")'
  print '(" No.",3x,"Aarsk",6x,"Fornavn",   &
        & 10x, "Efternavn")'
  do i = 1, N
    print '(i4, i8, 6x, a20, a30)', &
           i, stud(i)%aarskortnr, &
           stud(i)%fornavn, stud(i)%efternavn
  enddo

end subroutine SKRIV_DEM_UD

subroutine SKRIV_PTR_UD (p, N)
! Skriver N studenterne ud efter pegepindene.

  use DEFS;    implicit none
  integer         :: N
  type (pegepind) :: p(N)

  integer         :: i
  type (student_type),pointer :: q

  print &
  '(/" Studenterliste efter pegepindarray:")'
  print '(" No.",3x,"Aarsk",6x, "Fornavn", &
         &                 10x, "Efternavn")'
  do i = 1, N
     q => p(i)%pind
    print '(i4, i8, 6x, a20, a30)', &
      i, q%aarskortnr, q%fornavn, q%efternavn
  enddo

end subroutine SKRIV_PTR_UD
```

```
subroutine SORTER_DEM (p, N)
! Sorter pegepindene til de N studenter,
! efter navnene kaedede sammen
! (efternavn//fornavn).

  use DEFS;    implicit none
  integer         :: N
  type (pegepind) :: p(N)

  character(len=2*maxlen)     :: st1, st2
  integer                     :: i, j
  type (student_type),pointer :: q

! En simpel bubble-sort; eftersom der er
! mange ens efternavne, kaeder vi efternavn
! og fornavn sammen.

  do i = 1, N
    do j = 1, N-i
     st1 =    p(j)%pind%efternavn &
           // p(j)%pind%fornavn
     st2 =    p(j+1)%pind%efternavn &
           // p(j+1)%pind%fornavn
     if (st1 > st2) then
       q => p(j)%pind           ! Byt pindene
       p(j)%pind => p(j+1)%pind !
       p(j+1)%pind => q         !
     endif
    enddo
  enddo

end subroutine SORTER_DEM
```

Programmet kan udvides og forbedres. For eksempel burde man bruge en interface, og muligvis lægge alle procedurer ind i en module osv. Programmet blev kørt med et lille testdatasæt bestående af linjerne:

```
Jens Peter
Smith
123456
Hans Uffe
Smith
234567
Bogumil
Jensen
999999
Thorkild
Hansen
981234
```

og resultatet blev:

```
 Der blev   4 studenterrecords laest ind:
```

```
Studenterliste som den staar:
No.   Aarsk      Fornavn          Efternavn
  1  123456      Jens Peter          Smith
  2  234567      Hans Uffe           Smith
  3  999999      Bogumil             Jensen
  4  981234      Thorkild            Hansen

Studenterliste efter pegepindarray:
No.   Aarsk      Fornavn          Efternavn
  1  981234      Thorkild            Hansen
  2  999999      Bogumil             Jensen
  3  234567      Hans Uffe           Smith
  4  123456      Jens Peter          Smith
```

Det at vi her flytter pegepindene og lader selve listen af strukturerne blive urørt, har to fordele. Hvis strukturen nu udvides til flere komponenter, bliver det mere og mere effektivt at flytte pegepindene i stedet for strukturerne, hvor alle komponenter ellers skulle flyttes. For det andet giver det os mulighed for at bevare den oprindelige liste i usorteret tilstand, hvilket sommetider er ønskeligt, og ellers ville tvinge os til at oprette en kopiliste.

Pegepinde med implicit target

Det er muligt at have en pegepind der peger på et target som i sig selv ikke er erklæret som variabel. Man kan f.eks. erklære en arraypegepind og allokere den:

```
integer,pointer :: p(:)
...
ALLOCATE (p(100))
```

Her ses der den anden måde at associere en pegepind på (operatoren `=>` var den første), ved hjælp af kommandoen `ALLOCATE`. Den er generelt af formen

```
ALLOCATE (<ptr>(<size>) [,STAT=<svar>])
```

Den anden del af kaldet, `STAT` (som er valgfri), sætter variablen `svar` til nul hvis alt går godt, ellers til en positiv værdi. Det vil normalt betyde, at der ikke var plads i hukommelsen til den ønskede allokering. Vi beskriver samtidig modparten til denne kommando:

```
DEALLOCATE (<ptr>[,STAT=svar])
```

som frigør det bagvedliggende array som pegepinden peger på (hvis den var associeret, eller indikerer en fejl i `svar`).

Tilbage til vores eksempel. Pegepinden `p` peger på et array af 100 heltal, uden at de er blevet erklærede. De eksisterer dog i hukommelsen som et navnløst array. Man kan nu operere med `p`, som om det var arrayet. F.eks. kan man henvise til de enkelte elementer `p(i)`, hvor man så mener det bagvedliggende array.

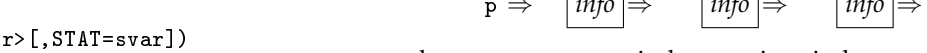
Alt dette er ikke noget særligt nyttigt, fordi man kan gøre det samme ved at oprette et rigtigt allokerbart array med navn, i stedet for en arraypointer. Men mekanismen, der tillader at allokere en pegepind der peger på en variabel der samtidigt får plads i hukommelsen, kan bruges i forbindelse med dynamiske strukturer, som nu skal behandles.

11.3 Pegepinde og dynamiske datastrukturer

Arrays er meget nyttige, men har visse begrænsninger. Alle elementer skal bestå af en enkelt skalar. Værre er det, når man vil sætte et nyt element ind et sted i midten af et array – det kræver, at hele resten af arrayet skal skubbes længere hen. Og selv når man erklærer et array `allocatable`, hvilket giver en vis fleksibilitet, sammenlignet med at erklære arrayet med så stort omfang, som man mener, det kunne blive, er det stadigvæk umuligt senere at udvide arrayet, når flere elementer kommer til undervejs i et program. De sidstnævnte problemer bliver løst af **kædede lister** (eng.: **linked lists**), som er elementer (altid strukturer) der danner en kæde, som kan udvides, både ved enderne og indeni. Desuden er der datastrukturer som **binære træer** (eng.: **binary trees**), der gør nogle operationer meget effektive. Der er andre datastrukturer, f.eks. køer, stakke, m.m.; her henvises den interesserede læser til de mange bøger om emnet, mest med ordene "data structures" i titlen. Vi fokuserer her på de to, kædede lister og det binære træ. De kræver begge brug af pegepinde.

Kædede lister

En kædet liste er en række strukturer, hvor hvert element peger på det næste, undtagen det sidste, der ikke peger på noget. Hele kæden begynder med hovedelementet, og der er en enkelt pegepind, der peger på dette element. Det kan se sådan ud:



hvor `p` er en pegepind, som giver indgangen til kæden, der her består af kun de tre elementer, hvor den sidste pegepind ikke peger på noget (er, med andre ord, ikke associeret). Lad os starte med at definere et element. Det foregår via pegepinden, der peger på det, hvilket ligner situationen i sidste afsnit, hvor en pegepind pegede på et array, som i sig selv aldrig blev defineret direkte. Her skal der dog defineres en strukturtype –

typen for de enkelte elementer. En anvendelse af denne datastruktur kunne være at repræsentere et polynomium:

$$p(x) = a_1 x^{p_1} + a_2 x^{p_2} + \ldots$$

hvor hvert element indeholder en koefficient a_i og en potens p_i. Denne struktur tillader operationer med hele polynomier: evaluering af hele polynomiet for en given x-værdi, addition af to polynomier, multiplikation, osv. Et enkelt element i polynomiet, som kan repræsenteres som $\boxed{info}\Rightarrow$ kan defineres som

```
type polyled
  integer               :: a, p
  type(polyled),pointer :: next
end type polyled
```

hvor `a` og `p` står for henholdsvis koefficienten og potens i det i'te led $a_i x^{p_i}$. Det interessante er at se pegepinden i strukturen; den har typen `polyled`, som den selv er en del af. Den kommer til at pege på det næste led i kæden. Hele denne kæde skal starte med en pegepind af samme type:

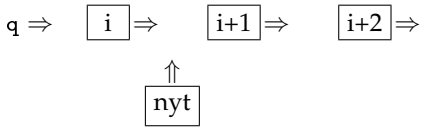
```
type(polyled),pointer :: start
```

Vi er nu i stand til at opbygge vores kædede liste, der skal repræsentere et polynomium. Der er visse operationer vi er nødt til at finde ud af:

- At starte kæden med et enkelt led
- at lægge et nyt element til enden af kæden
- at lægge et nyt element til starten af kæden
- at sætte et nyt element ind imellem to eksisterende
- at fjerne et givet element.

(Det sidste kan blive nødvendigt i polynomiumsammenhæng, f.eks. når en addition udsletter et led, dvs. bevirker, at dets koefficient bliver nul).

Der er to operationer, som går igen i et program med kædede lister: at sætte et nyt element ind, og at fjerne et. I følgende diagram



skal der sættes et nyt element ind imellem det i'te og det (i+1)'te. Vi er rykket frem i kæden, således at pegepinden `q` peger på det i'te, og vi har fat i de komponenter, der indgår i det nye element, der skal oprettes og indsættes – her, i vores polynomiumeksempel, de to tal, a og p. Først oprettes der et nyt element ved at allokere det til en ledig pegepind; lad os kalde den `qny`:

```
ALLOCATE (qny)  ! Et tomt element er skabt
```

Derefter indsætter vi komponenterne:

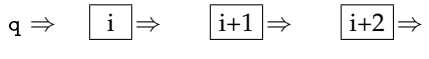
```
qny%a = a
qny%p = p
```

og elementet er nu klart til at indsættes. Det der nu skal ske er, at den pegepind, `q%next`, som lige nu peger på element (i+1), skal pege på vores nye element, og dets pegepind (`qny%next`) skal pege på det (i+1)'te. Det foregår på følgende måde:

```
qny%next => q%next ! qny peger paa (i+1)
q%next   => qny    ! q peger nu paa det nye
```

Der er tilsvarende operationer, når et nyt led skal indsættes foran alle andre eller bag efter alle i kæden. Disse kan ses i det større program som følger nedenfor.

For at illustrere eliminering af et led, tag diagrammet



hvor `q` igen peger på det i'te, og det er det (i+1)'te, der skal elimineres. Her er koden til det (`qtemp` er en midlertidig pegepind):

```
qtemp => q%next    ! qtemp peger paa (i+1)
q => qtemp%next    ! q peger nu paa (i+2)
DEALLOCATE (qtemp) ! Frigiv det fraklippede
```

Det sidste er vigtigt, for at plads i hukommelsen bliver frigivet igen (ellers kunne man nøjes med kun den midterste sætning af de tre). I seriøse programmer kan det knibe med hukommelsesplads. Som med indsætning af nye elementer, er der igen specielle tilfælde for eliminering, alt efter hvor i kæden der skal skæres ud, og læseren kan se eksempler på disse i det nævnte program.

Vi skal nu vise, hvordan man laver et program med sådanne polynomier. Programmet skal indlæse et antal (a,p) par, og læg dem til det polynomium, der er. Leddene skal kædes sammen med stigende potenser. Efter hver addition af et nyt led skal hele polynomiet skrives ud og evalueres for en bestemt x-værdi. Der er flere mulige situationer, alle skal tages højde for ved addition:

- Det nye led kan blive det allerførste; dvs. der var ikke et polynomium i forvejen;
- det nye led kan blive det nye første led, hvis dets potens er mindre end alle andres;
- det nye led kan blive det nye allersidste, hvis dets potens er større end alle andres;
- det nye led kan gå ind et sted midt i den gamle kæde;

- det nye led skal lægges til et bestående, hvis de har samme potens.

Der er tilsvarende muligheder når et led skal fjernes. Vi definerer en pegepind `start`, som altid peger på starten af kæden; den er så at sige indgangen til kæden. Når programmet starter, er denne pegepind uassocieret. Hele programmet følger her:

```
module POLYDEFS
  type polyled
    integer               :: a, p
    type(polyled),pointer :: next
  end type polyled
end module POLYDEFS


module SUBINTERF
  interface
    subroutine SAET_IND (start, a, p)
      use POLYDEFS
      type (polyled),pointer :: start
      integer                :: a, p
    end subroutine SAET_IND
    subroutine FJERN_NULLED (start)
      use POLYDEFS;   implicit none
      type(polyled),pointer   :: start
    end subroutine FJERN_NULLED
    subroutine VIS (start)
      use POLYDEFS;   implicit none
      type(polyled),pointer   :: start
    end subroutine VIS
    subroutine EVAL (x, sum, start)
      use POLYDEFS
      real                   :: x, sum
      type(polyled),pointer :: start
    end subroutine EVAL
  end interface
end module SUBINTERF


program POLYNOMIUM
! Programmet etablerer et polynomium som en
! kaedet liste. Der laeses en x-vaerdi ind,
! og derefter et antal af koefficienter og
! potenser, parvis, indtil end-of-input (eoi)
! signalerer slut. For hvert nyt input bliver
! kaeden udvidet og printed ud, sammen med
! p(x)-vaerdien.

  use POLYDEFS;  use SUBINTERF; implicit none
  real                    :: x, sum
  integer                 :: a, p, ios
  type(polyled),pointer   :: start, q

  print '(" x?")';          ! x-vaerdien,
  read *, x                 !.. for loebende
  print '(" x =", f8.3)', x ! ... vaerdi p(x)
  NULLIFY (start)
  do
    read (*,*,iostat=ios), a, p   ! nyt (a,p)
    if (ios /= 0) exit       ! ... indtil eoi
    call SAET_IND(start,a,p)  ! Insaet leddet
    print '(" Nyt (a,p) par:" 2i4)', a, p
    call FJERN_NULLED(start) ! Fjern nul-led?
    call VIS (start)
    call EVAL (x, sum, start)  ! Evaluer p(x)
    print '(" p(x) =", f12.3)', sum
  enddo

end program POLYNOMIUM


subroutine SAET_IND (start, a, p)
! De nye tal, a & p, er lagt ind i polynomiet
! som START peger paa. Der er fire mulige
! situationer: Det nye led skal gaa ind
! foran, midt i eller bag alle andre; eller
! (hvis p er det samme som et eksisterende
! led) skal a bare laegges til et gammelt a.

  use POLYDEFS;   implicit none

  type (polyled),pointer :: start
  integer                :: a, p
  type (polyled),pointer :: q, led, slut
  logical                :: fundet

  ALLOCATE (led)
  led%a = a;  led%p = p;  NULLIFY (led%next)
  fundet = .false.           ! Initialisering

  if (.not.ASSOCIATED(start)) then
    ! Leddet starter en ny kaede:
    ALLOCATE (start);   start => led
  else     ! Der er en kaede i forvejen:
    q => start
    do                    ! Find sidste og
      if (p == q%p) then ! .. se om p er der;
        q%a = q%a + a    ! Ja: saet a ind.
        fundet = .true.  ! Maerk dette.
      endif
      if (.not.ASSOCIATED(q%next)) then
        slut => q;  exit  ! Find nu endeleddet
      endif
      q => q%next
    enddo                   ! Vi har nu enden
    if (.not. fundet) then  !.. faerdig? Ikke:
      if (p < start%p) then ! Skal det nye
        led%next => start   !..gaa foran alt?
        start => led
      else if(p>slut%p)  then  ! Eller bagest?
        slut%next => led
      else  ! Hvis ingen af de to, find hvor:
```

```
        q => start     ! gennemsoeger kaeden:
        do
          if (p>q%p .and. p<q%next%p) then
            ! Fundet stedet!
            led%next => q%next   ! Saet ind
            q%next => led        ! ..
            exit                 ! .. faerdig.
          endif
          q => q%next ! Ikke fundet, ryk frem
        enddo
      endif
    endif
  endif
end subroutine SAET_IND


subroutine FJERN_NULLED (start)
! Finder evtl. led, der har nul a-koeff,
! og luger dem ud af kaeden.

  use POLYDEFS;   implicit none
  type(polyled),pointer   :: start

  type(polyled),pointer   :: q1, q2
  logical                 :: fundet_een

  if (ASSOCIATED(start)) then
    ! Kun hvis der ER led!
    do                    ! Flere gennemgange:
      if (.not.ASSOCIATED(start%next)) then
        ! Kun et led:
        if (start%a == 0) then ! Er det tomt,
          DEALLOCATE (start)   ! ..frigiv
          NULLIFY (start) ! kaeden er nu tom.
        endif
        exit      ! I hvert fald exit herfra,
      endif       ! .. der er ikke flere led.

      fundet_een = .false.   ! Mindst to led.
      if (start%a == 0) then ! Er startleddet
        q1 => start%next  ! ..tomt, eliminere
        DEALLOCATE (start)
        NULLIFY (start)
        start => q1            ! .. ny start.
        fundet_een = .true. ! Maerk sletning
      else
        q1 => start      ! Start er ikke tomt,
        do           ! .. og der er flere led:
          q2 => q1%next
          if (q2%a == 0) then  ! Naeste tomt,
            q1%next => q2%next     ! fjernes
            DEALLOCATE (q2)
            fundet_een = .true.    ! Maerkes.
          else if &
             (.not.ASSOCIATED(q2%next)) then
            exit                ! Ramt enden
          else
            q1 => q2        ! Flere? Ryk frem
          endif
          if (fundet_een) exit  ! Een er nok,
        enddo                   ! denne gang.
      endif
      if (.not. fundet_een) exit  ! Ingen lig
    enddo                !..fundet, faerdig.
  endif
end subroutine FJERN_NULLED

subroutine VIS (start)
! Display af polynomium p; leddene bliver sat
! ind i char-streng. Pegepinden next er
! indikeret med T hvis den er associeret,
! ellers F.

  use POLYDEFS;   implicit none
  type(polyled),pointer :: start

  type(polyled),pointer :: q
  character(len=70)     :: line
  integer               :: i

  i = 1;   q => start;  line = " "
  if (.not.ASSOCIATED(start)) then
    print '(" Kaeden er tom.")'
  else
    do
      if (.not.ASSOCIATED(q)) exit
      write(line(i:i+9),                    &
           '("[",i2,",",i2,",",L1,"]",1x)'),&
            q%a, q%p, ASSOCIATED(q%next)
      i=i+10
      q => q%next
    enddo
    print '(a)', line(1:LEN_TRIM(line))
  endif
end subroutine VIS

subroutine EVAL (x, sum, start)
! Display af polynomium p, samt evaluering
! af p(x).

  use POLYDEFS;   implicit none
  real                  :: x, sum
  type(polyled),pointer :: start, q


  if (ASSOCIATED(start)) then   ! Er der led?
    NULLIFY (q);  q => start
    do
      sum = sum+q%a*x**q%p  ! Laeg a*x**p til
      q => q%next           ! Flyt pegepinden
      if (.not.ASSOCIATED(q)) exit
    enddo
  endif
  sum = 0      ! Evaluer p(x), start med nul.
```

```
end subroutine EVAL
```

Programmet læser den x-værdi ind, for hvilken polynomiet skal evalueres efter hver addition, og derefter et (a,p) par ad gangen. Processen standser ved end-of-input (v.h.a. iostat-delen i read-sætningen). Programmet er forsynet med tilstrækkelig mange kommentarer at de enkelte operationer burde blive klare efter lidt studium. For at forenkle processen blev der valgt den simple strategi først at gennemgå hele kæden for at finde ud af, om det nye led skal lægges oven på et gammelt (additionstilfældet), og gennemgangen bliver benyttet samtidigt til at finde det sidste led og at oprette en pegepind til det. Det gør det let derefter at finde ud af (hvis der er mere at gøre), om det nye skal ind foran eller bag efter, det hele. Hvis heller ikke det var tilfældet, er der kun tilbage, at det nye led skal ind et eller andet sted imellem to gamle. Man kunne godt gøre alt dette i en enkelt gennemgang af kæden, men den brugte strategi viste sig noget nemmere at programmere.

Der er yderligere den mulighed, at efter en addition er et led blevet til et nul-led, ved at dets koefficient er blevet lig med nul. Sådan et led skal fjernes. Det tager underprogrammet FJERN_NULLED sig af.

Der var desuden brug for et underprogram (VIS) for at skrive kæden ud. Det format, der blev valgt dertil, var at repræsentere leddene indrammede af [...], med tre data indeni: selve koefficienten, potens, og et T, hvis leddets pegepind peger på et efterfølgende, eller F, hvis det er det allersidste. Og så er der selvfølgelig underprogrammet EVAL, der evaluerer $p(x)$ løbende.

Programmet blev kørt med følgende inputdata:

```
1             ! x-vaerdien for p(x) eval.
1, 1          ! Additioner
3, 3
1, 5
7, 0
1, 3
-1, 1         ! Subtraktioner, toemmer kaeden
-1, 5
-7, 0
-4, 3
eoi
```

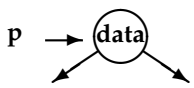
og de afprøver alle de situationer, der er nævnt ovenfor. En kørsel gav udskriften:

```
 x?
 x =   1.000
 Nyt (a,p) par:   1   1
[ 1, 1,F]
 p(x) =       1.000
 Nyt (a,p) par:   3   3
[ 1, 1,T] [ 3, 3,F]
 p(x) =       4.000
 Nyt (a,p) par:   1   5
[ 1, 1,T] [ 3, 3,T] [ 1, 5,F]
 p(x) =       5.000
 Nyt (a,p) par:   7   0
[ 7, 0,T] [ 1, 1,T] [ 3, 3,T] [ 1, 5,F]
 p(x) =      12.000
 Nyt (a,p) par:   1   3
[ 7, 0,T] [ 1, 1,T] [ 4, 3,T] [ 1, 5,F]
 p(x) =      13.000
 Nyt (a,p) par:  -1   1
[ 7, 0,T] [ 4, 3,T] [ 1, 5,F]
 p(x) =      12.000
 Nyt (a,p) par:  -1   5
[ 7, 0,T] [ 4, 3,F]
 p(x) =      11.000
 Nyt (a,p) par:  -7   0
[ 4, 3,F]
 p(x) =       4.000
 Nyt (a,p) par:  -4   3
 Kaeden er tom.
 p(x) =       0.000
```

som forhåbentlig overbeviser læseren om, at det hele virker. Så kunne det være en udfordring for læseren at finde data, som får programmet til at gå ned; der kan godt være nogle fejl tilbage.

Binære træer

En anden meget nyttig datastruktur er et træ, og en af de mest gængse af slagsen er det binære træ. Strukturen er bygget op af enkelte knudeelementer, også kaldt "blade" hvis de er ved enderne, som kan repræsenteres som



Cirklen er en struktur med et antal data i, samt to pegepinde, der peger på andre elementer af samme type. Pegepinden p peger på cirklen. Oftest er det sådan, at den venstre pegepind peger på et element, hvis data i en nærmere defineret forstand er mindre end dem i elementet, og højre pind peger på et, hvor data er større. Det kan f.eks. bruges til meget effektivt at sortere et antal datamængder, som det skal vises i et større eksempel.

Lad os sige, vi vil sortere en række data; vi bruger heltal for enkelheds skyld. Vi vil sortere talrækken (inputdata til programmet)

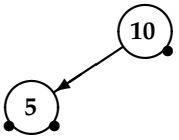
```
5
17
3
1
12
20
4
7
14
```

Processen er at indlæse tallene og sætte dem ind i et træ. Dette foregår på følgende måde. Indgangen til træet er i toppen, pegepinden `start`. Der ligger topbladet (-knuden). Er tallet mindre end knudens tal, går man til den næste knude til venstre, ellers til højre. Det gentager man indtil der ved det sted, hvor man skal gå, ikke er et blad; hvilket man så opretter med tallet. Alt dette kan udtrykkes, for hver indsætning af et tal, som en rekursiv algoritme:

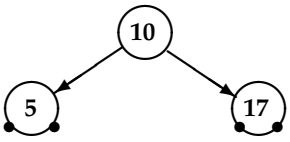
1. Er træt tomt? I så fald starter tallet et nyt; exit.
2. Hvis tallet er mindre end toppen af træet, indsættes det i deltræet til venstre, ellers i deltræet til højre;

Et tomt træ kan her være pegepinden fra et blad, som endnu ikke peger på noget. Det repræsenterer vi som en tyk prik. De ovenstående tal bygger træet op i følgende rækkefølge:

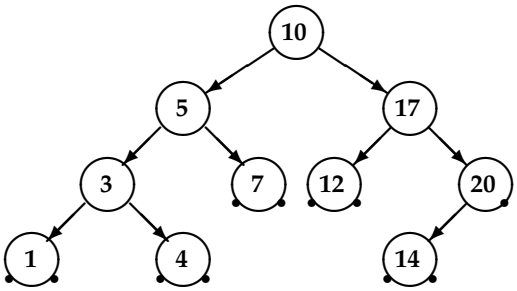
Næste input giver så



hvorefter 17-tallet giver



osv. Det endelige træ, efter alle tallene er blevet sat ind, er:



Efter alle datatal er sat ind i træet, sker der en "trægennemgang" på en bestemt måde, der igen kan beskrives rekursivt. Teknisk kaldes det (på engelsk) for en **in-order traverse**, og fremgangsmåden er, for hele traversen eller gennemgangen:

1. Hvis knuden er tom, gør intet; ellers:
2. Gennemgå træet til venstre for knuden;
3. gør det fornødne med indholdet af knuden;
4. gennemgå træet til højre for knuden.

Vi starter ved toppen, hvor 10-tallet er. Først skal vi gennemgå hele det mindre træ til venstre, hvilket fører os til dets top, knuden med 5-tallet i, og derfra til 3-tallet, derfra til 1-tallet, som ikke har et venstre træ (gør intet). Så bliver 1-tallet skrevet ud (det er det, der skal ske for hvert datatal her; bemærk, at tallet er det allermindste og er det første der bliver skrevet ud). Nu er det træet til højre for 1-tallet, der er næst, men der er ikke noget. Vi er færdig med træet til venstre for 3-tallet, tallet bliver skrevet ud, 4-tallet besøgt og skrevet ud, osv. Processen resulterer i, at alle tal bliver udskrevet i stigende (sorteret) rækkefølge. Her er programmet:

```
module DEFS
  implicit none
  type blad
    integer              :: vaerdi
    type (blad), pointer :: tv, th
  end type blad
end module DEFS

module INTERF
  interface
    recursive subroutine SAET_IND (tal, top)
      use DEFS
      integer              :: tal
      type (blad), pointer :: top
    end subroutine SAET_IND
    recursive subroutine GENNEMGAA (top)
      use DEFS
```

```
      type (blad), pointer :: top
    end subroutine GENNEMGAA
  end interface
end module INTERF

program BIN_TRAE
! Eksempelprogram, der viser et binaert trae.
! Traeet indeholder kun heltal i de enkelte
! blade, samt pegepindene til venstre og
! hoejre. Tallene bliver effektivt sorteret
! i stigende raekkefoelge ved en efter-
! foelgende 'in-order' gennemgang af traeet,
! hvor aktionen, der sker med indholdet her
! bare er en udskrivning af tallet.

  use DEFS;  use INTERF;  implicit none
  integer              :: tal, ios
  type (blad), pointer :: top

  NULLIFY (top)              ! Initialisering
  print '(" Input tallene, afslut med ^d :")'
  do
    read (*,*,iostat=ios) tal
    if (ios /= 0) exit
    call SAET_IND (tal, top)
  enddo
  call GENNEMGAA (top)
end program BIN_TRAE

recursive subroutine SAET_IND (tal, top)
! Hvis TOP for det trae der gaas ind i her er
! tomt, saettes tallet ind der. Ellers gaar
! det ind enten i undertraet til venstre
! (hvis tallet er mindre end topbladets tal),
! ellers til hoejre (hvis det er stoerre).
! Dette foregaar ved rekursion, som standser
! ved et tomt blad.

  use DEFS
  integer              :: tal
  type (blad), pointer :: top

  if (.not. ASSOCIATED(top)) then ! tomt trae
    ALLOCATE (top)             !.. start nyt,
    top%vaerdi = tal        ! .. saet tal ind
    NULLIFY (top%tv);  NULLIFY (top%th) ! nul
  else if (tal < top%vaerdi) then !Tal mindre
    call SAET_IND (tal, top%tv) !ind, venstre
  else                         !ellers hoejre
    call SAET_IND (tal, top%th)
  endif

end subroutine SAET_IND

recursive subroutine GENNEMGAA (top)
! In-order gennemgang af hele traeet. Foerst
! gennemgaas undertraeet til venstre, saa
! bliver  toppens inhold skrevet ud, og til
! sidst gennemgaas det hoejre undertrae.
! Processen standser paa et tomt blad.

  use DEFS
  type (blad), pointer :: top

  if (ASSOCIATED(top)) then
    call GENNEMGAA (top%tv)
    print *, top%vaerdi
    call GENNEMGAA (top%th)
  endif

end subroutine GENNEMGAA
```

Det er interessant at se, hvor kompakt de to procedurer er, i kraft af rekursionen. Især `GENNEMGAA` har kun fem linjer i den del, der laver arbejdet. Resultatet af en kørsel er ikke vist, tallene kom ud i sorteret rækkefølge, som de skulle. Normalt ville et program af denne art handle om noget mere indviklet end heltal, og den behandling, de enkelte elementer underkastes ved gennemgangen, vil normalt være mere end en enkelt udskrivning. Men eksemplet viser det væsentlige.

11.4 Input/output med dynamiske strukturer

Input/output for strukturer er allerede beskrevet i kapitel 10, og der var ingen særlige problemer ved det. Der er imidlertid et problem med dynamiske strukturer, dvs. strukturer der indeholder en eller flere pegepinde til andre strukturer. Det er selvsagt ikke muligt at udskrive sådan en struktur med f.eks. en simpel `print *`. Skulle den opfatte pegepinden som repræsentant for den næste struktur, så kunne det give anledning til en muligvis meget lang stribe `print`s. Det er derfor nødvendigt at skrive nogle programlinjer, der udskriver de relevante komponenter af en sådan struktur enkeltvis.

Kapitel 12
Fossiler og andet

Fortran 90 slæber, af tekniske årsager, en del gamle faciliteter med sig. Det er sprogelementer fra tidligere udgaver af Fortran. Filosofien er, at en oversætter for en ny Fortran-udgave skal være i stand til at kende gamle udgavers faciliteter. Det kalder man **nedad-kompatibilitet**. Det medfører, at en komplet beskrivelse af hver ny udgave også skulle indeholde alle de gamle ting. Mange af disse er imidlertid fossiler og ikke anbefalelsesværdige; de var nødløsninger dengang, og der er mere elegante løsninger nu. Der er dog nogle gamle faciliteter, der, i visse programmeringssituationer, er nyttige at kende. Og så er der nye faciliteter, der ikke er anbefalelsesværdige. Nogle af alle disse beskrives nedenfor, og nogle nævnes blot. Man støder nemlig af og til på gamle programmer, som man er nødt til at forstå, og så kan det være godt at kende nogle af disse gamle fossiler. Desuden er der nye fossiler, dvs., nye faciliteter indført i Fortran som ikke er at anbefale.

12.1 Brugbare gamle faciliteter

STOP

`STOP`-sætningen, allerede beskrevet i kapitel 2, bryder med reglen at hver enhed bør indrettes således, at der ikke er nogen smutveje ud af den. Mens denne regel er fornuftig, når den refererer til `RETURN` (se senere), er der imidlertid argumenter for brugen af `STOP`. Det hænder, at `STOP` er det eneste fornuftige i visse situationer, f.eks. hvis noget går rigtig galt, og der ingen mening er med at fortsætte. Man kan også bruge sætningen under programaflusning for at begrænse, hvor langt et program kører.

DATA

I den nye udgave af Fortran (90) er denne sætning overflødiggjort af, at man kan give startværdier til variabler i samme sætning, hvor de er erklæret. Eksempler er:

```
integer             :: n=100
real, dimension(3) :: x = (/1.0, 1.0, 1.0/)
```

I tidligere Fortran-udgaver måtte der `DATA`-sætninger til, og man kan stadig udtrykke det samme som det ovenstående med dem:

```
integer             :: n
real, dimension(3) :: x
DATA n / 100 /
DATA x / 3*1.0 /
```

hvor `3*1.0` siger at de tre elementer hvert skal få værdien `1.0`. Det kan være lettere at bruge `DATA` i visse situationer, selvom den anses som et fossil. Vil man f.eks. initialisere et todimensionalt array, kan udtrykket blive ret omstændeligt at skrive, fordi initialiseringsværdien jo altid skal være en konstant, og en arraykonstant for to dimensioner skal angives som en anordning af et endimensionalt konstantarray. For eksempel, for et `2*2` array:

```
real :: X(2,2) &
    = RESHAPE((/1.0,1.0,1.0,1.0/), (/2,2/))
```

hvilket ikke er lige så nemt som

```
real :: X(2,2)
DATA X / 4*1.0 /
```

12.2 Ikke-anbefalede gamle og nye faciliteter

GOTO

Denne sætning hører sammen med **labels**, dvs. de numre, man kunne mærke linjer med (og stadig kan), for så at hoppe til dem med en `GOTO`-sætning. Det giver hvad man kalder spaghettiprogrammer, som kan være umulige at finde ud af. Sætningen er helt overflødig nu, med de `IF...THEN...ELSE` strukturer man bl.a. har.

RETURN

I ældre generationer af Fortran var det påkrævet at afslutte alle procedurer med en eller flere `RETURN`-sætninger, og den forklarer sig selv. Den blev brugt af smarte programmører til at smutte ud af en procedure hvor som helst. Kroppen af en procedure kan f.eks. starte med

```
if (...) RETURN
```

Det gør det mindre klart, hvad mulighederne er, og man er bedre tjent med:

```
if (...) then
  ...
else
  ...
endif
```

Alternate RETURN

Alternate RETURN er en ret indviklet måde, i en procedure, at reagere på forskellige mulige situationer. Eftersom `RETURN` i sig selv nu er frarådet, er denne facilitet det også. Den interesserede læser må henvende sig til ældre Fortran-bøger. Formålet med den kan nu langt bedre realiseres ved hjælp af `IF...THEN...ELSE` eller `CASE`-strukturen.

Løkker med labels

I tidligere Fortran var en `DO`-løkke forbundet med en label, som mærkede den sidste sætning i løkken (ofte en `CONTINUE`). Det foregik sådan:

```
      do 10 i = 1, 100
        ...
10    continue
```

og det var endda muligt at stakke flere løkkeniveauer oven på hinanden, alle sluttende ved den samme label:

```
      do 10 i = 1, 100
        do 10 j = 1, 10
           do 10 k = 1, 20
             ...
10    continue
```

Eneste fordel ved dette var, at man let kunne se, hvor en løkke endte. Det er ikke helt så let med den nye måde at gøre det på, f.eks.

```
  do i = 1, 100
    do j = 1, 10
      do k = 1, 20
        ...
      enddo
    enddo
  enddo
```

hvis løkkerne bliver længere end vist her, eller hvis indrykning ikke bliver brugt. Men, for at bøde på det, er der de nye labels, som gør det mere overskueligt:

```
iloop:  do i = 1, 100
jloop:    do j = 1, 10
kloop:      do k = 1, 20
              ...
            enddo kloop
          enddo jloop
        enddo iloop
```

Labels (som linjenumre) er altså blevet helt overflødige.

REAL løkketællere

Som noget nyt blev der indført den mulighed, at bruge typen `REAL` som tælleren i en `DO`-løkke. Hvis vi f.eks. vil bruge værdierne 0.1..1.0 i trin 0.1, *kunne* vi bruge

```
do x = 0.1, 1.0, 0.1
  ...
enddo
```

Det er dog risikabelt, fordi man, p.g.a. afrunding, aldrig kan være sikker på, hvor mange gange løkken udføres. Er f.eks. 1.0 med, eller ej? Hvis regnestykket ender på 1.00001, ville den værdi ikke blive brugt, fordi den er større end 1.0. Eksemplet implementeres med fuld sikkerhed sådan:

```
do i = 1, 10
  x = REAL(i) / 10
  ...
enddo
```

Tællere af typen `REAL` er absolut ikke at anbefale.

DO WHILE

Denne facilitet blev introduceret med Fortran 90, med uden at der var brug for den. Formen er

```
do while (<betingelse>)
  ...
enddo
```

men det kan realiseres lige så nemt ved hjælp af `EXIT`:

```
do
  if (.not.<betingelse>) exit
  ...
enddo
```

(Bemærk at betingelserne i de to former er modsat hinanden). Her er det helt tydeligt, hvornår der exiteres fra løkken. I tilfældet af `DO WHILE` kan man være usikker på, om der er exit ved første sætning i løkken, eller inde i løkken, så snart som betingelsen ikke holder.

Implicit

Vi har brugt sætningen `implicit none` i det meste af bogen. Det er for at undertrykke det, der ellers gælder, at variabler hvis navne begynder med `'a'-'h'` og `'o'-'z'` automatisk er af typen `REAL`, ellers integer, medmindre andet er erklæret. Man kan ændre på dette og selv definere hvordan det skal være. I de gamle dage brugtes den mulighed til at lade alle de navne, der normalt ville blive `REAL`, blive til den (gamle) type `DOUBLE PRECISION`, som heller ikke længere er anbefalet (se nedenfor). Som forklaret i kapitlet om sætninger, er implicitte typer en dårlig idé, og derved `IMPLICIT`-sætningen også.

COMMON, BLOCK DATA, EQUIVALENCE, SEQUENCE

Fortran 90 tillader fortsat brug af `COMMON`, som definerer en blok af variabler, der er tilgængelig for alle de enheder, der erklærer blokken. Man kan lave numre med `COMMON`, f.eks. snyde i en enhed ved at dele et array, erklæret i hovedblokken, op i flere dele under andre navne i en anden enhed. Disse numre fører til snedig og ugennemskuelig programmering. De mere seriøse formål ved `COMMON` er nu tjent langt bedre af modules.

`COMMON`-blokke kunne tildeles startværdier med en speciel form for `DATA`-sætning, `BLOCK DATA`. Har man ikke brug for `COMMON`, så har man heller ikke brug for `BLOCK DATA`.

En anden fidus man havde fornøjelse af engang – før der fandtes `ALLOCATE` – var at genbruge arrayplads ved at overlægge et givet array med et eller flere andre, med sætningen `EQUIVALENCE`. Man kunne ikke bare, f.eks., kalde to halvdele af et array to andre navne, hver af dem repræsenterende en halvdel, men de ækvivalente variabler kunne også have forskellige typer. Det var altså ren genbrug af hukommelsesplads. Alt dette fører imidlertid til ugennemskuelig programmering og derved til den slags fejl, der kan være svære at finde. Alt dette er tilladt i Fortran 90, men er frarådet kraftigt.

Sammen med disse hører den nye facilitet `SEQUENCE`, som skulle sikre, at de enkelte komponenter af strukturer er gemt væk inden for en struktur, i samme rækkefølge, som de er erklærede i erklæringen af strukturtypen. Det kan ellers være, at oversætteren finder på at anordne dem i en anden, mere pladssparende, ordning. Eneste grund til at forhindre dette er, at bruge `EQUIVALENCE`, og det burde man alligevel holde sig fra, og derfor ligeledes fra `SEQUENCE`.

Assumed size

Dette er allerede blevet nævnt i kapitel 8, dvs. overførsel af arrays til procedurer med `*` som længden i en dimension eller flere. Det medfører nogle restriktioner (mod, f.eks., brug af arrayudsnit eller helarrayudtryk) og er derfor ikke at anbefale. Desuden er det let at undgå denne facilitet.

DOUBLE PRECISION

I tidligere Fortran-udgaver kunne man opnå større præcision ved variabler og konstanter ved at erklære dem som `DOUBLE PRECISION`. Konstanter blev ofte glemt; en konstant midt i et program som `1.0` havde således enkelt præcision, uden at man bemærkede det. For at markere dobbelt præcision, skulle man bruge den anden form for "engineering notation", `1.0D00`, hvor D-et, i stedet for E, indikerede den dobbelte præcision. Glemmer man det, kan man reducere et regnestykke til enkelt præcision. Hvis man f.eks. har sætningerne

