Lärobok i Fortran 95

© Bo Einarsson[1]

21 december 2018

[1]Matematiska institutionen, Linköpings Universitet, 581 83 LINKÖPING

Innehåll

Tabeller

Förord

Version 4.5

Programspråket Fortran är det helt dominerande språket för tekniskt vetenskapliga beräkningar. Det utvecklades ursprungligen 1954 vid IBM under ledning av John Backus (1924 – 2007) och har reviderats flera gånger.

Syftet med denna bok är att vara en lärobok i det mycket vanliga och populära programspråket Fortran 95, vilket innehåller både kraftfulla konstruktioner och möjlighet till en strikt kontroll av programmen. Fortran 95 erbjudes numera från så gott som samtliga datorleverantörer, några erbjuder till och med de flesta av tilläggen i Fortran 2003 och dessutom en del av tilläggen i Fortran 2008 och Fortran 2018. Fortran är det helt överlägsna och dominerande programspråket för tekniskt-vetenskapliga beräkningar.

Alla exempel i denna bok har provkörts såväl under UNIX på Sun SPARC och DEC Station ULTRIX som på PC under MS-DOS. Läsaren bör ha tillgång till ett Fortran 95 (eller helst högre) system.

De i boken införda programexemplen är avsedda att illustrera programmeringstekniken och aktuella kommandon, avsikten är däremot inte att ge optimerad tillämpningsprogramvara. Detta gäller speciellt avsnittet svar och kommentarer till övningarna. Notera dock att en del av de senare exemplen är mycket fullständiga vad avser de gränssnitt och deklarationer som erfordras vid körning av funktioner och subrutiner.

Inriktningen av boken är allmän, men med en kraftig koncentration på numeriska och tekniska beräkningar, dock med utförlig behandling av erforderlig texthantering. Den täcker dock ej uppgifter av typ telefonkatalog eller hur man skriver preprocessorer eller kompilatorer i Fortran. Övningar till de flesta avsnitt finns, med i många fall utförliga lösningar. Antalet övningsuppgifter är relativt litet, det är därför desto viktigare att de görs av läsaren.

Jag vill tacka John Reid för hjälp när jag haft problem med tolkningen av en del nya begrepp i Fortran 95. Jag vill likaså tacka mina kolleger vid matematiska institutionen vid Linköpings universitet för värdefulla synpunkter, samt eleverna vid kurserna i Fortran under åren 1992 till 2007.

I boken behandlas Fortran enligt standarderna Fortran 77, Fortran 90 och Fortran 95, vilka är uppbyggda så att allt i Fortran 77 finns i Fortran 90 och allt i Fortran 90 även i Fortran 95. När jag skriver att något införts i Fortran 90 så finns det således även i Fortran 95. Även vissa exempel från de nya standarderna Fortran 2003 finns i kapitel 17 respektive Fortran 2008 i kapitel 18 samt något om Fortran 2018 i kapitel 19. Jag skriver alla programexempel liksom Fortranord normalt med stora bokstäver och utnyttjar stilen Courier även för UNIX-kommandon.

Synpunkter, förslag och eventuella rättelser är mycket välkomna inför nästa upplaga. I denna upplaga har jag reviderat boken så att allt nytt i Fortran 95 nu är inkluderat på sin naturliga plats och inte längre bara som tillägg eller i bilaga, samt naturligtvis rättat ett antal mindre fel. Jag har tagit bort en del mindre aktuellt material och i stället hänvisat till internetversionen.

Boken finns nämligen även i en form för webb-läsare, på NSC:s webbplats `http://www.nsc.liu.se/~boein/f90/`. Den versionen har en annan numrering och innehåller en del extra material, delar av detta är nu av mindre intresse. Dessutom finns där källkoden till de flesta exemplen som körklara filer.

I Sakregistret kommer alla Fortran-ord först, liksom filnamn och UNIX-kommandon, alla dessa är skrivna i Courier. Alla förslag till förbättringar mottages tacksamt. Förutom på papper kan boken läsas även online, versionen i PDF har fungerande klickbara länkar, ett fåtal av dessa länkar pekar tyvärr inte helt rätt. Detta förhållande gäller främst avsnitten i början och slutet av boken, man måste då gå fram ytterligare några sidor för att hamna rätt.

Linköping den 21 december 2018.

Bo Einarsson
`bo.g.einarsson@gmail.com`
`bo.einarsson1@comhem.se`

Kapitel 1

Ett första program i Fortran 95

1.1 Inledning

Det första programmet i Fortran [21] räknar ut en avrundningsenhet μ eller **u** som ofta användes i samband med numerisk analys av datorberäkningar. Denna konstant kan enklast uppfattas som det halva avståndet mellan 1 och nästa flyttal. Det ger en begränsning av det relativa fel med vilka godtyckliga tal kan lagras som flyttal, om korrekt avrundning sker. Professor William Kahan har angett en listig metod att räkna ut detta värde. Denna metod fungerar på alla datorer som ej har basen tre (eller en multipel av tre).