```
double precision x
...
x = x + 1.0
```

så bliver værdien 1.0 konverteret til dobbelt præcision, men uden garanti for, hvad der fyldes i de sidste mange cifre.

Alt dette er der nu lavet om på, og `DOUBLE PRECISION` er et fossil, et undertilfælde af en af mange mulige `KIND`, og konstanter angives med det `KIND`-vedhæng man ønsker. Har man, f.eks. defineret sig en `KIND`-parameter `dbl`:

```
integer,parameter :: dbl &
                     = SELECTED_REAL_KIND(14)
```

så får man en konstant med den præcision ved at udtrykke den som `1.0_dbl`. Det giver bedre fleksibilitet. Variabler der skal have den præcision, er erklæret med `KIND`-delen specificeret, som vist i kapitel 6.

Fastform-Fortran

Før Fortran 90 var det nødvendigt at holde sig til ret stramme regler for, hvor en sætning og en label må være på hver linje. Labels skulle sidde i position 1-5 på linjen, sætninger skulle starte i position 7 eller efter, og position 6 blev brugt som en indikation af, at linjen er en fortsættelse af den foregående linje. Det var muligt at købe Fortranark med disse positioner fortrykt (der var næppe mange, der brugte dem). Den aktive del af hver linje sluttede i position 72, og alt det, der kom efter, blev overset af oversætteren, så man måtte passe på ikke at gå over de 72.

Denne fastform-måde er stadig i orden, men den må enten bruges i hele programmet eller slet ikke. Der er dog ingen grund i dag til at indskrænke sig på den måde.

12.3 Helt frarådede gamle faciliteter

Her opsummeres der, i forkortet form og uden nøje beskrivelser, andre faciliteter, der er endnu mere forkastelige end de ovennævnte. Hvis læseren vil – af hvilken som helst grund – vide mere om disse ting, kan de findes i ældre bøger om Fortran.

Der er alternative former af `IF`-sætningen, såsom den såkaldte **arithmetic if**, en spredning i flere end to retninger. Den var, lige som **assigned GOTO** og **computed GOTO**, et sandt monster,

årsagen til rigtige ugennemskuelige spaghettiprogrammer, som ikke engang programmøren selv kunne forstå efter nogle uger, og hvor det ofte var helt umuligt at forudse, hvilken vej en programkørsel ville gå.

PAUSE blev brugt til at tillade interaktion mellem programmet og brugeren, og bliver ikke brugt mere. Det, der engang fungerede som det nuværende `IOSTAT` i input/output-sætninger, var de to `ERR=<label>` og `END=<label>` items i en `READ`-sætning. Hvis der altså var en fejl, eller ikke mere input, skulle programmet hoppe til sætningen med den label. Det var en slags `GOTO`. Nu klarer man sig bedre med `IOSTAT`.
Vi har allerede nævnt D-formen af konstanter i afsnittet der handler om `DOUBLE PRECISION`. Der er en tilsvarende formatbeskrivelsesform, D-formen, som brugtes i stedet for E-formen til at indikere dobbelt præcision. Den er overflødig nu, hvor alle `REAL` værdier er af en eller anden `KIND`.

I tidligere Fortran-udgaver var der ikke tekststrenge i den form, vi kender dem nu, tekst indrammet af anførselstegn, som `"This is text"`; der var oprindeligt kun Hollerith-formen, hvor en sådan streng specificeredes som `12HTHIS IS TEXT` (der var heller ingen små bogstaver i Fortran dengang). Man måtte tælle hvor mange tegn der følger efter `H`. Hvis man så derefter ændrede teksten, måtte man passe på at gentælle. Disse strenge blev brugt i formatbeskrivelser, og kan stadig væk bruges i dag, hvis man har lyst. De er dog yderst fossilske.

Til allersidst er der `ENTRY`, hvilket tillader at gå ind i en procedure ved forskellige steder, efter valg. Det var en form for sammenklumpning af hvad der egentligt skulle være flere procedurer, under et navn. Igen, en dårlig idé og ikke at anbefale.

Kapitel 13
Fortran 95

Fortran skal ikke blive statisk, og der er nye impulser hele tiden. Omkring 1995 var der en rapport om ændringer til Fortran 90, som resulterede i dokumentet "Working Draft X3J3/95-007R2" (X3J3 er det udvalg, der tager sig af videreudvikling af Fortran); og senere udkom et officielt dokument om Fortran 95, ISO/IEC 1539-1:1997. Det første af de to siges at indeholde alt det der gik ind i det officielle. Fortran 95 er kun en relativt lille ændring af Fortran 90, og der er nu oversættere til den nye udgave. I dette kapitel beskrives disse ændringer. De består af tre kategorier:

- Slettede sprogelementer, der stadig var med i Fortran 90, men ikke var anbefalede;
- nye forældede sprogelementer, dvs. elementer som fra nu af anses som forældede;
- nye sprogelementer.

13.1 Slettede sprogelementer

De følgende er nu helt ude af Fortran. Op til Fortran 90 er de tilladt, omend ikke anbefalede:

- Løkketællere af typen `real` og `double precision`
- Et hop til en `ENDIF`-sætning fra et sted uden for den `IF...ENDIF`-struktur den afslutter. Dette var tilladt før, selvom det var en mærkelig ting at programmere, med uforudsigelige virkninger.
- `PAUSE`-sætningen
- De to `GOTO`-relaterede, `ASSIGN` og `assigned GOTO`
- Hollerith edit-item, `H...`.

13.2 Nyligt forældede sprogelementer

Disse er sprogelementer som allerede var frarådede, men stadig var (og er) en del af Fortran. Men nu er de officielt erklæret som forældede, hvilket er et signal for, at de givetvis dropper ud af sproget ved den næste ændring (der er tale om Fortran 2000). Nogle af disse er slet ikke nævnt eller beskrevet i denne bog, fordi de rent faktisk har været fossiler i lang tid. De er:

- **Computed GOTO**; det var ikke frarådet af dem, der etablerede Fortran 90 standarden, men er det nu.
- **Statement function**, dvs. functiondefinitioner på en enkelt linje i starten af en enhed.
- Fælles slutning af flere `DO`-løkker, f.eks.

```
do 10 i = 1, n
  do 10 j = 1, m
    do 10 k = 1, l
      ...
10 continue
```

- **DATA**-sætning blandt udførbare sætninger; det er (mærkeligt nok) tilladt, men er nu gået over til status af ikke anbefalet.
- `Character`-functions med **assumed length**. Man *kan* programmere en function, hvis type er `character(len=*)`, men det er nu frarådet; faciliteten burde realiseres ved at bruge en subroutine i stedet.
- Fastform kildetekst; den gamle Fortranmåde, at starte sætninger i position 7, slutte i position 72, osv.
- Brug af `character*..` i erklæringer. Man kan f.eks. erklære et antal af `character`-variabler med forskellige længder i en enkelt erklæring som

```
character*80 :: str1, str2*40, a*1
```

  Den venstre side burde nu altid skrives i formen

```
character(len=80) :: str1, str2*40, a*1
```

  som tjener samme formål.

13.3 Helt nye sprogelementer

De nyindførte sprogelementer består af nogle ændringer i bestående ting, samt to nye intrinsics, `NULL` og `CPU_TIME`.

NULL([arg])
Denne nye function kan erstatte NULLIFY for at definere uassocieret tilstand af en pegepind, med en sætning som

```
peg => NULL()
```

i stedet for

```
NULLIFY (peg)
```

Hvis der er et argument (arg), så resulterer et kald af NULL i en pegepind, som stadig er uassocieret, men kan pege på et target af samme type som arg peger på. Der er en nyttig anvendelse af NULL: som initialisering af en pegepind i en erklæring. I kapitlet om pegepinde blev det nævnt, at en pegepind, efter den er erklæret, er i en udefineret tilstand, hverken associeret eller løs. Før dens status kan afprøves med en function som ASSOCIATED, skal pegepinden nulstilles, hvortil NULLIFY skulle bruges. Glemmer man at sørge for det, kan programmet køre galt. Nu er det muligt at gøre dette i erklæringssætninger, f.eks.

```
real,pointer :: peg => NULL()
```

og det er anbefalet altid at initialisere pegepinde på denne vis.

Auto-deallocate

I Fortran 90 er der et lille problem med lokale allokerede arrays inden for en procedure. Hvis der ikke er en DEALLOCATE-sætning sidst i proceduren, har sådan et array en udefineret tilstand efter programmet har forladt proceduren, medmindre arrayet er erklæret med attributtet SAVE. Det var altså vigtigt, for at have orden med hukommelsesplads, at bruge sætningen. I Fortran 95 er dette nu overflødigt, eftersom sådanne arrays er automatisk deallokeret når programmet går ud af proceduren.

CPU_TIME(TIME)

Der er (mindst) to forskellige slags tid forbundet med en regnemaskine: klokketid og den såkaldte processortid eller **cpu**-tid, hvilken er den tid, maskinen har arbejdet med en bestemt proces, f.eks. et givet program. Maskinen udfører flere processer tilsyneladende samtidigt; i virkeligheden foregår det på den måde, at en proces bliver udført et stykke tid, derefter gemt væk mens en anden proces får lidt tid, osv. indtil de er færdigudført. Den samlede cpu-tid, processoren brugte til en bestemt proces, er mindre end klokketiden under forløbet. Argumentet TIME skal være af typen real og skalar, og det får værdien af processor- eller cputidsforbruget. Det rejser spørgsmålet, fra hvornår tiden bliver målt? Det er et udefineret tidspunkt, og derfor, hvis man vil måle cpu-tid, skal det gøres som differens mellem to tider, f.eks.:

```
call CPU_TIME (t1)   ! t1 er tid op til her
...
call CPU_TIME (t2)   !       tid op til her
t_cpu = t2 - t1      ! cpu mellem de to.
```

Standarden foreskriver at enhederne skal være sekunder. Det er op til den enkelte maskine, hvordan værdien approksimeres, dvs. hvor præcis tidsværdien er. Hvis en maskine ikke er i stand til at levere cpu-tid, afleverer den TIME som negativ værdi for at indikere dette.

Det er normalt sådan, at de enkelte systemer tilbyder andre systemrutiner der måler forskellige tidsforbrug samt andet, og de tillader desuden også ofte at nulstille "klokken". Men disse funktioner er ikke en del af Fortran. Under systemet Unix er der f.eks. rutinerne ETIME og DTIME, og de kan normalt kaldes fra et Fortran-program.

Ændring i intrinsic SIGN

I den intrinsic function SIGN er der en mindre ændring. Den har at gøre med, at i nogle maskiner er der to forskellige slags nul: positiv og negativ nul. Den intrinsic function SIGN(A,B) (se Bilag A) bruger fortegnet af det andet argument, B, til at aflevere enten +|A| eller -|A| som værdi. I Fortran 95 gør den det nu, selv hvis B er nul. Nogle maskiner skelner ikke mellem de to nuller, og hos dem bliver nul regnet som positiv i denne sammenhæng.

Ændring i intrinsic CEILING og FLOOR

I Fortran 90 er der kun et enkelt argument i functionen CEILING, der giver den mindste heltalsværdi større end eller lig med argumentet A, som skal være af typen real. I Fortran 95 er det nu muligt at give et yderligere (valgfrit) argument, KIND, som specificerer den KIND, resultatet skal have; det er altså en af KINDs for heltal (integer), der skal angives.

Analogt til CEILING gælder det samme for FLOOR, som afleverer det heltal, der er mindre end eller lig med argumentet, med et ekstra argument der bestemmer resultatets (integer-) KIND.

Ændring i intrinsic MAXLOC og MINLOC

Disse to intrinsics har, i Fortran 90, to argumenter, f.eks. kaldes den første med MAXLOC(ARRAY,MASK), hvor MASK er valgfrit. Resultatet er positionen af det første element i arrayet, der har den maksimale værdi, muligvis med en restriktion givet af MASK, hvis dette argument er til stede. I Fortran 95

har man nu tilpasset disse to intrinsics til de beslægtede to, `MAXVAL` og `MINVAL`, som begge har tre mulige argumenter, f.eks. `MAXVAL(ARRAY,DIM,MASK)` resulterer i den maksimale værdi af ARRAY, langs dimensionen `DIM`, med restriktionen som i `MASK`. Der er altså indført argumentet `DIM` i både `MAXLOC` og `MINLOC`. Eftersom begge `DIM` og `MASK` er valgfrie, er der det potentielle problem, hvilken af de to er ment, hvis man kun angiver en af dem. Det er imidlertid løst ved at de har to forskellige typer: `DIM` er `integer`, og `MASK` er en betingelse (typen `logical`); og oversætteren kan fortolke argumentet ud fra dette forhold, hvis man kalder enten med `MAXLOC(ARRAY,DIM)` eller `MAXLOC(ARRAY,MASK)`. Hvis de kaldes med alle tre, `MAXLOC(ARRAY,DIM,MASK)` er der ikke et problem, så er det positionen, der bestemmer betydningen. Man kan også bruge keywords.

FORALL

Denne nye sætning tillader visse kompakte formuleringer, og gør det samtidig muligt for visse parallelle processorer at optimere arrayoperationer. For eksempel, hvis vi vil oprette enhedsmatricen i en matrix `A` (erklæret som A(N,N)), ville vi i Fortran 90 kunne gøre det med sætninger som

```
A = 0
do i = 1, N
  A(i,i) = 1
enddo
```

Ved hjælp af den nye sætning kan vi nu erstatte løkken med den enkelte sætning

```
FORALL (i=1:N) A(i,i) = 1
```

Den generelle form er

```
FORALL (<spec>)
  ...
end FORALL
```

hvor <*spec*> betyder et eller flere indeksområder af typen vist i det ovenstående eksempel, der ligner en specifikation af et arrayudsnit; disse er alle af formen `index = start:slut[:trin]` med `trin` valgfrit. Der kan være flere af disse efter hinanden. Til sidst kan der desuden være en maske, som allerede beskrevet i kap. 7, i afsnittet om `WHERE`-sætningen. Her er en mere indviklet `FORALL`, med flere indeksspecifikationer og en maske. Eksemplet er taget ud af Standarden:

```
real :: A(10,10), B(10,10) = 1.0
...
FORALL (i=1:10, j=1:10, B(i,j)/=0.0)
  A(i,j) = REAL(i+j-2)
  B(i,j) = A(i,j) + B(i,j)*REAL(i*j)
end FORALL
```

`FORALL` ligner i vis grad en løkke, men er ikke en løkke; der er intet, der foreskriver i hvilken rækkefølge tilskrivningerne skal foretages. Et af de vigtigste anvendelsesområder - og motiveringen for den nye sætning – er i parallel regning, hvor nogle af de tilskrivninger beskrevet i sætningsblokken bliver udført samtidigt. Det er mest i denne sammenhæng, `FORALL` bliver brugt.

13.4 Nye tiltag i bestående sprogelementer

Den nye standard, Fortran 95, har lagt nogle enkelte faciliteter til bestående sætninger og derved elimineret nogle knaster.

PURE procedurer

Procedurer (functions og underprogrammer) kan nu erklæres som `pure`, hvilket tvinger programmøren til at angive alle overførte argumenters `intent` i kroppen af den pure procedure (undtaget overførte procedurer eller pegepinde), som ellers er valgfrit. Alle argumenter til en pure function skal have `intent(in)`. Der er en del regler forbundet med dette attribut, ud over tvunget `intent`: Lokale variabler i en `pure` procedure må ikke have attributtet `SAVE`; en sådan procedure skal have et explicit interface; hvis en `pure` procedure indeholder interne procedurer, skal de også alle være `pure`; andre procedurer kaldt fra en `pure` procedure skal også være `pure`; `pure` procedurer må ikke indeholde `print`-, `OPEN`-, `CLOSE`-, `BACKSPACE`-, `ENDFILE`-, `REWIND`-, `INQUIRE`- eller `STOP`-sætninger.

ELEMENTAL procedurer

I kap. 8 blev det nævnt, at mange intrinsic functions er **elemental**, hvilket betyder, at hvis de får arrays som argument, bliver resultatet også et array, med operationen udført ved hvert arrayelement. Denne egenskab var i Fortran 90 forbeholdt intrinsic functions, men kan i Fortran 95 også angives for andre procedurer. Attributtet `elemental` sættes, lige som `pure` eller `recursive` ved at nævne det foran specifikationen. En elemental procedure er automatisk også en `pure`, og så gælder alle regler nævnt i ovenstående afsnit om pure procedurer. Det er tilladt også at specificere attributtet `pure`, men ikke nødvendigt, da det følger med. En elemental procedure må ikke være rekursiv; og der skal altid være et explicit interface i (eller kendt i) den enhed, der kalder en elemental procedure.

I praksis foregår det på den måde, at proceduren specificeres med erklæringerne, som om argumenterne var skalarer; men hvis proceduren bliver kaldt med arrays som argument(er), bliver

operationerne udført på alle elementer, og evt. resultater går i tilsvarende arrays. Hvis der er flere arrayargumenter, skal de selvfølgelig være kompatible med hinanden. Her er et lille eksempelprogram, der demonstrerer en function (INV) og en subroutine (ADD), begge elemental:

```
program ELEM_TEST
! Demo elemental function og subroutine.
  implicit none
  real :: x, y, z, A(4), B(4), C(4)

  interface
    elemental real function INV (x)
      real,intent(in) :: x
    end function INV
    elemental subroutine ADD (x, y, z)
      real,intent(in)  :: x, y
      real,intent(out) :: z
    end subroutine ADD
  end interface

  x = 1; y = 2; A = 2

  z = INV(y)
  print '(" INV(2)       =", f6.2)', z
  B = INV(A)
  print '(" INV(2,2,2,2) =", 4f6.2)', B

  call ADD (x, y, z)
  print '(" 1 + 2 =", f6.2)', z
  B = 1
  call ADD (A, B, C)
  print '(" A+B   =", 4f6.2)', C

end program ELEM_TEST

elemental real function INV (x)
   real,intent(in) :: x
   INV = 1 / x
end function INV

elemental subroutine ADD (x, y, z)
   real,intent(in)  :: x, y
   real,intent(out) :: z
   z = x + y
end subroutine ADD
```

og resultatet af en kørsel af dette program er:

```
 INV(2)       =  0.50
 INV(2,2,2,2) =  0.50  0.50  0.50  0.50
 1 + 2 =  3.00
 A+B   =  3.00  3.00  3.00  3.00
```

13.5 Andre mindre tiltag

WHERE

`WHERE`-sætningen er udvidet til

```
WHERE (<betingelse>)
  ...
ELSEWHERE (<betingelse>)
  ...
END WHERE
```

hvor der i Fortran 90 kun var mulighed for `ELSE`.

Specification expressions

Der er et indviklet aspekt af Fortran, kaldt (på engelsk) **specification expression**. Disse udtryk skal være af typen heltal, og bliver brugt til at definere attributter af f.eks. arrays, såsom længder. De kan enten være selve heltal eller udtryk med heltal, men også intrinsic functions med heltalsresultat. Her er et lille eksempel af nogle:

```
subroutine SUBR (a, n)
  integer :: n
  real    :: a(n)

  real    :: loca(n), locb(SIZE(a))
  ...
end subroutine SUBR
```

De to lokale arrays `loca` og `locb` har længder bestemt henholdsvis af argumentet (værdien) `n` og den erklærede længde af arrayet `a`, hvor den intrinsic function `SIZE` bliver brugt. Kort sagt, er det i Fortran 90 kun tilladt at bruge intrinsics til sådan noget, mens det i Fortran 95 er muligt også at bruge andre functions, med visse (logiske) restriktioner. Det drejer sig om, eksempelvis, at de skal have resultat `integer` (eller i visse tilfælde `character`); de skal være elemental.

Strukturkomponenters initialisering

Der er mulighed i Fortran 95 for at initialisere strukturkomponenter i selve strukturdefinitionen. Sådan en typedefinition er i sig selv ikke en erklæring; den/de følger senere i programmet, for de individuelle variabler. For eksempel defineres et polært koordinationspar med strukturen

```
type polar_koord_typ
  real :: vinkel = 0.0
  real :: radius = 0.0
end type polar_koord_typ
```

Når man derefter bruger den type til at erklære nogle variabler med,

```
type (polar_koord_typ) :: x, y
```

så er disse variabler initialiserede med de værdier, der står i definitionen af typen. Det er ikke muligt i Fortran 90. Hvis en komponent er en pegepind, kan også den initialiseres, men, som nævnt tidligere, kun med brug af den nye function `NULL()` og => i stedet for lighedstegnet. Denne initialisering er faktisk kraftigt anbefalet.

Navngivet END INTERFACE

Nogle interfaces, for eksempel de, der styrer operatorfunctions og generiske functions osv., har navne. En interfaceblok termineres med en `end interface` sætning. I Fortran 95 er det tilladt, i modsætning til Fortran 90, at lægge navnet til denne sætning, så det er umiddelbart klart, hvilken interfaceblok det er, sætningen afslutter. Eksempelvis kan man skrive de interfaces vist i kap. 9 som

```
interface ARRAYBYT
  module procedure SGL_BYT
  module procedure DBL_BYT
end interface ARRAYBYT
```

og i operatoreksemplet,

```
interface operator (-)
  module procedure NOT
end interface operator (-)

interface operator (*)
  module procedure AND
end interface operator (*)

interface operator (+)
  module procedure OR
end interface operator (+)
```

- man kan diskutere, hvor meget denne nyhed bidrager til bedre forståelighed af programmer, men eksemplerne viser i hvert fald hvordan man skal gøre.

Garanteret tilstrækkelig feltbredde i edit items

Hvis man specificerer en feltbredde i en formatbeskrivelse, der senere viser sig at være utilstrækkelig, får man de velkendte "`****`" i outputtet, hvilket irriterer. Det kan man nu forhindre i Fortran 95 ved at specificere nul som feltbredde for I, F, B, O og Z-items. Det giver den mindste feltbredde, der er nødvendig for at skaffe plads til de tal, som kommer ud. Man vil altså altid se et tal, hvis man skriver, f.eks., `I0` eller `F0.4`. Det vil samtidig nok gøre det umuligt at udskrive en ordentlig tabel, så man kunne egentligt nøjes med frit format. Men det er altså et nyt tiltag i Fortran 95. Det kan bruges til at forhindre output som

```
Antallet fundet er           13
```

NAMELISTkommentar

En af måderne at læse data ind på er at bruge faciliteten `NAMELIST`, hvor man bruger hvad der ligner tilordningssætninger til at angive datainputs. I kap. 5 var der et eksempel på sådan en inputliste:

```
1, 2, 3
&ijk i = 4, j = 5, k = 6 /
&ijk i = 7, k = 9 /
```

Det nye er, at det er tilladt at indsætte kommentarer i sådan en inputliste, i den samme form man bruger i programmer. Eksemplet kunne f.eks. blive til

```
1, 2, 3
&ijk i = 4, j = 5, k = 6   ! Alle tre er med
/
&ijk i = 7, k = 9          ! Her er j udeladt
/
```

osv. Det er en lille ting, men i givet fald noget nyttigt. Bemærk, at kommentaren skal være det sidste på en linje og at "/"-tegnet, der afslutter listen, derfor skal komme efter (kommentaren må ikke være efter dette tegn).

Kapitel 14

Bitmønstre, afrunding, programdesign

Selvom den måde, information er lagret på i en regnemaskine, ikke formelt hører med til Fortran, er det godt at vide lidt om emnet. Uvidenhed kan blive årsag til visse programmeringsfejl. Afrundingseffekter skal man også være klar over, og de kan ofte minimeres med passende valg (design) af algoritmen til et bestemt problem.

14.1 Bits, bytes, words

En regnemaskine siges at være baseret på binær logik, hvilket betyder, at der er et stort antal steder i maskinen, som kan have en ud af kun to tilstande, som vi kan vælge at give navnene "0" og "1". Disse er de såkaldte **bits**. Bits er de mindste informationsenheder.

En regnemaskine kommunikerer med sig selv (dvs. mellem de enkelte dele inden for maskinen) og med omverdenen. Det betyder, at bits bliver sendt fra et sted til et andet. Denne kommunikation foregår hurtigere, hvis der bliver sendt mere end en enkelt bit ad gangen. Det foregår i parallelle bundter af bits, og antallet af bits i et bundt varierer fra den ene maskine til den anden. Samling af bits kaldes **word**, det engelske ord for "ord". Hvor der tidligere har været maskiner med ordlængden 8, 10, 12, 16, 18, 24, 28, 32 og 60 bits, m.fl., er der nu mest maskiner med 32-bit ord, og en bestemt "supercomputer", der har brugt 60-bit ord i mange år.

Indtil for nogle år siden var computerverdenen mere eller mindre enig om, at 8 bits er det antal, man sender ad gangen mellem regnemaskiner og ydre enheder (som kan være printere eller andre maskiner, osv.). Denne enhed er blevet en slags referencestørrelse, og har navnet **byte**. Selvom kommunikation nu i dag ofte foregår i mere end en byte ad gangen, er byte stadig referencen; for eksempel er størrelsen af en regnemaskines hukommelse udtrykt i antal bytes, kilobytes (kb) eller megabytes (Mb). I hver maskine er der også normalt en blanding af forskellige samlingsstørrelser. For eksempel kan en maskine kommunikere med en anden med 8 bits, mens maskinens ordstørrelse er 32 bits, og i visse regneenheder i maskinens hjerte sågar 64 bits. Hver maskine er dog normalt karakteriseret med dens "ordstørrelse", og man taler om 16-bit eller 32-bit maskiner, osv.

Vi skal nu se lidt nærmere på, hvordan de forskellige værdityper i Fortran 90 er, eller kan være, lagret i maskiner.

14.2 Binære tal

Først noget generelt om talsystemer. Når vi skriver et tal som 127, så er det en slags notation; tallet er faktisk et polynomial udtryk for dets værdi. Tallet betyder $1\times10^2 + 2\times10^1 + 7\times10^0$. Hvis der er decimaler "efter punktummet", for eksempel 127.35, så fortsætter polynomiet med negative potenser af 10, her altså med tilføjelsen $+3\times10^{-1} + 5\times10^{-2}$. Systemet siges at have basen 10, og man kunne vælge en anden base, for eksempel 2. Med 2 som base kan man opbygge tal lige som med 10. Forskellen er, at man har brug for kun to cifre: 0 og 1. Hvert led i et sådant "binært" tal består altså af et 0 eller 1 ganget med 2^p, altså en vægtet sum (for heltal) af $\ldots 8, 4, 2, 1$, hvor i det decimale system, det er en vægtet sum af $\ldots 100, 10, 1$. Et binært tal som 1001 står således for $1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$, eller $8 + 0 + 0 + 1 = 9$.

Lad os antage, at en fiktiv maskine har en ordstørrelse på tre bits, og se hvilke binære mønstre, der kan være i disse ord. Der er 8 forskellige, og de er:

```
000    0
001    1
010    2
011    3
100    4
101    5
110    6
111    7
```

Ved siden af de binære mønstre er der anført nogle mulige decimale tal, de binære kunne stå for, eller ville, ifølge det ovenstående polynomsystem. Disse tal kunne altså være de otte mulige heltal i denne maskine (generelt er der, for en ordstørrelse n, 2^n mulige mønstre eller tal). Der er dog et problem: de er alle positive. For at indikere om

et tal er positivt eller negativt, er man nødt til at beslaglægge en bit; og traditionen er, at man tager den der ligger længst til venstre, altså det mest betydende ciffer, som fortegn. Man fandt på et system, der deler de ovenstående mønstre i to dele, de positive og de negative tal. Desuden er systemet sådan, at lægger man et tal sammen med det samme negative tal ved et binært regnestykke, så er resultatet lig nul, som man jo ville forlange. Under dette system kan vi tabellere den ovenstående gruppe således:

```
000     0
001     1
010     2
011     3
100    -4
101    -3
110    -2
111    -1
```

hvoraf det fremgår, at hvis den mest betydende bit er 0, er tallet positivt, ellers negativt. Det mulige talområde er her $-2^2 \ldots 2^2 - 1$. Generelt er det, for n bits, $-2^{n-1} \ldots 2^{n-1} - 1$. Her følger nu et eksempel af et regnestykke, hvor der lægges 3 sammen med -3. Bemærk reglen i binær regning: $1 + 1 = 10$:

```
      011
+     101
      ---
= (1)000
```

Eftersom ordene kun har 3 bits i denne fiktive maskine, går det førende 1-tal tabt, fordi der ikke er plads til det, og resultatet er, som ønsket, lig nul. Systemet virker efter hensigten.

Ligesom der er decimaltal der udtrykker brøker som 0.35, er der en binær måde ("binaltal") at udtrykke dem på. Som med decimalsystemet fortsætter man bare til højre efter at have sat et punktum, og lægger negative potenser af 2 sammen, sådan at et binært tal er en vægtet sum af $\ldots, 8, 4, 2, 1, \frac{1}{2}, \frac{1}{4}, \ldots$.

14.3 Lagring af de fire grundtyper i rigtige maskiner

Her beskrives der kort hvordan de fire grundtyper: `integer`, `real`, `logical` og `character` er lagret som bitmønstre i maskinernes hukommelse.

Heltal

I dag har mange regnemaskiner ord med 32 bits, men der er stadig nogle med 16. I de 16 bits er der mulighed for i alt $2^{16} = 65536$ forskellige mønstre (tal), og de bliver delt op i positive og negative. Det giver et talområde fra $-2^{15} \ldots 2^{15} - 1$, eller $-32768 \ldots 32767$. Dette er ikke altid tilstrækkeligt, og disse maskiner tilbyder derfor programmøren en sammenkædning af to 16-bits ord til de 32 bits for heltal med flere cifre. Det giver så et større område, $-2^{31} \ldots 2^{31} - 1$ eller $-2147483648 \ldots 2147483647$. Det er som regel nok. Her har der været tale om to forskellige `KIND` heltal (se kap. 6).

REAL tal

Som beskrevet i tidligere kapitler, består `real` tal af to dele. Mennesker kan bedst lide at forestille sig disse tal som en kombination af en brøk ikke langt fra 1, samt en multiplikator, en potens af 10. Et tal som 123.456 ville altså blive skrevet i formen 0.123456×10^3, eller, hvis vi skal have fortegnene med, $+0.123456 \times 10^{+3}$. Hvis vi altid skriver $0.\cdots$ og altid $\times 10^{\cdots}$, så kunne vi jo udelade disse dele for at skaffe mere plads til de væsentlige, for eksempel som | + | 123456 | + | 03 |, hvor vi har tegnet kasser for at vise de separate dele (de fire dele kan ligge i en anden orden end den viste i de enkelte maskiner).

Regnemaskinerne bruger imidlertid det binære system, og basen 10 er derfor (underforstået) erstattet med basen 2 i potensdelen; men ellers er systemet netop det lige viste. Værdier af typen `real` er lagret i denne form, hvor et vist antal n_b bits er givet til brøken og et vist antal n_p til (2-er) potensdelen, samt de to fortegnbits. Der er etableret en vis standard for disse tal (n_b, n_p). Det er klart, at 16 bits ikke er tilstrækkeligt til formålet, så der er få maskiner, der ikke bruger mindst 32 bits til `real` tal. Der er 30 bits tilbage efter de to fortegn, og af dem bliver typisk 23 brugt til brøken, og 7 til eksponenten. Det betyder en præcision i brøken af 23 binære cifre, hvilket svarer til ca. 7 decimaler, og et potensområde $-128 \ldots +128$, vel at mærke som potenser af 2. Omregnet til potenser af 10 svarer det til ca. de ± 38 man kan læse i manualer for den slags maskine. Hvis de ca. 7 decimaler eller potensområdet ikke er nok, kan man kæde ordene sammen til længere enheder, for eksempel 64 bits. I nogle maskiner bliver det så til en 52-bits brøk (ca. 15 decimaler) og en 10-bits eksponent (område $10^{-308} \ldots 10^{+308}$); det er den type `real`, der før hed `double precision`, og nu bare er en af flere `KIND` inden for `reals`.

Problemer ved blanding

Det ovenstående burde nu gøre klart, hvorfor blanding af typer ved aritmetiske operationer kan være problematisk. Hvis et program for eksempel indeholder en linje med

```
  n = i + x
```

med `n` og `i` heltal og `x` `real`, skal der foregå en hel del bag kulisserne for at gøre additionen mulig; sådan en sætning er tilladt af Fortran 90. Variablerne `i` og `x` har helt forskellige strukturer, som de er lagret i maskinen. Det der sker er, at den mindre komplicerede type i en operation (her `i`) bliver lavet om til den samme type som den mere komplicerede (`x`), før regningen udføres. I dette eksempel er resultatet derfor af typen `real`, og værdien skal igen konverteres til et heltal, før den kan lægges ind i heltalsvariablen `n`. Dette er sådan set maskinens problem, ikke så meget programmørens. Det kan dog være interessant at være sig bevidst om processen, især når noget går galt. Denne diskussion forklarer måske også, hvorfor man skal være varsom med et udtryk som `x*3/4`, hvilket vel kan resultere i nul. Ved at skrive udtrykket om til `3*x/4`, kan man undgå problemet.

Men der er nogle værre problemer, forbundet med subroutines. Når et underprogram bliver kaldt med en stribe overførte argumenter, er oversætteren normalt lige glad, om erklæringerne for disse i den kaldende enhed passer sammen med dem i selve underprogrammet. Det er for det meste en programmeringsfejl hvis de ikke passer, som i dette eksempel:

```
  real :: x
  call SUB1 (x)
  ...
END ...
subroutine SUB1 (a)
  integer :: a
  ...
END subroutine SUB1
```

Det kaldende program "mener" at `x` er en `real`, og dens værdi har altså den struktur der svarer til en `real`. Underprogrammet `SUB1` opfatter værdien imidlertid som et heltal, og tager hele den sammensatte `real` struktur som et heltal. Dette kan ikke gå godt. Det fører til fejl, der kan være svære at opspore. I kapitel 8 bliver de såkaldte **interfaces** diskuteret, som sikrer, at det ovenstående program ikke kommer forbi oversætteren.

Logical lagring

Typen `logical` er i sig selv noget binært, eftersom den kun kan have to mulige værdier (`.false.` og `.true.`). De kan repræsenteres ved hjælp af en enkelt bit. Af tekniske årsager er det imidlertid sådan, at maskiner normalt bruger hele ord til at lagre logiske værdier. Det kan virke som spild af plads i maskinens hukommelse. Hvis man i et program har store arrays af `logicals`, er det et naturligt ønske at lagre dem bitvis. Det ville jo være godt, hvis der var en `KIND` af `logical`, der gør dette, men det er ikke krævet af standarden, så det kan man ikke være sikker på at få. Værre er, at selv hvis en given maskine tilbyder denne såkaldte bit-packing, holder det givetvis ikke for en anden maskine, man muligvis vil transportere sit program til. Man kan omgå problemet ved at lave sin egen `logical`, ved at manipulere med de enkelte bits i, for eksempel, heltal. Der er intrinsics, der tillader dette.

Tegnlagring

Man forbinder regnemaskiner, af naturlige grunde, med talregning. Men som vi har set, er tegn og tegnstrenge også en vigtig del af, hvad man forlanger af en computer. Tegn skal også lagres i maskinen. Det kræver en bestemt mapping af alle de tegn, man kan tænke sig, til bestemte bitmønstre. Man besluttede i første omgang, at 128 forskellige er nok, og det kan være i en byte, som har 256 mulige mønstre. Ved at bruge kun halvdelen gjorde man det muligt at bruge en bit til at teste, om kommunikation mellem maskiner eller dele af en maskine foregår uden fejl. Der er et antal systemer for den ovennævnte mapping, dvs. hvilke mønstre står for hvilke tegn. Det system, der i dag er den klare vinder, er ASCII-systemet. Man kan forestille sig de 128 mønstre som heltal, og tale om hvilket tal står for hvilket tegn. I Bilag B er der en tabel over tegnene i deres tal-position i ASCII-tabellen.

14.4 Afrunding

Fra afsnittet om lagring af `real` tal bliver det klart, at der næsten altid vil opstå en lille fejl i disse tal, ved at kæden af cifrene er begrænset i dens længde. Man kender problemet fra decimaltal; vi kan skrive et tal som 0.1 eksakt, men et tal som skal repræsentere brøken 1/3 vil i praksis approksimeres med et endeligt antal 3'ere. Der er mange brøker der har gentagne ciffergrupper, og det samme gælder i det binære talsystem. Netop tallet 1/10, som vi kan skrive eksakt som decimaltal, bliver til en uendelig række cifre som binært tal. Det bliver lavet om til formen 0.8×2^{-3} og hvis man regner på det, finder man ud af, at brøken 0.8 bliver til et gentaget mønster af grupper 1100. Systemet bruger kun de første 23 (i nogle maskiner) og resultatet er at nogle cifre går tabt. Udfører man en regneoperation med flere tal, er resultatet altså behæftet med en lille fejl, som kaldes **afrunding**. Maskinen vil nemlig normalt runde op eller ned på en passende måde.

Der er nogle enkle regler man må være opmærksom på for at modvirke afrundingseffekter (alle handlende om `real` tal):

- Lægger man to tal sammen, er resultatet ueksakt hvis de to er vidt forskellige i størrelse.
- Trækker man et tal fra et andet, er resultatet ueksakt, hvis de to tal er af nogenlunde samme størrelse.

Det er ikke altid muligt at gøre noget ved det (se i næste afsnit hvordan man sommetider kan), men det hjælper at være klar over effekterne. Her er et eksempel. Vi vil beregne et derivat af en funktion $f(x)$ som den velkendte approksimation (en såkaldt centraldifferens), $(f(x+\delta x) - f(x-\delta x))/2\delta x$. Normalt har man en ide om, hvad $f(x)$ bliver, og problemet er nu at vælge en passende værdi for δx. Er den for stor, er approksimationen for grov; men er den for lille, bliver subtraktionen $(f(x+\delta x) - f(x-\delta x)$ ueksakt, fordi de to tal rykker tæt på hinanden. Det blev afprøvet med $f(x) = exp(x)$, og med mindre og mindre δx, i rækken $0.1, 0.01, \ldots$. Argumentet x blev sat til 1, og resultatet blev evalueret som forhold mellem approksimationen og den kendte løsning; det burde være lig med præcist 1. Her er resultatet:

```
LOG10(dx)      approks/eksakt
  -1            1.001668
  -2            1.000013
  -3            1.000017
  -4            1.000324
  -5            1.004271
  -6            0.964802
  -7            1.315639
  -8            0.000000
```

Regningen bliver tydeligt mere eksakt når vi går fra 0.1 ned til 0.01, men derefter bliver det værre igen, og når δx går under 10^{-7} er der slet ingen forskel mere på de to tal, der trækkes fra hinanden. Dette er indlysende, i betragtning af, at $f(x)$ for funktionen ikke er langt fra 1 (den er 2.7..) og den har en præcision af ca. 10^{-7}.

Problemet med addition af to meget ulige størrelser illustreres her med det følgende lille program:

```
program ADD

  implicit none

  integer :: i
  real    :: x, delta=1.0E-08

  x = 1
  do i = 1, 1000
    x = x + delta
  enddo
  print *, " Sum =", x

  x = 0
  do i = 1, 1000
    x = x + delta
  enddo
  x = x + 1
  print *, " Sum =", x
end program ADD
```

Resultatet er:

```
 Sum =   1.000000
 Sum =   1.000010
```

Den første sum viser overhovedet ingen ændring fra 1, efter (tilsyneladende) 1000 additioner af 10^{-8}. Det skyldes, at hver gang der lægges `delta` til 1, gør det ingen forskel overhovedet. I anden løkke lægges de 1000 `delta` sammen først, og derefter summen til 1, og det kan mærkes. Dette illustrerer, at det er en god ide af overveje algoritmen i visse situationer, hvilket der skal vises flere eksempler på i næste afsnit.

14.5 Valg af udtryk

Nogle gange er det muligt at formulere et bestemt udtryk på flere, matematisk identiske, måder, hvor nogle af dem er bedre egnet til regning end andre. Et eksempel er de to ligninger

$$a = \sqrt{1+x^2} - 1$$

$$a = \frac{x^2}{1+\sqrt{1+x^2}}$$

som er matematisk de samme. Men direkte oversat til Fortran ville de blive til henholdsvis

```
a = SQRT(1+x**2) - 1
```

og

```
a = x**2 / (1 + SQRT(1+x**2))
```

Hvis vi har med små `x`-værdier at gøre, er det første udtryk uheldigt; først lægges der en lille værdi (x^2) til 1, og så trækkes der 1 fra noget, der er tæt på 1. Det første af de problemer er ikke løst i den anden formulering, men den har i hvert fald ingen subtraktion mere, og viser sig at være en tydelig forbedring.

Et andet eksempel hentes fra et udtryk, som Archimedes brugte til at indkredse værdien af π. Ideen er, at forstille sig en trekant inden for

kredsen, og doble antallet af sider til en sekskant, tolvkant, osv., mens man hele tiden lægger sidelængderne sammen for at approksimere kredsens omfang. For et polygon med n sider, hver side af længde k_n, bliver sidelængderne til k_{2n} når antallet af sider dobles til $2n$, og formlen for de nye sidelængder er

$$k_{2n}^2 = 2\left(1 - \sqrt{1 - \left(\frac{k_n}{2}\right)^2}\right)$$

og en direkte oversættelse til Fortran vil igen have de samme to problemer som det sidste udtryk. Der er en anden måde, matematisk identisk med ligningen:

$$k_{2n} = \frac{k_n}{\sqrt{2 + \sqrt{4 - k_n^2}}}$$

og den viser sig at være langt bedre i numerisk forstand.

Et sidste eksempel er udtrykket $1 - cos(x)$; for små værdier af x bliver det unøjagtigt, og i disse tilfælde ville det ækvivalente udtryk $2sin^2(\frac{x}{2})$ være bedre.

14.6 Valg af algoritme

Hvis man vil opsummere en langsomt konvergerende serie, er det godt at tage mange led med i summen. Her opstår der det problem, at leddene bliver små, og når de lægges til en løbende sum, kan der snige sig en unøjagtighed ind pga. addition af to meget ulige tal. Som eksempel kan nævnes serieekspansionen for $\pi/4$,

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

Det første, der nok falder programmøren ind er en simpel løkke der går fremad de n led (n er det totale antal terme, inklusive 1-tallet), f.eks.

```
sum = 1              ! Sum, og
fortegn = -1.0       ! .. fortegn begge REAL
do i = 3, 2*n-1, 2
  sum = sum + fortegn/i
  fortegn = - fortegn
enddo
```

En lille ændring af algoritmen forbedrer summeringen, ved at man starter ved den lille ende:

```
sum = 0              ! Sum, og
fortegn = -1.0       ! .. fortegn begge REAL
do i = 2*n-1, 3, -2
  sum = sum + fortegn/i
  fortegn = - fortegn
enddo
sum = sum + 1
```

Her skal man bare sørge for, at n er et lige tal, ellers går det galt med fortegnene. Denne simple ændring kan give betydeligt bedre resultater.

Bilag A
Intrinsics

Her beskrives alle indbyggede procedurer i Fortran 90 (og 95) i detaljer. De viste argumentnavne er dem, der bruges ved keywordkald, hvor man kan ændre rækkefølgen (se kapitel 4, afsnit 4.11). Argumenter i firkantede parenteser [..] er valgfrie. Der noteres om procedurer er **elemental** og **generic** (se kapitel 4). Ordet "model" henviser til type og `kind`, taget sammen, eller bitmønstret i lageret. De generiske functions har bagvedliggende typespecifikke (angivet i parenteser). For eksempel ligger der `ABS`, `CABS`, `DABS` og `IABS` bag den generiske, som hedder `ABS`. De bagvedliggende kan alle kun modtage argumenter med den rette type (`CABS`: kompleks; `IABS`: heltal; osv.). Det er ikke nødvendigt at kende alle disse bagliggende ikke-generiske, og derfor er de ikke detaljeret her. I eksemplerne er der nogle arraykonstanter, der går igen. De defineres her:
`B`= $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$; `C`= $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$; `P`= $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. Argumentet `MASK` betyder altid et logisk array (eller arrayudtryk). For eksempel fås der sådan et array fra en logisk sammenkobling af to arrays; hvis vi tager de ovenfor definerede arrays, giver operationen `B /= C` → $\begin{bmatrix} .true. & .false. & .false. \\ .true. & .false. & .true. \end{bmatrix}$. Hvor `DIM` er nævnt, er det en dimension i et array, og giver normalt et arrayresultat. For eksempel, hvis `B` og `DIM=1` er argumenter, menes der de tre udsnit $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $\begin{bmatrix} 3 \\ 4 \end{bmatrix}$ og $\begin{bmatrix} 5 \\ 6 \end{bmatrix}$ i `B` hver især, det vil sige alle udsnit langs `DIM`.

A.1 Fortran 90 intrinsics

ABS (A)

Absolutværdi. Elemental, generic (`ABS`, `CABS`, `DABS`, `IABS`). A: `integer`, `real`, `complex`. **Resultat**: samme model som `A`, for `A integer` og `real`, men `real` for `complex A`.
Eksempler: `ABS(3.0,4.0)` → `5.0`; `ABS(-1)` → `1`.

ACHAR (I)

Det tegn der svarer til position `I` i ASCII-tabellen. Elemental. I: `integer`. **Resultat**: `character`.
Eksempel: `ACHAR(55)` → `"7"`.

ACOS (X)

Arccosinus. Elemental, generic (`ACOS`, `DACOS`). `X`: `real` ($|X| \leq 1$). **Resultat**: `REAL`, $-\pi/2 \leq$ `ACOS` $\leq \pi/2$.
Eksempel: `ACOS(0.54030231)` → `1.0`.

ADJUSTL (STRING)

Flytter strengen til venstre og lægger blanktegn til, til højre. Elemental. `STRING`: tegnstreng. **Resultat**: tegnstreng.
Eksempel: `ADJUSTL("  abc")` → `"abc  "`.

ADJUSTR (STRING)

Flytter strengen til højre og lægger blanktegn til, til venstre. Elemental. `STRING`: tegnstreng. **Resultat**: tegnstreng.
Eksempel: `ADJUSTR("abc  ")` → `"  abc"`.

AIMAG (Z)

Imaginær komponent. Elemental. `Z`: `complex`. **Resultat**: `real`.
Eksempel: `AIMAG(2.0,3.0)` → `3.0`.

AINT (A[,KIND])

Nedrunding af `A`. Elemental. `A`: `real`. **Resultat**: `real` og med `KIND` hvis den er til stede, ellers af samme `KIND` som `A`.
Eksempler: `AINT(2.7)` → `2.0`; `AINT(-2.7)` → `-2.0`.

ALL (MASK[,DIM])

`Logical` skalar hvis kun `MASK` er til stede; er `.true.` hvis alle elementer i `MASK` er `.true.`, ellers .false.. Er `DIM` til stede, er resultatet et `logical` array, med så mange elementer som der er udsnit langs `DIM`.
Eksempler:
`ALL((/.true., .false., .true./))` → `.false.`;
`ALL(B/=C)` → `.false.`;
`ALL(B/=C,1)` → `(.true., .false., .false.)`;
`ALL(B/=C,2)` → `(.false., .false.)`.

ALLOCATED (ARRAY)

Inquiry function. `ARRAY`: hvilken som helst type, allokerbar. **Resultat**: `logical`; `.true.` hvis `ARRAY` er allokeret, ellers `.false.`. Resultatet er udefineret hvis `ARRAY` er i udefineret tilstand.

ANINT (A[,KIND])

Afrunding af A. Elemental. A: real; KIND: integer. **Resultat**: real, af KIND hvis den er til stede, ellers samme KIND som A.
Eksempler: ANINT(2.51) → 3.0;
ANINT(-2.51) → -3.0.

ANY (MASK[,DIM])

Logical skalar hvis kun MASK er til stede; .true. hvis nogle elementer i MASK er .true.. Er DIM til stede, er resultatet et logical array, med så mange elementer som der er udsnit langs DIM.
Eksempler:
ANY((/.true., .false., .true./)) → .true.;
ANY(B/=C) → .true.;
ALL(B/=C,1) → .true., .false., .true.;
ALL(B/=C,2) → .true., .true..

ASIN (X)

Arcsinus. Elemental. Generic (ASIN, DASIN). X: real. **Resultat**: real, $-\pi/2 \leq$ ASIN(X) $\leq \pi/2$.
Eksempel: ASIN(0.84147098) → 1.0.

ASSOCIATED (POINTER[,TARGET])

Inquiry function. POINTER: pegepind; TARGET: target eller pegepind. **Resultat**: logical (skalar). Uden TARGET er resultatet .true. hvis POINTER er associeret, ellers .false.. Er TARGET til stede, er resultatet .true. hvis POINTER er associeret med TARGET, ellers .false. Er TARGET selv en pegepind, er resultatet .true. hvis POINTER er associeret med samme target som TARGET er, ellers .false.

ATAN (X)

Arctangens. Elemental. Generic (ATAN, DATAN). X: real. **Resultat**: real, $-\pi/2 \leq$ ATAN(X) $\leq \pi/2$.
Eksempel: ATAN(1.5574077) → 1.0.

ATAN2 (Y,X)

Arctangens af koordinat (X,Y) eller komplekstallet $x + iy$. Elemental. Generic (ATAN2, DATAN2). Y,X: real. I modsætning til ATAN muliggør denne function vinkler fra 0 til π og også den anden halvdel, $-\pi$ til 0. **Resultat**: real. Hvis Y er nul, må X ikke være nul.
Eksempler: ATAN2(1.0,1.0) → $\pi/4$;
ATAN2(1.0,-1.0) → $3\pi/4$;
ATAN2(-1.0,-1.0) → $-3\pi/4$;
ATAN2(-1.0,1.0) → $-\pi/4$.

BIT_SIZE (I)

Antallet af bits. I: integer. **Resultat**: integer.

BTEST (I,POS)

Tester om bitten i position POS i argumentet er sat (lig med 1). Elemental. I, POS: integer. **Resultat**: logical. POS er talt fra højre og starter med nul.
Eksempel: (8,3) → .true. (8 er binært 1000).

CEILING (A)

Det nærmeste heltal $\geq$ A. Elemental. A: real. **Resultat**: integer.
Eksempler: CEILING(1.3) → 2;
CEILING(-1.3) → -1.

CHAR (I[,KIND])

Det tegn der svarer til position I i processorens tegntabel (collating table). Elemental. I: integer. **Resultat**: character. Eksempler vil være afhængige af processorens tegntabel. Hvis den er ens med ASCII-tabellen (som er tilfældet ved de fleste systemer), er resultatet det samme som ved ACHAR, bortset fra, at hvis KIND er til stede, har resultatet denne tegn-KIND.

CMPLX (X[,Y][,KIND])

Konverterer til typen complex. Elemental. X: integer, real eller complex; Y: integer eller real. Er X complex, må Y ikke være til stede. **Resultat**: complex. Der er tre muligheder:
Kun X er til stede, ikke complex: resultatet er $x + i.0$;
Kun X er til stede, og er complex: resultatet er x;
Både X og Y er til stede, begge ej complex: resultatet er $x + iy$.
Bemærk: hvis ikke KIND er angivet, er resultatet default real, uanset hvilke typer X eller Y er.
Eksempler: CMPLX(-1) → (-1.0,0.0);
CMPLX((1.0,2.0)) → (1.0,2.0);
CMPLX(1.0,2.0) → (1.0,2.0).

CONJG (Z)

Kompleks-konjugat. Elemental. Z: complex. **Resultat**: complex.
Eksempel: CONJG((1.0,2.0)) → (1.0,-2.0).

COS (X)

Cosinus. Elemental. Generic (COS, CCOS, DCOS). X: real eller complex. **Resultat**: real eller complex.
Eksempel: COS(1.0) → 0.54030231.

COSH (X)

Hyperbolsk cosinus. Elemental. Generic (COSH, CCOSH, DCOSH). X: real. **Resultat**: real.
Eksempel: COSH(1.0) → 1.5430806.

COUNT (MASK[,DIM])

Antallet af `.true.` elementer i arrayet `MASK`. **Resultat**: `integer`, skalar hvis `DIM` ikke er til stede. Er `DIM` med, er resultatet et array, hvert element svarende til det arrayudsnit af `MASK`, langs `DIM`.
Eksempler: `COUNT((.true.,.false.,.true.))` → 2.
`COUNT(B/=C)` → 3; `COUNT(B/=C,1)` → 2,0,1; `COUNT(B/=C,2)` → 1,2.

CSHIFT (ARRAY,SHIFT[,DIM])

Udfører en hel cirkulær shift på et 1-D array, eller på et 1-D udsnit af et flerdimensionalt array, langs `DIM`. `ARRAY`: hvilken som helst type, ikke skalar. `SHIFT`: `integer` array med et antal dimensioner én mindre end `ARRAY` har (skalar hvis `ARRAY` er endimensional). `SHIFT` bestemmer antallet af shifts, til venstre (til højre for `SHIFT` < 0). **Resultat**: et nyt array.
Eksempler:
`CSHIFT((/1,2,3,4,5/),2)` → 3,4,5,1,2.

`CSHIFT(P,-1,2)` → $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$.

DATE_AND_TIME ([DATE][,TIME][,ZONE][,VALUES])

Subroutine. `DATE`, `TIME`, `ZONE`: tegnstrenge med henholdsvis mindst 8, 10 og 5 tegn. `VALUES`: `integer` array med mindst 8 elementer. Mindst et argument skal være til stede. Resultaterne for de tre tegnargumenter står i den venstre del af strengene. Resultaterne er systemklokkens tid og dato.
`DATE` bliver til `ccyymmdd`, (`cc`: århundredet; `yy`: året; `mm`: måneden; `dd`: dagen). Hvis systemet ikke kender datoen, er resultatet blank.
`TIME` bliver til `hhmmss.sss` (`hh`: timen; `mm`: minutterne; `ss.sss`: sekunder ned til msec). Hvis systemet ikke kender tiden, er resultatet blank.
`ZONE` bliver til tidszonen i form af ±`hhmm`, den tid (afstand), i timer og minutter, der skal lægges til Greenwich Mean Time (GMT) for at få den lokale tid.
Argumentet `VALUES` er en alternativ måde at kalde subroutinen på. Benytter man `VALUES`, er resultatet: `VALUES(1)`: årstal.; `VALUES(2)`: månedstal; `VALUES(3)`: dagstal i måneden; `VALUES(4)`: tidsafstand i minutter fra GMT; `VALUES(5)`: timetal, 0-23; `VALUES(6)`: minuttal, 0-59; `VALUES(7)`: sekundtal, 0-60; `VALUES(8)`: millisekundtal, 0-999. I alle 8 tilfælde er resultatet `-HUGE` eller nul, hvis systemet ikke kender resultatet.
Eksempler: hvis vi har erklæret