1.2 Programlistningen för programmet KAHAN

```
PROGRAM KAHAN
IMPLICIT NONE
REAL :: A, B, C, D, E

A = 1.0/3.0
B = 4.0*A - 1.0
C = 3.0*B - 1.0
D = 0.5*C
E = ABS(D)

WRITE(*,*) ' my = ', E

END PROGRAM KAHAN
```

Programmet är som synes skrivet med stora bokstäver (VERSALER), vilket är det vanligaste vid Fortran, men det går numera lika bra att använda små bokstäver (gemena). Processorn gör ingen skillnad mellan små och stora bokstäver, utom vid in- och utmatning.

Första raden ger namnet på programmet, den andra är en mycket bra sats som kommer att diskuteras senare. I den tredje raden talar vi om för systemet

att de fem namnen `A, B, C, D` och `E` svarar mot flyttal.[1]

Nu följer fyra rader med själva beräkningen. Om du skriver ner det matematiska uttrycket0 för `D` får Du $0,5(3(4(1/3)-1)-1) = 0,5(3(4/3-1)-1) = 0,5(3(1/3)-1) = 0,5(1-1) = 0$, men det förefaller ju inte meningsfullt att beräkna noll! Det är naturligtvis så att beräkningsfelet är det intressanta här. Den första beräkningen är av en tredjedel, som ej kan representeras exakt på en dator med basen[2] 2, 8, 10 eller 16. Ovanstående innebär bland annat att Du inte får förenkla de fyra satserna för mycket.

Notera att en tredjedel måste skrivas som 1.0/3.0, för att tala om för systemet att det är fråga om flyttal och inte om heltal. De fyra räknesätten anges med + - * respektive /. Notera att i matematik användes ofta en punkt, ett kryss, eller ingenting för att markera multiplikation, som $2 \cdot 3, a \times b$, eller $5c$, men i Fortran måste asterisken * användas för att markera multiplikation. Tilldelning sker med ett likhetstecken =, varvid storheten till vänster får värdet av det till höger, något kolon lika med som i Algol eller Pascal finns inte. Radslut behöver normalt inte markeras.

Eftersom storheten `D` kan bli negativ tar vi bort tecknet med hjälp av den inbyggda funktionen `ABS`.

Utmatningen sker med `WRITE`-satsen, här utnyttjande standardenheten för utmatning och standardformatet att skriva ut (markerat med de båda asteriskerna). Texten inom apostroferna skrivs ut, följt av värdet på storheten `E`. Resultatet blir på ett vanligt system (Sun, DEC station, PC eller VAX/VMS)

```
my =   5.9604645E-08
```

och på ett ovanligt system (nämligen Cray C90)

```
my =   7.105427357601E-15
```

Den sista raden med `END` talar om för systemet att programmet slutar där.

1.3 Körning av programmet `KAHAN`

Den första åtgärden är att ordna en fil som innehåller texten till programmet. Ett lämpligt namn på en sådan fil är `kahan.f90`. Filen skapas lämpligen med UNIX standard-editor `vi` eller den i den akademiska världen vanliga editorn `emacs`.

[1] **Flyttal.** Vi tycker som vetenskapsmän inte om att skriva ut små tal som 0,000 000 007 89 eller stora tal som 6 540 000 000, utan skriver dem ofta på exponentform som $7{,}89 \cdot 10^{-9}$ respektive $6{,}54 \cdot 10^{9}$. För att spara lagringsutrymme gör datorn på motsvarande sätt, men datorn har då normalt bara ett fixt antal positioner för taldelen (mantissan) och exponenten. Det kan således bli ett avrundningsfel om mantissan inte får plats och spill (eng. overflow) om exponenten är för stor och bottning (eng. underflow) om exponenten är för liten. Ett flyttal har således en mantissa och en exponent. På ett vanligt system (till exempel de flesta UNIX system, PC eller Apple Macintosh) kan positiva tal mellan $1.18 \cdot 10^{-38}$ och $3.40 \cdot 10^{38}$ representeras. Dessa tal skrives i Fortran som `1.18E-38` och `3.40E38`. Notera att i Fortran måste man använda punkt som decimaltecken och basen 10 markeras med ett `E`, ibland med ett `D`.

[2] **Bas.** Bas är det tal som man utnyttjar när man bildar exponenter, i vanliga fall räknar vi med basen 10 (det decimala talsystemet), medan datorer oftast utnyttjar någon av baserna 2, 8 eller 16, vilka samtliga kan betraktas som varianter av den binära. Det är faktiskt svårt att definiera basen, en metod är att utnyttja att om ett godtyckligt tal multipliceras eller divideras med basen erhålles inget avrundningsfel.