```
character(len=10) :: str
integer           :: val(10)
```

giver de følgende kald resultaterne i `str` (eller `VALUES`) (vi befinder os i Danmark, og det er kl. 14:15:00 den 1.2.1999):
`CALL DATE_AND_TIME(DATE=str)` → '19990201'.
`CALL DATE_AND_TIME(TIME=str)` → '141500.000'.
`CALL DATE_AND_TIME(ZONE=str)` → '+0100'.
`CALL DATE_AND_TIME(VALUES=val)` → 1999,2,1,1,14,15,0,0.

DBLE (A)

Konverterer til `double precision`. Elemental. Generic (`COS`, `CCOS`, `DCOS`). `A`: `integer`, `real` eller `complex`. **Resultat**: `double precision` (`real`).
Eksempel: `DBLE(1.0)` → `1.0D00`.

DIGITS (X)

Antal cifre i `X`, ifølge dens model. `A`: `integer` eller `real`, skalar eller array. **Resultat**: `integer`, antallet af binære cifre (`X integer`), eller antallet af binære cifre i brøken af et `real X`.
Eksempel: For en normal 32-bit maskine, `DIGITS(1)` → 31 (fortegnet er ikke med) og `DIGITS(1.0)` → 24.

DIM (X,Y)

Differencen `X-Y` hvis $X > Y$, ellers nul. Elemental. `X`, `Y`: begge `integer` eller begge `real`. **Resultat**: samme model som argumenterne.
Eksempler: `DIM(3,1)` → 2; `DIM(-3.0,1.0)` → `0.0`.

DOT_PRODUCT (VECTOR_A,VECTOR_B)

Prikprodukt. Vektorerne: `integer`, `real`, `complex` eller `logical`, endimensionale arrays. **Resultat**: skalar; af samme model som argumenterne. Hvis disse er `logical`, er resultatet det givet ved `ANY(VECTOR_A.and.VECTOR_B)`.
Eksempel:
`DOT_PRODUCT((/1,2,3/),(/2,3,4/))` → 20.

DPROD (X,Y)

`Double precision` produkt `X*Y`. Elemental. `X`, `Y`: `real`. **Resultat**: `double precision`.
Eksempel: `DPROD(2.0,3.0)` → `6.0D00`.

EOSHIFT (ARRAY,SHIFT[,BOUNDARY][,DIM])

"End-off" shift af `ARRAY` eller et udsnit af det. `ARRAY`: hvilken som helst type, ikke skalar. `SHIFT`: `integer`; `BOUNDARY`: samme type som `ARRAY` med dimension en mindre end dimensionen af `ARRAY`. `DIM`: `integer`. **Resultat**: samme type og dimensionalitet som `ARRAY`, eller af den givet af `DIM`. Er `DIM` til stede, er resultatet et endimensionalt udsnit af et flerdimensionalt array, langs `DIM`. `SHIFT` er antallet af shifts til venstre (højre for `SHIFT`<0)).

Der tabes et antal SHIFT elementer ved en ende, og ved den anden ende bliver enten nuller (blanktegn) indført, eller den størrelse angivet i BOUNDARY, hvis det er til stede. For logical arrays er default BOUNDARY .false. og for tegnarrays er det blanktegn.
Eksempler:
EOSHIFT((/1,2,3,4,5/),2) → 3,4,5,0,0.

$$\texttt{EOSHIFT(P,-1,2,99)} \rightarrow \begin{bmatrix} 99 & 1 & 2 \\ 99 & 4 & 5 \\ 99 & 7 & 8 \end{bmatrix}.$$

EPSILON (X)

Det positive tal, som er næsten negligibelt i forhold til 1-tallet af samme model som X; dvs., den relative opløsning for argumentet. X: real, skalar eller array. **Resultat**: samme model som X.
Eksempel: EPSILON(99.9) → 2^{-23}

EXP (X)

Eksponentialfunktion. Elemental. Generic (EXP, CEXP, DEXP). X: real eller complex. **Resultat**: real eller complex.
Eksempel: EXP(1.0) → 2.7182818.

EXPONENT (X)

Eksponentdelen af argumentet. Elemental. X: real. **Resultat**: integer, og svarer til den binære potens i det bitmønster dannet af X. Hvis X er lig nul, er resultatet nul.
Eksempler: EXPONENT(1.0) → 1 (1.0 er lagret som 0.5×2^1), og EXPONENT(4.1) → 3 (4.1 = 0.5125×2^3).

FLOOR (A)

Største heltal $\leq$ A. Elemental. A: real. **Resultat**: integer.
Eksempler: FLOOR(1.999) → 1. FLOOR(-3.7) → -4.

FRACTION (X)

Brøkdelen af X. Elemental. X: real. **Resultat**: real.
Eksempler: FRACTION(1.0) → 0.5 (1.0 er lagret som 0.5×2^1), og FRACTION(3.0) (0.75×2^3) → 0.75.

HUGE (X)

Størst muligt tal af samme model som X. X: integer eller real, skalar eller array. **Resultat**: real.
Eksempler: HUGE(1.0) → $(1-2^{-24})\times2^{127}$.
HUGE(99) → 2147483647 = 2^{31}.

IACHAR (C)

Positionen af tegnet C i ASCII-tabellen. Elemental. C: character. **Resultat**: integer.
Eksempel: IACHAR("X") → 88.

IAND (I,J)

Bitvis I.and.J. Elemental. I, J: integer, begge af samme KIND. **Resultat**: integer, samme KIND som I og J. Den logiske operation er udført på de enkelte bits i argumenterne og giver et resultat for hvert bitpar.
Eksempler: IAND(1,1) → 1; IAND(1,0) → 0. IAND(1,3) (=binære 001 og 011) → 1 (= 001).

IBCLR (I,POS)

Nulstiller bit POS i I. Elemental. I: integer. **Resultat**: integer, lig med I med bit nr. POS udslettet. POS tælles fra højre, startende med 0.
Eksempel: IBCLR(14,1) (= binær 1110) → 12 (1100).

IBITS (I,POS,LEN)

Udtager en række bits. Elemental. I, POS, LEN: integer; LEN $>$ 0. **Resultat**: integer. Det er et heltal dannet af de LEN bits fra POS til POS-LEN+1. POS tælles fra højre, startende med nul.
Eksempel: IBITS(14,1,3) (binær 1110) → 7 (111).

IBSET (I,POS)

Sætter en bit (til 1). Elemental. I, POS: integer. **Resultat**: integer, lig med I med bit nr. POS sat til 1. POS tælles fra højre, startende med 0.
Eksempel: IBSET(12,1) (= binær 1100) → 14 (1110).

ICHAR (C)

Positionen af tegnet C i processorens tegntabel. Elemental. C: character. **Resultat**: integer.
Eksempel: ICHAR("X") → 88 (hvis processoren har samme tegntabel som ASCII).

IEOR (I,J)

Bitvis ekslusiv .or., I $\oplus$ J. Elemental. I, J: integer. **Resultat**: integer. Eksklusiv .or. giver resultat 1 hvis enten en bit i I eller den tilsvarende bit i J er sat (til 1), og 0 hvis de begge er sat eller begge er nul.
Eksempler: IEOR(1,0) → 1; IEOR(1,1) → 0; IEOR(0,0) → 0; IEOR(12,10) (binære 1100 og 1010) → 6 (binær 0110).

INDEX (STRING,SUBSTRING[,BACK])

Startposition af, hvor SUBSTRING er indeholdt i STRING. Elemental. STRING, SUBSTRING: tegnstrenge. BACK: logical. **Resultat**: integer. Hvis BACK mangler eller er til stede med værdien .false., er resultatet startposition af SUBSTRING i STRING. Hvis BACK er til stede med værdien .true., er resultatet det størst mulige, eller søgningen foretaget

baglæns. Hvis understrengen ikke er fundet, er resultatet nul.
Eksempler: `INDEX("IDG FORLAG","G")` → 3; `INDEX("IDG FORLAG","G",.true.)` → 10; `INDEX("IDG FORLAG","Z")` → 0.

INT (A[,KIND])

Konverterer til heltal. Elemental. `A`: `integer`, `real` eller `complex`. **Resultat**: `integer`; den nedrundede heltalsværdi af `A`. Hvis `A` < 0, går nedrunding mod nul. Hvis `A` er `complex`, tages kun den reelle del.
Eksempler: `INT(3.7)` → 3; `INT(-3.7)` → -3.

IOR (I,J)

Bitvis `I.or.J`. Elemental. `I`, `J`: `integer`. **Resultat**: `integer`. Resultatet er 1 hvis enten en bit i `I` eller den tilsvarende bit i `J`, eller begge, er sat (til 1).
Eksempler: `IOR(1,0)` → 1; `IOR(1,1)` → 1; `IOR(0,0)` → 0; `IOR(12,10)` (binære 1100 og 1010) → 14 (1110).

ISHFT (I,SHIFT)

Bitvis (logisk) shift. Elemental. `I`, `SHIFT`: `integer`. **Resultat**: `integer`. `SHIFT` er til venstre, til højre for `SHIFT`<0. Bits tabes i den ene ende, og nuller går ind i den anden ende.
Eksempel: `ISHFT(6,-1)` (binær 110) → 3 (011).

ISHFTC (I,SHIFT[,SIZE])

Bitvis cirkulær shift af de `SIZE` højrestående bits. Elemental. `I`, `SHIFT`, `SIZE`: `integer`. **Resultat**: `integer`. `SHIFT` er til venstre, til højre for `SHIFT`<0. Et antal `SIZE` bits til højre bliver flyttet om `SHIFT` positioner, på cirkulær vis. `SIZE` skal være positiv og mindre end antal bits i `I`. Hvis `SIZE` ikke er til stede, betyder det alle `BITSIZE(I)` bits i modellen. Bits til venstre for de `SIZE` bits er uberørt.
Eksempel: `ISHFT(5,1,3)` (binær 000101) → 3 (binær 000011).

KIND (X)

`KIND`-værdien af `X`. `X`: hvilken som helst type. **Resultat**: `integer`, systemafhængigt.

LBOUND (ARRAY[,DIM])

Nedre indeksgrænse af `ARRAY` (i dimension `DIM`). `ARRAY`: Hvilken som helst type, ikke skalar, ikke en uassocieret pegepind eller et uallokeret array. `DIM`: `integer`. **Resultat**: `integer`, og er den nedre indeksgrænse af `ARRAY` som erklæret. Hvis `DIM` ikke er til stede, er resultatet et array med så mange dimensioner som `ARRAY` har; Er `DIM` med, skalar.
Eksempler: For erklæringen `A(0:10,-5:5)`, `LBOUND(A)` → 0,-5, og `LBOUND(A,2)` → -5.

LEN (STRING)

Længden af strengen, som erklæret. `STRING`: tegnstreng. **Resultat**: `integer`.
Eksempel: For erklæringen `character(len=10) :: S`, `LEN(S)` → 10.

LEN_TRIM (STRING)

Længden af strengen, uden blanktegn i halen. Elemental. `STRING`: tegnstreng. **Resultat**: `integer`, og de blanktegn der kommer efter det sidste ikke-blank tegn til højre tæller ikke med.
Eksempler: `LEN_TRIM(" XY Z ")` → 5; `LEN_TRIM(" ")` → 0.

LGE (STRING_A,STRING_B)

Test for, om strengen `_A` er leksikalsk ≥ `_B`, i ASCII-tabellen. Elemental. `STRING_A`, `STRING_B`: tegnstrenge. **Resultat**: `logical`. Hvis strengene er ulige i længden, bliver den kortere forlænget med blanktegn under testen. Resultatet er `.true.` hvis de to strenge er lig med hinanden, eller hvis der i strengen `_A` er et tegn der kommer efter det tilsvarende tegn i ASCII-tegntabellen; ellers `.false.` Tegnene længst til venstre tæller mest. Hvis strengene indeholder ikke-ASCII tegn, er resultatet systemafhængigt. Hvis de to strenge har nul-længde, er resultatet `.true.`
Eksempler: `LGE("en","to")` → `.false.`; `LGE("en","EN")` → `.true.`; `LGE("en","en ")` → `.true.`

LGT (STRING_A,STRING_B)

Test for, om strengen `_A` er leksikalsk > `_B`, i ASCII-tabellen. Elemental. `STRING_A`, `STRING_B`: tegnstrenge. **Resultat**: `logical`. Hvis strengene er ulige i længden, bliver den kortere forlænget med blanktegn under testen. Resultatet er `.true.` hvis der i strengen `_A` er et tegn der kommer efter det tilsvarende tegn i ASCII-tegntabellen; ellers `.false.` Tegnene længst til venstre tæller mest. Hvis strengene indeholder ikke-ASCII tegn, er resultatet systemafhængigt. Hvis de to strenge har nul-længde, er resultatet `.false.`
Eksempler: `LGT("en","to")` → `.false.`; `LGE("en","EN")` → `.true.`; `LGE("en","en ")` → `.false.`

LLE (STRING_A,STRING_B)

Test for, om strengen `_A` er leksikalsk ≤ `_B`, i ASCII-tabellen. Elemental. `STRING_A`, `STRING_B`: tegnstrenge. **Resultat**: `logical`. Hvis strengene er ulige i længden, bliver den kortere forlænget

med blanktegn under testen. Resultatet er `.true.` hvis de to strenge er lig med hinanden, eller hvis der i strengen `_A` er et tegn der kommer før det tilsvarende tegn i ASCII-tegntabellen; ellers `.false.` Tegnene længst til venstre tæller mest. Hvis strengene indeholder ikke-ASCII tegn, er resultatet systemafhængigt. Hvis de to strenge har nul-længde, er resultatet `.true.`
Eksempler: `LLE("en","to")` → `.true.`; `LLE("en","EN")` → `.false.`; `LLE("en","en ")` → `.true.`

LLT (STRING_A,STRING_B)

Test for, om strengen `_A` er leksikalsk $<$ `_B`, i ASCII-tabellen. Elemental. `STRING_A`, `STRING_B`: tegnstrenge. **Resultat**: `logical`. Hvis strengene er ulige i længden, bliver den kortere forlænget med blanktegn under testen. Resultatet er `.true.` hvis der i strengen `_A` er et tegn der kommer før det tilsvarende tegn i ASCII-tegntabellen; ellers `.false.` Tegnene længst til venstre tæller mest. Hvis strengene indeholder ikke-ASCII tegn, er resultatet systemafhængigt. Hvis de to strenge har nul-længde, er resultatet `.false.`
Eksempler: `LGE("en","to")` → `.true.`; `LGE("en","EN")` → `.false.`; `LGE("en","en")` → `.false.`

LOG (X)

Naturlig logaritme. Elemental. Generic (`ALOG`, `CLOG`, `DLOG`). `X`: `real` eller `complex`. **Resultat**: samme model som `X`. Hvis `X` er `real`, skal $X > 0$.
Eksempel: `LOG(10.0)` → `2.3025851`.

LOG10 (X)

Logaritme, base 10. Elemental. Generic (`ALOG10`, `DLOG10`). `X`: `real` eller `complex`. **Resultat**: samme model som `X`. Hvis `X` er `real`, skal $X > 0$.
Eksempel: `LOG(10.0)` → `1.0`.

LOGICAL (L,KIND)

`KIND`-konvertering af `L`. Elemental. `L`: `logical`; `KIND`: `integer`. **Resultat**: `logical`. Konverterer `L`, som kan være af en af (muligvis) flere `kinds`, til den indikeret af `KIND`.

MATMUL (MATRIX_A,MATRIX_B)

Matrixmultiplikation. `MATRIX_A` og `_B`: hvilken som helst numerisk type eller `logical` (begge den samme type), med en eller to dimensioner. **Resultat**: Numerisk, hvis de to arrays er numeriske, og af samme type; `logical` hvis arrays er det. Antal dimensioner er givet af dem for de to arrays, efter reglerne for matrixmultiplikation. Hvis en af de to arrays er endimensional, skal den anden være todimensional. Er `MATRIX_A` en vektor, antages den som "vandret", mens hvis `MATRIX_B` er en vektor, antages den som "lodret".
Eksempler:
Hvis `A` = $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 4 \end{bmatrix}$, `B` = $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 4 \end{bmatrix}^T$; `X` er vektoren (1,2) og `Y` er vektoren (1,2,3):
`MATMUL(A,B)` → $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$;
`MATMUL(X,A)` → (5,6,11) og `MATMUL(A,Y)` → $\begin{bmatrix} 14 \\ 18 \end{bmatrix}$.

MAX (A1,A2,A3...)

Maksimumværdi. Elemental. `A1...`: `integer` eller `real`, alle af samme model. **Resultat**: samme model som argumenterne.
Eksempel: `MAX(-9,7,2)` → `7`.

MAXEXPONENT (X)

Maksimum exponent (af 2) `X` kan have. `X`: `real`.
Resultat: `integer`.
Eksempel: `MAXEXPONENT(1.0)` → `127` for `real X` i en normal 32-bit maskine. Svarer til ca. 10^{38}.

MAXLOC (ARRAY[,MASK])

Indekser for det største element. `ARRAY`: `integer` eller `real`, ikke skalar. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: `integer`, endimensional array. Resultatet er det første "koordinat" i `ARRAY` med den største værdi. Hvis `MASK` er til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.`
Eksempler: `MAXLOC(B)` → `2,3`; `MAXLOC(B,B<6)` → `1,3`. Søgningen tæller indekser fra 1, uanset indeksgrænserne.

MAXVAL (ARRAY[,DIM][,MASK])

Største element. `ARRAY`: `integer` eller `real`, ikke skalar. `DIM`: `integer`. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: samme model som `ARRAY`. Resultatet er det største element i arrayet. Er `DIM` til stede, er resultatet et array, en værdi for hvert arrayudsnit langs dimensionen `DIM`. Er `MASK` til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.`
Eksempler: `MAXVAL(B)` → `6`; `MAXVAL(B,MASK=B<6)` → `5`; `MAXVAL(B,DIM=1)` → `2,4,6`.

MERGE (TSOURCE,FSOURCE,MASK)

Elemental. `TSOURCE`, `FSOURCE`: hvilken som helst type, begge af den samme. `MASK`: `logical`, konform med de to andre. **Resultat**: Samme model som de to sources. Functionen afleverer et array, hvis elementer er fra `TSOURCE` hvor `MASK` er `.true.` og fra `FSOURCE` hvor `MASK` er `.false.`
Eksempel: `MERGE(B,C,B>C)` → $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$.

MIN (A1,A2,A3...)

Minimumsværdi. Elemental. `A1...`: `integer` eller `real`, alle af samme model. **Resultat**: samme model som argumenterne.
Eksempel: `MIN(-9,7,2)` → `-9`.

MINEXPONENT (X)

Minimum exponent (af 2) `X` kan have. `X`: `real`.
Resultat: `integer`.
Eksempel: `MINEXPONENT(1.0)` → `-126` for `real X` i en normal 32-bit maskine. Svarer til ca. 10^{-38}.

MINLOC (ARRAY[,MASK])

Indekser for det mindste element. `ARRAY`: `integer` eller `real`, ikke skalar. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: `integer`, endimensional array. Resultatet er det første "koordinat" i `ARRAY` med den mindste værdi. Er `MASK` til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.`
Eksempler: `MINLOC(B)` → `1,1`; `MINLOC(B,B>3)` → `2,2`. Søgningen tæller indekser fra 1, uanset indeksgrænserne.

MINVAL (ARRAY[,DIM][,MASK])

Mindste element. `ARRAY`: `integer` eller `real`, ikke skalar. `DIM`: `integer`. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: samme model som `ARRAY`, array. Resultatet er det mindste element i arrayet. Er `DIM` til stede, er resultatet et array, en værdi for hvert arrayudsnit langs dimensionen `DIM`. Er `MASK` til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.`
Eksempler: `MINVAL(B)` → `1`; `MINVAL(B,MASK=B>3)` → `4`; `MINVAL(B,DIM=1)` → `(1,3,5)`.

MOD (A,P)

Restfunktion. Elemental. Generic (`AMOD`, `DMOD`). `A`, `P`: `integer` eller `real`, begge af samme model. **Resultat**: Samme model som `A` og `P`. Hvis $P \neq 0$, er resultatet lig med resten efter divisionen A/P. Hvis $P = 0$, er resultatet afhængigt af processoren.
Eksempler: `MOD(3.0,2.0)` → `1.0`; `MOD(-8,5)` → `-3`; `MOD(8,-5)` → `3`; `MOD(-8,-5)` → `-3`.

MODULO (A,P)

Modulusfunktion. Elemental. `A`, `P`: `integer` eller `real`, begge af samme model. **Resultat**: Samme model som `A` og `P`. Hvis $P \neq 0$ er resultatet værdien R defineret ved $A = Q \times P + R$, hvor Q er et heltal, og $0 \leq R < P$ for $P > 0$, $P < R \leq 0$ for P<0. Hvis $P = 0$, er resultatet afhængigt af processoren.
Eksempler: `MODULO(8,5)` → `3`; `MODULO(-8,5)` → `2`; `MODULO(8,-5)` → `-2`; `MODULO(-8,-5)` → `-3`.

MVBITS (FROM,FROMPOS,LEN,TO,TOPOS)

Flytter en bitsekvens fra et sted til et andet i ordet. Elemental subroutine. Alle argumenter: `integer`, og kun `FROM` og `TO` må være <0. Alle undtagen `TO` har `INTENT(IN)`. Resultatet ligger i `TO`. Et antal `LEN` bits i `FROM`, fra bit `FROMPOS` (tælling går fra højre, startende med nul), kopieres over i `TO` fra bitposition `TOPOS`.
Eksempel: Hvis `TO` i begyndelsen er `6` (binær `110`), resulterer kaldet `CALL MVBITS(7,2,2,TO,0)` i at `TO` = 5, da de to bits fra position 2 og 3 i `FROM` = 7 (binær `0111`, altså `01`) bliver kopieret i `TO` i position 0 og 1, hvilket giver `101` eller 5.

NEAREST (X,S)

Det nærmeste tal forskelligt fra `X`, i retningen bestemt af `S`. Elemental. `X` og `S`: `real`, og $S \neq 0$. **Resultat**: Samme model som `X`. Functionen giver udsagn om opløsningen omkring `X`, eller dens præcision. `S` giver retningen; hvis $S \geq 0$, går det i positiv retning, ellers i negativ retning.
Eksempel: `(3.0,2.0)` → $3 + 2^{-22}$ i normale maskiner.

NINT (A[,KIND])