Programmet skall nu kompileras, länkas och köras. Detta sker i två steg, först kompilering och länkning med kommandot

```
f90 kahan.f90
```

och sedan körning med

```
a.out
```

Notera att UNIX skiljer på kommandon och filnamn givna med små eller stora bokstäver. Vid användning av Cray (och andra maskiner där `PATH` inte är satt på normalt sätt) bör aktuell filkatalog anges, varför körning sker med

```
./a.out
```

Namnet på kompilatorn varierar mellan olika system, vanliga är `f77, f90, f95` med uppenbar betydelse. Intel kallar sin `ifort` medan de från GNU är `g77` och `g95` samt `gfortran`.

Inte alla kompilatorer inkluderar alla nyheterna i Fortran 2003, Fortran 2008 och Fortran 2018, se vidare [6].

Kapitel 2

Grunder i Fortran 95

Inledning

I detta kapitel skall vi gå igenom de regler som gäller för alla program i Fortran 95, men koncentrera oss till de mest grundläggande reglerna. Efter studium av detta kapitel skall Du kunna skriva och köra enkla program i Fortran 95. I de följande kapitlen kommer sedan fler egenskaper i Fortran att introduceras.

Denna bok kräver inga förkunskaper i Fortran, men väl i programmeringsteknik. Om läsaren är nybörjare i programmering rekommenderar jag boken av Brainerd m fl [3]. Den formella standarden [21] är dyr, svår att få tag på samt ganska svårläst. En enklare och billigare variant är boken "Fortran 2003 Handbook" [1] som fungerar utmärkt som uppslagsbok. En mer kompakt men utmärkt bok om Fortran 95/2003/2008 är den av Metcalf m fl [27].

På svenska fanns en bra bok av Katarina Blom [2], som dock nu är slutsåld, samt den som bara behandlade Fortran 77 av Ekman [13].

2.1 Tillgängliga tecken vid skrivning av program i Fortran 95

Under utvecklingen av Fortran har antalet olika tecken som kan användas vid skrivning av program växt. Det är dock ofta så att de tecken som inte fanns med från början inte vunnit så allmän acceptans, varför jag här ger en successiv beskrivning av de som ingår. I Fortran 66 (FORTRAN IV) ingår de 26 engelska bokstäverna

```
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
```

de 10 siffrorna

```
0 1 2 3 4 5 6 7 8 9
```

samt de 10 specialtecknen

```
= + - * / ( ) , . $
```

Specialtecknen, utom den sista, har en naturlig användning i Fortran. Dessutom ingår naturligtvis blank i teckenuppsättningen.

Det låga antalet tecken (totalt 47) beror på att dåtidens utrustning (hålkortsstans) hade en mycket begränsad teckenrepertoar. Notera att de små bokstäverna inte fanns med. De kom "nästan" i Fortran 90, där standarden föreskriver att om implementeringen tillåter små bokstäver så skall dessa vara ekvivalenta med motsvarande stora bokstäver, utom i text-sammanhang. De flesta Fortran 77 implementationer har samma tolkning av små bokstäver.

I Fortran 77 tillkom följande båda specialtecken

```
' :
```

I Fortran 90 tillkom ytterligare nio specialtecken

```
_ ! " % & ; < > ?
```

De båda symbolerna `$` och `?` har ingen specificerad användning i Fortran 95, utan är tänkta främst för utmatning. Symbolen `$` har dock haft en viss specificerad betydelse i vissa utvidgningar av Fortran 77, den har dessutom en viktig betydelse i UNIX. Mer om detta senare. Naturligtvis kommer jag även att förklara de övriga nytillkomna specialtecknen efter hand.

I Fortran 2003 tillkommer ytterligare 11 specialtecken,

```
~ | \ [ ] { } @ # ‘ ^
```

Endast **[** and **]** har en semantisk innebörd.

De nationella bokstäverna (av typ å ä ö é è ë ü ñ) kan oftast användas i textsammanhang, om stöd för dem finns i implementeringen. De kan nästan aldrig användas i variabelnamn, och bör aldrig användas i variabelnamn om något system skulle råka tillåta det!

2.2 Variabler

Ett fundamentalt begrepp inom programmering är variabler, vilka kan tilldelas olika värden. För att kunna använda variabler måste sådana ges namn. Variabelnamn i Fortran 77 (och tidigare) inleds av en bokstav och fortsätter eventuellt med ytterligare bokstäver och siffror. Högst sex tecken var tillåtna. Exempel på variabelnamn är

```
A ADAM        H2SO4   ILL     HEJSAN     H2J3A4     O
J JAMEN       SLASK   TMP     ULI        LIU        MAI
```

I Fortran 90 har största tillåtna längden på ett variabelnamn ökats från 6 till 31, och understrykningstecknet `_` får ingå inuti ett variabelnamn. Tillåtna variabelnamn, förutom de ovanstående, är

```
T_1   AVSTAAND_TILL_MAANEN  DISTANCE_TO_THE_SUN_
HEJSANSVEJSAN     0123456789ABCDEFHIJKLMNOPQRSTUV
```

I Fortran 2003 har största tillåtna längden på ett variabelnamn ökats ända till 63.