Afrunding af `A`. Elemental. `A`: `real`; `KIND`: `integer`.
Resultatet: `integer`, af `KIND` hvis den er til stede, ellers default `integer`.
Eksempler: `NINT(2.51)` → `3`; `ANINT(-2.51)` → `-3`.

NOT (I)

Bitvis logisk komplement (omvending). Elemental. `I`: `integer`. **Resultat**: `integer`. Hver bit i `I` bliver vendt om.
Eksempel: Binær `(1001)` → `0110`.

PACK (ARRAY,MASK[,VECTOR])

Pakker arrayet, under kontrol af `MASK`, i det endimensionale array `VECTOR`. `ARRAY`: hvilken som helst type, ikke skalar; `MASK`: `logical` array (eller skalar), konform ned `ARRAY`; `VECTOR`: endimensional, samme model som `ARRAY`. `VECTOR` skal have mindst lige så mange elementer som `ARRAY`. **Resultat**: Samme model som `ARRAY` og er et endimensionalt array. Antallet af dets elementer er afhængigt af argumenterne. Hvis `VECTOR` er til stede, er resultatet af samme længde som `VECTOR`, ellers er antallet antallet t af `.true.` elementer i `MASK`, medmindre `MASK` er skalar og `.true.`, hvor det bliver det samme antal som antallet af elementer i `ARRAY`. Resultatvektorens element i er det element taget fra `ARRAY` hvor det tilsvarende element i `MASK` er det i'te `.true.` element. Tælling går i standard arrayrækkefølge. Hvis `VECTOR` er til stede og $n > t$, har element i værdien af `VECTOR`(i), for $i = t + 1, \ldots, n$.

Eksempler: Hvis M er arrayet $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$ og V er (/2,4,6,8,10,12/), giver PACK(M,MASK=M>0) vektoren 9,7, og PACK(M,M>0,V) vektoren 9,7,9,7,6,8,10,12.

PRECISION (X)

Decimalpræcision af X. X: real eller complex, givetvis et array. **Resultat**: integer. Resultatet er afhængigt af processoren og giver antallet af decimaler i brøkdelen af X. For en normal 32-bitsmaskine med 23 binære bits i brøken af reals, er resultatet 6.

PRESENT (A)

Tester om argumentet A er til stede. A: hvilken som helst type, men skal være erklæret OPTIONAL i enheden hvor functionen kaldes. **Resultat**: logical skalar. Det er .true. hvis A er til stede, ellers .false.

PRODUCT (ARRAY[,DIM][,MASK])

Produktet af elementerne i ARRAY. ARRAY: integer, real eller complex, ikke skalar. DIM: integer, skalar. MASK: logical og konform med ARRAY. **Resultat**: samme model som ARRAY, skalar hvis DIM ikke er til stede, ellers et array. Hvis DIM ikke er til stede, er resultatet produktet af alle de elementer i ARRAY hvor MASK er .true. Er DIM til stede, er resultatet et array af produkterne af udsnit af ARRAY langs den dimension, igen med MASK som restriktion.
Eksempler: PRODUCT((/1,2,3/)) → 6;
PRODUCT(B,MASK=B<4) → 6;
PRODUCT(B,1) → 2,12,30.

RADIX (X)

Basen af X. X: integer eller real, skalar eller et array. **Resultat**: integer, skalar. Resultatet er den base, der ligger til grund for bitmønstret af argumentet. I praksis betyder det tallet 2 for de fleste maskiners vedkommende.

RANDOM_NUMBER (HARVEST)

Et tilfældigt tal, ("pseudorandom"). Elemental subroutine. HARVEST: real, skalar eller array. **Resultatet** går i selve HARVEST, er dannet af en generator af tilfældige tal, og ligger mellem nul og 1.0, med en lige fordeling i dette område. Hvis HARVEST er et array, bliver hele arrayet fyldt med forskellige tal i de enkelte elementer. Normalt er det sådan at når denne function er kaldt et antal gange, bliver den samme sekvens af tilfældige tal genereret for hver ny kørsel af programmet.

RANDOM_SEED ([SIZE][,PUT][,GET])

Genstarter eller afspørger generatoren for tilfældige tal brugt ved et kald af RANDOM_NUMBER. Subroutine. Argumenterne er alle integer. SIZE skal være skalar, mens de to andre skal være endimensionale arrays. Subroutinen kan kaldes uden argumenter, i hvilket tilfælde den resætter generatoren af tilfældige tal i processoren. Er den kaldt med SIZE, sættes generatoren så med denne værdi, som bestemmer antallet af cifre processoren bruger til at opbevare det nuværende tilfældige tal. Jo større SIZE, jo finere en opløsning er der i fordelingen af tallene mellem nul og 1. PUT er et array med længde > SIZE, og kan bruges til at sætte et bestemt tilfældigt tal, det såkaldte seed. GET giver det array, der svarer til det nuværende seed.

RANGE (X)

Decimaleksponentområde for X. X: integer, real eller complex, skalar eller et array. **Resultat**: integer. For et heltal er resultatet lig med antallet af cifre i det største heltal af den KIND, eller INT(LOG10(HUGE)). For X real, er resultatet den mindre af de to absolutte værdier af eksponenterne af henholdsvis HUGE og TINY for denne model.
Eksempel: For en normal 32-bitsmaskine, er RANGE for et heltal ca. 10, og ca. 38 for et real argument.

REAL (A[,KIND])

Konverterer til real. Elemental. A: integer, real eller complex. KIND: integer. **Resultat**: real, af KIND hvis det er til stede, ellers af default kind. **Resultat**: real for integer og real A. For complex A er resultatet den reelle del af argumentet.
Eksempler: REAL(3) → 3.0; REAL(1.0,2.0) → 1.0.

REPEAT (STRING,NCOPIES)

Sammenkædning af strengen, et antal gange. STRING: tegnstreng; NCOPIES: integer, skalar. **Resultat**: tegnstreng. Den er den originale streng, gentaget NCOPIES gange.
Eksempel: REPEAT('ho',3) → 'hohoho'.

RESHAPE (SOURCE,SHAPE[,PAD][,ORDER])

Laver et array med specificeret form fra elementer af kildearrayet. SOURCE: hvilken som helst type, et array. Hvis PAD ikke er til stede, skal SOURCE være af størrelsen mindst lig med PRODUCT(SHAPE). SHAPE: integer, et endimensionalt array. Dens størrelse skal være positiv og mindre end 8. PAD: samme type som SOURCE, og et array. ORDER: integer og samme form som SHAPE. Det skal være en permutation af sekvensen (1,2,...,n); hvis

ORDER ikke er til stede, er det som om det var lig med sekvensen (1,2,...,n). **Resultat**: Et array med den samme form som beskrevet af SHAPE og af samme type som SOURCE. Elementerne i SOURCE, et efter det andet, taget i arrayorden, bliver lagt i resultatet, som er af formen givet af SHAPE, og eventuelle ekstra elementer i resultatet bliver fyldt med PAD sekvensen. Hvis PAD ikke er til stede, skal resultatet have lige så mange elementer som SOURCE.
Eksempler: Lad `V = (/1,2,3,4,5,6/)`.
`RESHAPE(V,(/2,3/))` $\rightarrow \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, og
`RESHAPE(V,(/2,4/),(/99,99/),(/2,1/))` $\rightarrow$
$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 99 & 99 \end{bmatrix}$.

RRSPACING (X)

Reciprok af den relative afstand mellem to nærmeste tal nær X. Elemental. X: real. **Resultat**: real, samme model som X.
Eksempel: RRSPACING(3.0) $\rightarrow 0.75\times2^{24}$ for normale 32-bit processorer.

SCALE (X,I)

Giver $x\times b^i$, hvor b er basen for X, dvs. 2. Elemental. X: real; I: integer. **Resultat**: som X.
Eksempel: SCALE(3.0,2) $\rightarrow$ 12.0, eller 3.0×2^2.

SCAN (STRING,SET[,BACK])

Scanner strengen for tegnene i SET. Elemental. STRING, SET: tegnstrenge, samme KIND; BACK: logical. **Resultat**: integer. Er BACK ikke til stede og STRING indeholder mindst et tegn i SET, er resultatet positionen af tilfældet længst til venstre. Er BACK til stede, bliver resultatet den position der er længst til højre. Hvis intet tegn i SET er i STRING, er resultatet nul.
Eksempler: SCAN('FORTRAN','TR') $\rightarrow$ 3;
SCAN('FORTRAN','TR',BACK=.true.) $\rightarrow$ 5;
SCAN('FORTRAN','XYZ') $\rightarrow$ 0

SELECTED_INT_KIND (R)

KIND-tal for den integer-type med R cifre. R: integer. **Resultat**: integer. Resultatet er det heltal der specificerer den KIND ud af integer-typerne, der har mindst R cifre i det størst mulige tal i typen. Hvis der ikke er en KIND med præcist dette antal cifre, bliver den næstbedste valgt. Hvis dette ikke er muligt, er resultatet -1. Resultatet er systemafhængigt.

SELECTED_REAL_KIND ([P][,R])

KIND-tal for den real-type med P decimalers præcision og R som maksimal eksponent (base 10). P og R: integer. Mindst en af de to skal være til stede. **Resultat**: integer. Resultatet er det heltal der specificerer den KIND ud af real-typerne, der har mindst P decimalers præcision (hvis til stede), og en største eksponent (base 10) R. Hvis der ikke er en KIND med præcist disse specifikationer, bliver den næstbedste valgt. Hvis P ikke er mulig, er resultatet -1; hvis R ikke er mulig, er resultatet -2; hvis de begge er umulige, -3. Resultatet er systemafhængigt.

SET_EXPONENT (X,I)

Det tal der har samme brøk som X og potensdel lig med I. Elemental. X: real; I: integer. **Resultat**: real. Potensdelen (eksponenten i X) bliver erstattet med I.
Eksempel: (3.0,1) (repræsenteret som 0.75×2^2) $\rightarrow$ 1.5 (repræsenteret som 0.75×2^1).

SHAPE (SOURCE)

Form (antal dimensioner) af arrayet SOURCE. SOURCE: array (eller skalar) af hvilken som helst type. Det må ikke være en uassocieret pegepind eller et uallokeret array. **Resultat**: integer endimensionalt array. Resultatet er antallene af dimensioner 1, 2, ... i SOURCE. Der ses bort fra indeksgrænserne.
Eksempel: SHAPE(A(2:5,-1:1)) $\rightarrow$ 4,3. SHAPE(1) $\rightarrow$ et array med en dimension 1, og længden nul.

SIGN (A,B)

$|A|$ forsynet med fortegnet af B. Elemental. A, B: integer, real, begge samme model. **Resultat**: samme model som A og B.
Eksempel: SIGN(-3,2) $\rightarrow$ 3.

SIN (X)

Sinus. Elemental. Generic (SIN, CSIN, DSIN). X: real eller complex. **Resultat**: real eller complex.
Eksempel: SIN(1.0) $\rightarrow$ 0.84147098.

SINH (X)

Hyperbolsk sinus. Elemental. Generic (SINH, CSINH, DSINH). X: real. **Resultat**: real.
Eksempel: SINH(1.0) $\rightarrow$ 1.1752012.

SIZE (ARRAY[,DIM])

Længden af arrayet (i den angivne dimension). ARRAY: array af en hvilken som helst type, men ikke en uassocieret pegepind eller et uallokeret array. DIM: integer. **Resultat**: integer skalar. Hvis DIM ikke er til stede, er resultatet det totale antal elementer i arrayet, som det er erklæret; ellers antallet af elementer i den angive dimension DIM.
Eksempler: SIZE(A(2:5,-1:1),2) $\rightarrow$ 3;
SIZE(A(2:5,-1:1)) $\rightarrow$ 12.

SPACING (X)

Den absolutte afstand mellem to nærmeste tal nær `X`. Elemental. `X`: `real`. **Resultat**: `real`, samme type som `X`. Hvis `X` ikke er lig nul, er resultatet det at lægge et ciffer til det sidste (binære) ciffer i brøkdelen af tallets repræsentation i maskinen; dvs. opløsningen (præcisionen) af `X`. Hvis `X` er nul, er resultatet det samme som `TINY(X)`.
Eksempel: `SPACING(3.0)` $\rightarrow 2^{-22}$ for normale 32-bit processorer.

SPREAD (SOURCE,DIM,NCOPIES)

Spreder et array over et nyt array med flere dimensioner. `SOURCE`: array af hvilken som helst type; `DIM`: `integer`; `NCOPIES`: `integer`. **Resultat**: array af samme type som `SOURCE` og af et antal dimensioner en flere end antallet for `SOURCE`. Det vil sige, at `SOURCE` bliver et udsnit af det ny array, i hvilket der ligger `NCOPIES` langs dimension `DIM`, af udsnittet. Hvis `SOURCE` er skalar, har resultatet et antal elementer lig med `MAX(NCOPIES,0)`.
Eksempel: `SPREAD((/2,3,4/),1,2)` $\rightarrow \begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$.

SQRT (X)

Kvadratrod. Elemental. Generic (`CSQRT`, `DSQRT`, `SQRT`). `X`: `real` eller `complex`. Hvis `X` er `real`, skal det være ≥ 0. **Resultat**: samme model som `X`.
Eksempler: `SQRT(4.0)` $\rightarrow$ `2.0`; `SQRT((-3.0,4.0))` $\rightarrow$ `(1.0,2.0)`.

SUM (ARRAY[,DIM][,MASK])

Summen af elementerne i `ARRAY`. `ARRAY`: `integer`, `real` eller `complex`, ikke skalar. `DIM`: `integer`, skalar; `MASK`: `logical` og konform med `ARRAY`. **Resultat**: samme type som `ARRAY`, skalar hvis `DIM` ikke er til stede, ellers et array. Hvis `DIM` ikke er til stede, er resultatet summen af alle de elementer i `ARRAY` hvor `MASK` er `.true.` Er `DIM` til stede, er resultatet et array af summerne af udsnit af `ARRAY` langs den dimension, med `MASK` som restriktion.
Eksempler: `SUM((/2,3,4/))` $\rightarrow$ `9`; `SUM(B,MASK=B<4)` $\rightarrow$ `6`; `SUM(B,1)` $\rightarrow$ `3,7,11`.

SYSTEM_CLOCK ([COUNT][,COUNT_RATE][,COUNT_MAX])

Heltalsdata for real-time-systemuret. Subroutine. Alle tre argumenter: `integer` skalar, og alle har `INTENT(OUT)`. Hvis `COUNT` er til stede, får den værdien af processorens ur, eller `HUGE(0)` hvis der ikke er et ur. Der bliver lagt et til urets tal for hvert urtik, indtil maksimalværdien er opnået, hvorefter uret resættes til nul ved næste tælling. Hvis `COUNT_RATE` er til stede, processorurets tiks per sekund, eller nul hvis der ikke er et ur. Hvis `COUNT_MAX` er til stede, processorurets maksimumværdi, før det bliver nulstillet. Nul, hvis der ikke er et ur.

TAN (X)

Tangens. Elemental. Generic (`TAN`, `CTAN`, `DTAN`). `X`: `real` eller `complex`. **Resultat**: `real` eller `complex`.
Eksempel: `TAN(1.0)` $\rightarrow$ `1.5574077`.

TANH (X)

Hyperbolsk tangens. Elemental. Generic (`TANH`, `CTANH`, `DTANH`). `X`: `real`. **Resultat**: `real`.
Eksempel: `TANH(1.0)` $\rightarrow$ `0.76159416`.

TINY (X)

Det mindste positive tal med samme model som `X`. `X`: `real`, skalar eller et array. **Resultat**: samme model som `X`, skalar.
Eksempel: `TINY(1.0)` $\rightarrow 2^{-127}$ for normale 32-bits processorer.

TRANSFER (SOURCE,MOLD[,SIZE])

Resultatet er fortolkning af bitsekvensen `SOURCE`, som om den var af typen `MOLD`. `SOURCE`: hvilken som helst type, skalar eller et array; `MOLD`: hvilken som helst type, skalar eller et array; `SIZE`: `integer`, skalar. **Resultat**: samme type som `MOLD`. Hvis `MOLD` er skalar og `SIZE` ikke er til stede, er resultatet skalar. Hvis `MOLD` er et array og `SIZE` ikke er til stede, er resultatet et array med en dimension. I dette tilfælde bliver så meget af `SOURCE` som muligt fortolket som værende af typen `MOLD`, og eventuelle overskydende elementer danner noget der kan være delvis udefineret. Hvis `SIZE` er til stede, er resultatet et array med en dimension og længden lig med `SIZE`. Der bliver i dette tilfælde et antal `SIZE` enheder af typen `MOLD` fortolket. Ideen er her at bruge de givne bitmønstre som de forskellige typer er lagret i og omfortolke deres betydning.
Eksempler: For en normal 32-bits processor, kan `TRANSFER(1082130432,0.0)` give fortolkningen (altså fra `integer` til `real`) `4.0`; og hvis `byte` er erklæret som en `integer` med `SELECTED_INT_KIND(2)`, giver `TRANSFER("0",byte)` værdien 48.

TRANSPOSE (MATRIX)

Transponering. `MATRIX`: hvilken som helst type, todimensionalt array. **Resultat**: samme model, den transponerede matrix.
Eksempel: `TRANSPOSE(B)` $\rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$.

TRIM (STRING)

Skærer blankhalen bort. `STRING`: tegnstreng. **Resultat**: tegnstreng. Den er forkortet til den uden de ekstra blanktegn til højre i `STRING`.
Eksempel: `TRIM(' a b  '))` → `TRIM(' a b')`.

UBOUND (ARRAY[,DIM])

Øvre indeksgrænse af `ARRAY` (langs dimension `DIM`). `ARRAY`: Hvilken som helst type, ikke skalar, ikke en uassocieret pegepind eller et uallokeret array. `DIM`: `integer`. **Resultat**: `integer`, og er den øvre indeksgrænse af `ARRAY` som erklæret. Hvis `DIM` ikke er til stede, er resultatet et array med så mange dimensioner som `ARRAY` har. Er `DIM` med, skalar.
Eksempler: For erklæringen `A(0:10,-5:5)`, `UBOUND(A)` → `10,5`, og `UBOUND(A,2)` → `5`.

UNPACK (VECTOR,MASK,FIELD)

Folder vektoren ud, under kontrol af `MASK`, i det flerdimensionale `FIELD`. `VECTOR`: hvilken som helst type, endimensionalt array; `MASK`: `logical` array (eller skalar); `FIELD`: samme type som `VECTOR` og konform med `MASK`. **Resultat**: Samme type som `VECTOR` og er et array med den samme form (antallet af dimensioner) som `MASK`. Det element (i array-orden) i resultatet som svarer til det i'te `.true.` element i `MASK` har værdien `VECTOR(i)`, for $i = 1,2,\ldots,t$, hvor t er antallet af `.true.` elementer i `MASK`. Alle andre elementer er taget fra `FIELD`, hvis det er et array.
Eksempler: Hvis `M` er arrayet $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, `V` er `(/1,2,3/)` og `Q` er arrayet $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$ (hvor T betyder `.true.` og `.` betyder `.false.`), giver `UNPACK(V,Q,M)` resultatet $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$; hvis alle elementer i `Q` var `.true.`, ville resultatet af samme kald være $\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$ (udfoldning af `V` i retningen arrayorden). Består `FIELD` kun af nuller, giver `UNPACK(V,Q,0)` resultatet $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

VERIFY (STRING,SET,BACK)

Indikerer om alle tegn i strengen `STRING` er indeholdt i `SET`, ved at identificere den position af det første tegn i `STRING` som ikke er til stede i `SET`. Positionen kan scannes fremad (hvis `BACK` enten mangler eller er `.false.`), eller baglæns (hvis `BACK` er til stede og `.true.`). Elemental. `STRING`, `SET`: tegnstrenge. **Resultat**: `integer`.
Eksempler: `VERIFY('abba','a')` → 2 (fordi i strengen `'abba'`, er det første `'b'`, i position 2, ikke indeholdt i `SET`); `VERIFY('abba','ab')` → 0 (fordi der ikke mangler noget).

A.2 Fortran 95 intrinsics

Disse intrinsics findes kun i Fortran 95, eller er ændret fra deres form i Fortran 90 (se også kap. 12 og 13).

CEILING (A[,KIND])

Det nærmeste heltal $\geq$ `A`. Elemental. `A`: `real`; `KIND`: `integer`. **Resultat**: `integer`, af integer `KIND` hvis til stede, ellers af default integer `KIND`.
Eksempler: `CEILING(1.3)` → `2`;
`CEILING(-1.3)` → `-1`

CPU_TIME (TIME)

Processortidsforbrug (cpu-tid). Subroutine. `TIME`: `real` og skalar. Det får værdien af processorens tidsforbrug fra et ikke nærmere defineret tidspunkt, i sekunder. Hvis processoren ikke kan finde ud af dette, er resultatet negativt.
Brugs**Eksempel**:

```
real :: t1, t2, dt
...
call CPU_TIME(t1) ! cpuforbrug op til her
...
call CPU_TIME(t2) ! cpuforbrug op til her
dt = t2 - t1      ! afstand imellem de to.
```

FLOOR (A[,KIND])

Største heltal $\leq$ `A`. Elemental. `A`: `real`. `KIND`: `integer`. **Resultat**: `integer`, af integer `KIND` hvis til stede, ellers af default integer `KIND`.
Eksempel: `FLOOR(1.999)` → `1`; `FLOOR(-3.7)` → `-4`

MAXLOC (ARRAY[,DIM][,MASK])

Indekser for det største element i `ARRAY`. `ARRAY`: `integer` eller `real`, ikke skalar. `DIM`: `integer`. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: `integer`, endimensional array. Resultatet er det første "koordinat" i `ARRAY` med den største værdi. Hvis `MASK` er til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.` Er `DIM` til stede, indikerer den den dimension, langs hvilken søgningen skal gå. Hvis kun et af de to valgfrie argumenter er til stede, er det typen der bestemmer, hvilken der er ment, selv uden at bruge keywords.
Eksempler: `MAXLOC(B)` → `2,3`; `MAXLOC(B,B<6)` → `1,3`. `MAXLOC((/5,-9,3/),DIM=1)` → `1`. Søgningen tæller indekser fra 1, uafhængigt af indeksgrænserne.

MINLOC (ARRAY[,DIM][,MASK])

Indekser for det mindste element i `ARRAY`. `ARRAY`: `integer` eller `real`, ikke skalar. `DIM`: `integer`. `MASK`: `logical` og samme form som `ARRAY`. **Resultat**: `integer`, endimensional array. Resultatet er det første "koordinat" i `ARRAY` med den mindste værdi, Hvis `MASK` er til stede, holder søgningen sig til de steder, hvor `MASK` er `.true.` Er `DIM` til stede, indikerer den den dimension, langs hvilken søgningen skal gå. Hvis kun et af de to valgfrie argumenter er til stede, er det typen der bestemmer, hvilken der er ment, selv uden at bruge keywords.
Eksempler: `MINLOC(B)` → `1,1`; `MINLOC(B,B>3)` → `2,2`. `MINLOC(B,DIM=2)` → `1,1`. Søgningen tæller indekser fra 1, uafhængigt af indeksgrænserne.

NULL ([MOLD])

Giver en uassocieret pegepind. `MOLD` skal være en pegepind. **Resultatet** er af samme (target) type som `MOLD` hvis det er til stede, ellers er det defineret af sammenhængen. Denne function kan bruges til at sætte en pegepind i uassocieret tilstand, inklusive som initialisering sammen med en erklæring. Se i øvrigt kap. 11 og 13.

SIGN (A,B)

$|A|$ forsynet med fortegnet af `B`. Elemental. `A`, `B`: `integer`, `real`, begge samme model. **Resultat**: samme model som `A` og `B`. Hvis processoren kan skelne mellem to slags nul (positiv og negativ nul), og `B` er nul, retter fortegnet sig efter dette fortegn.
Eksempel: `SIGN(-3,2)` → `3`.

Bilag B
ASCII-tabel

B.1 Kontrolkoderne

ASCII-tabellen er den mest brugte tegnkode i dag. De første 32 tegn, med værdierne 0...31 er de såkaldte kontrolkoder (control codes), og nogle af dem kan indtastes ved at bruge CTRL-knappen. Disse er vist.

Kodeværdi	navn	beskrivelse
0	nul	
1	soh	
2	stx	
3	etx	
4	eot	end of input, `^d`
5	enq	
6	ack	
7	bel	biptone
8	bs	
9	ht	horisontal TAB, `^i`
10	lf	linjeskift, `RET`
11	vt	vertikal TAB
12	ff	sideskift
13	cr	vognretur, `RET`
14	so	
15	si	
16	dle	
17	dcl	
18	dc2	
19	dc3	
20	dc4	
21	nak	
22	syn	
23	etb	
24	can	
25	em	
26	sub	
27	esc	ESC knap
28	fs	
29	gs	
30	rs	
31	us	

B.2 Tastaturtegn

32		64	@	96	‘
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	’	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	del

Stikord

.FALSE., 9
.TRUE., 9

ABS, 107
ACHAR, 107
ACOS, 107
ADJUSTL, 107
ADJUSTR, 107
afrunding, 102, **104**
AIMAG, 107
AINT, 107
aktuel længde, 21, 64
alfanumerisk, 6
algoritme, 106
ALL, 56, 107
ALLOCATABLE, 55
ALLOCATE, **55**, 65
ALLOCATED, **57**, 107
alternate RETURN, 94
ANINT, 108
antal decimaler, 8
ANY, 56, **108**
argument, 25, 26
    dummy, 59
    formel, 27
    overført, 27
arithmetic IF, 95
aritmetisk udtryk, 11
array pointers, 83
array-valued functions, 66
arrays, 20, 53
    -I/O, 23
    adresse, 28
    af pegepinde, 84
    aktuel længde, 21, 28, 64
    allocatable, 55
    ALLOCATE, 55
    DEALLOCATE, 55
    dimension, 20
    dynamiske, 55
    elementorden, **21**, 23
    erklæring, 20
    functions, 33
    fysisk længde, **21**, 28–30, 64
    hel-, operationer, 23
    indeksgrænser, 20
    initialisering, 21
    intrinsics, 56
    konformitet, 20, 22
    konstant, 21
    længde, **20**, 64
    many-one, 53
    masked assignment, 54
    orden, 28
    overførsel, 27, 29, 63–65
    pegepinde, 82
    RESHAPE, 53, 54, **114**
    section, 22
    sætninger, 22
    tegnstreng-, 20
    tilordningssætning, 22
    udsnit, **22**, 24, 28, 29, 53
    WHERE, 54
ASCII tabel, 119
ASIN, 108
ASSIGN sætning, 97
assigned GOTO, 95, 97
assignment statement, *Se* tilordningssætning
ASSOCIATED, 108
assumed shape, 65
assumed size, 64, 65, 95
ATAN, 108
ATAN2, 108

BACKSPACE, 44
betingelse, 13
bibliotek, 32
binær operator, 72
binære tal, 102
binære træer, 90–92
bit, 102
bitmanipulationsfunctions, 33
bitmønstre, 102
BIT_SIZE, 108
blanding af typer, 11, 103
BLOCK DATA, 95
BTEST, 108
byte, 102

CALL, 27
    keyword, 62, 63
CASE, 13, 15
    default, 15
CEILING, 98, **108**, **117**
CHAR, 108
character, 9
character-udtryk, 13
character-valued functions, 67
CLOSE, 44
CMPLX, 108
COMMON, 95
compiler, 6
COMPLEX, 8
computed GOTO, 95, 97
CONJG, 108
CONTAINS, 31, 66
continuation lines, 7
control-d, 19
COS, 108
COSH, 108
COUNT, 56, **109**
CPU_TIME, 98, **117**
CSHIFT, 58, **109**
CYCLE, 16, 18

DATA, 93, 97
DATE_AND_TIME, 109
DBLE, 109
DEALLOCATE, 55
defined assignment, 77
demoprogram, 5
derived types, 74
DIGITS, 109
DIM, 56, 107, **109**
direkte kald, 26
direktiv, 5, 10
division, 11
DO, 15
DO WHILE, 16, 17, 94
DO-sætning, 15
dobbelt kolon, 8
DOT_PRODUCT, 56, 57, **109**

DOUBLE PRECISION, 8, 95, 97
DPROD, 109
dummyargument, 59
dummyprocedure, 31, **59**, 67
dummyvariabel, 26
dynamiske arrays, 55

edit items, 35, 36
  feltbredde, 101
eksekverbar sætning, 10
ekstern procedure, 26
elemental procedurer, 33, 59, 99
elementorden, 21
END, **5**, 6, 25, 26
end of record, 41
end-of-file, 18
ENDDO, 16
ENDFILE, 45
enhed, 25–27, 31
EOF, 18
EOSHIFT, 58, **109**
EPSILON, 110
EQUIVALENCE, 95
erklæring, 5, 6, 10
  function, 62
evalueringsrækkefølge, 12
executable sentence, 10
EXIT, 16, 17
EXP, 110
EXPONENT, 110
EXTERNAL, 67, 68

fastform-Fortran, 95, 97
fejl, 10
filer, 6, 41
  intern, 48
FLOOR, 98, **110**, **117**
FORALL, 99
forgrening, 14
format, 35
  A edit item, 37
  andre edit items, 38
  B, O og Z edit items, 36
  E og D edit items, 37
  EN edit item, 37
  ES edit item, 37
  F edit item, 36
  feltbredde, 36, 37
  fri, 36, 39
  G edit item, 37
  H edit item, 97
  I edit item, 36
  item-sammensætning, 38
  L edit item, 38
  printer control, 40
  regler, 39, 40
  repeat og kolon edit items, 39
  S, SP, SS edit items, 38
  T edit item, 38
  ved input, 36, 39
format items, 35
formatbeskrivelse, 18, 36
Fortran 90 intrinsics, 107–117
Fortran 95, 97–101
Fortran 95 intrinsics, 117–118
fossiler i Fortran, 93–96
FRACTION, 110
frit format, 39
function, 25, 26
  array valued, 66
  array-, 33
  bitmanipulations-, 33
  character valued, 67
  inquiry-, 33
  kald
    direkte, 26
    indirekte, 26
  KIND-, 33
  matematiske, 33
  numeriske, 33
  regler, 26
  tegn-, 33
  vektor/matrix, 34
fysisk længde, 21, 64

generic, 33, 59
globale størrelser, 27
GOTO, 93, 95

heltal, 7, 103
Hollerith edit item, 97
hoved, 25
HUGE, 110

I/O
  filer, 41
  med/uden ny linje (ADVANCE), 47
  NAMELIST, 46
  sætning, 46
    PRINT, 46
    READ, 46
    WRITE, 46
  uformateret, 48
IACHAR, 110
IAND, 110
IBCLR, 110
IBITS, 110
IBSET, 110
ICHAR, 110
IEOR, 110
IF, 13
  -niveau, 14
  -sætning, 13
ikke-numeriske typer, 9
IMPLICIT NONE, 5, 8, 10
IMPLICIT sætning, 94
implicit typing, 8
IMPLIED-DO, 21, 24
indbygget procedure, 33
indeksgrænser, 20
indeksvektorer, 53
INDEX, 110
indirekte kald, 26
indrykning, 6, 14
initialisering, 10
  arrays, 21
  pegepinde (NULL), 98
  strukturer, 100
  variabler, 10
input, 5
INQUIRE, 45
inquiry functions, 33
INT, 111
INTEGER, 7
INTENT, 60
  IN, 61
  INOUT, 53, 61
  OUT, 53, 61
interface, 61, 69, 76
  blok, 61, 62
  explicit, 61, 64
  implicit, 61

navngivet END-, 101
strukturer, 76
intern procedure, 26
interne filer, 48
intrinsic subroutines, 34
intrinsics, 33–34
IOR, 111
IOSTAT, 18, 19, 42, 45–47
ISHFT, 111
ISHFTC, 111
iterativ procedure, 69

kald af intrinsics, 34
kaldende program, 26
keyword, 5, 62
keyword kald, 34, 62, 63
KIND, **49**, 50–52, 62, 111
brug af, 50
brug af PARAMETER, 51
erklæring med, 50
functions, 33
intrinsic functions, 50
konstant, 32
med modules, 51
typer, 49
ved intrinsics, 52
kommentar, 5, **7**, 26
kompatibel, 20, 22
komplekstal, 8
konform, 20, 22
konstant, 7, 8, 10
kontrolkoder, 119
kædede lister, 86–90

label, 14, 15, 35
lagring, 103
heltal, 103
LOGICAL, 104
REAL, 103
tegn, 104
LBOUND, 57, 66, **111**
LEN, 111
LEN_TRIM, 111
LGE, 111
LGT, 111
linje
fortsættelse, **7**, 39
længde, 6
tom, 6
LLE, 111
LLT, 112
LOG, 112
LOG10, 112
LOGICAL, 9
konstant, 12
lagring, 104
operator, 12
udtryk, 12, 13
variabel, 12
værdi, 12
LOGICAL function, 112
logical unit, 41
logisk type, 9
lokal variabel, 27
løkker, 15
med labels, 94
REAL tæller, 94, 97

many-one arrays, 53
MASK, 107
maskinsprog, 6
matematiske functions, 33
MATMUL, 23, 56, 58, **112**
MATOUT, 30, 40
matrixmultiplikation, 23
MAX, 112
MAXEXPONENT, 112
MAXLOC, 58, 98, **112**, **117**
MAXVAL, 56, 58, 112
MERGE, 57, **112**
MIN, 113
MINEXPONENT, 113
MINLOC, 58, 98, **113**, **118**
MINVAL, 56, 58, **113**
MOD, 113
module, 25, 27, 28, **31**, 32, 59, 70
module procedure, 72
MODULO, 113
MVBITS, 113

NAMELIST, 46
kommentarer i, 101
NEAREST, 113
NINT, 113
NOT, 113
NULL, 98, 118
NULLIFY, 98
numeriske functions, 33
numeriske typer, 8
nøgleord, 5, 6

object-modul, 6
OPEN, 41
ACTION, 43
defaults, 44
FILE, 42
IOSTAT, 42
standard unit, 42
STATUS, 42
UNIT, 42
operator, 11
-tegn, 13
aritmetisk, 11
binær, 72
defineret, 71, 72
logisk, 12
prioritet, 11, 12
udvidelse, 71–73
unitær, 72
opfordring, 5
OPTIONAL, 62, 63
orden, arrayelementer, 21
overflødige kommentarer, 7
overførsel
arrays, 27, 63–65
assumed shape, 65
assumed size, 64, 65
positional/keyword, 34
procedurer, 31, 59
strukturer, 75
tegnstrenge, 30, 66
overloading, **71**, 72, 77
overskuelighed, 14
oversætter, 6

PACK, 58, **113**
PARAMETER, **9**, 10, 20, 31
PAUSE, 96, 97
pegepinde, 81–92
arrays, 82
arrays af, 84
association, 81

binære træer, 90
disassociation, 81
dynamiske datastrukturer, 86
erklæring, 81
i strukturer, 83
implicit target, 86
initialisering, 98
intrinsics, regler, 82
kædede lister, 86
NULL, 98
strukturer, 82
target, 81
pointers, *Se* pegepinde
positional argument, 34
PRECISION, 114
PRESENT, 63, **114**
PRINT, 46
-sætning, 19
linje, 5
sætning, 5
printer control, 40
prioritet, 11
PRIVATE, 66, 78
procedure, 25, 26
dummy-, 67
ekstern, 26, 31, 59
elemental, 33, 99
Fortran 95, 34
generic, 33, 71
intern, 26, 31, 59
intrinsic, 33
iterativ, 69
module, 72
module-, 59
overførsel, 31, 59
PURE, 99
rekursiv, 69
strukturer, 75
PRODUCT, 56, **114**
program, 6
program unit, 25
programdesign, 102
programenhed, 25, 26
programhoved, 5, 10
programnavn, 5
prompt, 5
præcision, 8, **103**
PUBLIC, 66, 78
PURE procedurer, 99

RADIX, 114
RANDOM_NUMBER, 114
RANDOM_SEED, 114
RANGE, 114
READ, 5, 18, 46
-sætning, 18
read-while, 18, 47
REAL, 7, 8
REAL (function), 114
REAL løkketæller, 94, 97
REAL tal, 103
record, 41
recordlængde, 41
rekursion, 26, 68, **68**, 69, 70
rekursiv procedure, 69
REPEAT, 114
REPEAT UNTIL, 16, 17
RESHAPE, 53, 54, 58, **114**
RESULT, 60, **60**, 68, 69
RETURN, 93
alternate, 94
REWIND, 44
RRSPACING, 115
rækkevide, 26

S, SP, SS edit items, 38
SAVE, 59–60
SCALE, 115
SCAN, 115
scope, 26
SELECTED_INT_KIND, 50, **115**
SELECTED_REAL_KIND, 50, **115**
semikolon, 14
SEQUENCE, 75, 95
SET_EXPONENT, 115
SHAPE, 57, **115**
SIGN, 98, **115**, **118**
simpel PRINT, 18
simpel READ, 18
SIN, 115
SINH, 115
SIZE, 57, 67, **115**
skalar, 20
små bogstaver, 6
sortering, 69, 84
SPACING, 116
specification expressions, 100
SPREAD, 58, **116**
SQRT, 71, **116**
standard-input, 19
standardområde, 8
startværdi, 10
statement, 6, 10
statement function, 25, 59, 97
STOP, 19, 93
store bogstaver, 6
strukturer, 74
dynamiske, I/O, 92
I/O, 74, 92
initialisering, 100
konstanter, 75
med pegepinde, 83
overførsel, 75
procedurer, 75
sortering, 84
subroutine, 25, 27
SUM, 56, 57, 116
symbol, 6
SYSTEM_CLOCK, 116
sætning, 6, 10
ASSIGN, 97
BACKSPACE-, 44
CASE, 13, 15
CLOSE-, 44
DATA, 97
DO, 15
eksekverbar, 10
ENDFILE-, 45
erklæring, 10
EXTERNAL, 67, 68
format-, 35
I/O, 35, 46
IF, 13
INQUIRE-, 45
løkker, 15
NAMELIST, 46
OPEN-, 41
REWIND-, 44
rækkefølge, 10
tilordnings-, 10, 11

tal

binær, 102
INTEGER, 103
REAL, 103
talområde, 8, 49
TAN, 116
TANH, 116
target, 81, 86
tastatur, 5, 9, 18, 35, 36, 119
tegn, 9
-I/O til/fra, 48
-konstant, 9
-streng, 10, 18
-strengarray, 20
-strenge, 30
-sæt, 6, 119
-udtryk, 13
-variabel, 7, 9
-værdi, 9
functions, 33
lagring, 104
strenge
overførsel, 66
tabel, 119
tekstfiler, 41
tilordningssætning, 10, 11, 13
TINY, 116
tom linje, 5
TRANSFER, 116
TRANSPOSE, 58, **116**
TRIM, 117
type constructors, 75
typer, 7, 10
blanding, 11
character, 9
COMPLEX, 8
derived, 74
double precision, 7
implicit, 8
logisk, 7, 9
numerisk, 7
tegn, 7, 9
tælleløkke, 16
tæller, 16, 17

UBOUND, 57, 66, **117**
udførbare linjer, 5
udtryk, 11
aritmetisk, 11
logisk, 12
tegn, 13
valg, design, 105
uformateret I/O, 48
underprogram, 25, 27
UNIT, 42, 46
unitær operator, 12, 72
UNPACK, 58, **117**
USE, 32, 62

variabel, 6, 7
variabelnavn, 5
vektor/matrix functions, 34, 56
VERIFY, 117
vognretur, 18
værdi, 7, 9

WHERE, 54, 100
word, 102
WRITE, 46