De nya reglerna innebär dock inte att man bör välja krångliga namn. Avsikten med understrykningstecknet är att användas när det är lämpligt med namn bestående av mer än ett ord. Blanka är ju inte tillåtna inuti namn. Man bör vara försiktig med tecken som kan misstolkas, ofta blir det fel mellan bokstaven O och siffran 0, dessa är väldigt olika i vissa typsnitt, men ganska lika i typsnittet Courier, nämligen `O` respektive `0`. Tyvärr finns det ingen bestämd regel mellan olika typsnitt om att just bokstaven skall vara "bredare" än siffran.

Övning

(2.1) Vilka av följande variabelnamn är tillåtna under Fortran 77 respektive Fortran 90?

```
A2      GUSTAVUS        ADOLFUS          GUSTAV_ADOLF
2C      2_CESAR         ÅKE              $KE
C-B     DOLLAR$         OOOOOO           DO
K**2    HEJ_DU_GLADE    _STOCKHOLM_      GOETEBORG
EIIR    Bettan          ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

2.3 Deklaration av variabler

Eftersom allt i datorer lagras binärt, som ettor och nollor, måste varje variabel ha information om hur bitmönstret skall tolkas. Detta sker genom att tilldela varje variabel en datatyp, som heltal, flyttal, logisk, eller textsträng. Några ytterligare datatyper kommer att introduceras senare, men dessa fyra är de viktigaste. En variabel deklareras att ha en viss typ på följande sätt.

```
REAL                  :: A, B, C
INTEGER               :: I, J, K, L, M, N
LOGICAL               :: BO
CHARACTER (LEN=10)    :: TEXT1
CHARACTER (10)        :: TEXT2
CHARACTER*10          :: TEXT3
```

Dessa deklarationer talar om att variablerna `A, B` och `C` är flyttal, variablerna `I, J, K, L, M` och `N` är heltal, variabeln `BO` är logisk (boolsk), och variablerna `TEXT1, TEXT2` och `TEXT3` är textsträngar som rymmer 10 tecken. De båda första deklarationerna är strängt taget onödiga, eftersom Fortran har den regeln att odeklarerade variabler som börjar på någon av bokstäverna `I, J, K, L, M` eller `N` automatiskt blir heltal, och de som börjar på någon annan bokstav blir flyttal. Denna regel har gett upphov till mycket elände, eftersom kompilatorn då inte upptäcker felstavade variabler, liksom att felaktig användning av Fortran-kommandon i stället kan ge upphov till nya variabler. Det har därför införts ett helt nytt kommando i Fortran 90, nämligen `IMPLICIT NONE`. Detta kan placeras först i varje programenhet[1], och slår av regeln om initialbokstäverna. Jag kommer att försöka att i fortsättningen genomgående använda

[1] Programenhet. En programenhet är ett huvudprogram (hittills har vi bara tittat på sådana), en subrutin, en funktion, en modul eller en `BLOCK DATA` programenhet. Satsen `IMPLICIT NONE` bör således upprepas först i var och en av dessa.

`IMPLICIT NONE` i denna bok, det är ett mycket bra verktyg för att få korrekta program.

I textsträngsdeklarationen ingår en specifikation av hur många tecken som skall kunna rymmas i variabeln. Man kan här utelämna `LEN=` och direkt ange antalet tecken inom parentesen. Alternativt kan man i stället för parentesuttrycket använda den gamla (Fortran 77) beteckningen * följt av antalet tillåtna tecken. Alla tre varianterna har använts ovan. Om längdspecifikationen utelämnas helt blir längden 1.

I Fortran 77 användes inte beteckningen med dubbelkolon ::, att den behövs i Fortran 90 beror på att man infört attribut som tilläggsspecifikationer vid deklarationer.

2.4 Tilldelningar och beräkningar

När vi nu har lyckats deklarera variabler är det dags att använda dem. Normalt arbetar man med en programrad i taget. De fyra räknesätten anges med + - * respektive /. Vi upprepar att i matematik användes ofta en punkt eller ingenting för att markera multiplikation, som $2 \cdot 3$ eller $5x$, men i Fortran måste asterisken * användas. Upphöjt till markeras med dubbelstjärna **. Tilldelning sker med ett likhetstecken =, varvid storheten till vänster får värdet av det till höger.

```
IMPLICIT NONE
INTEGER       :: I
REAL          :: AREA, R
LOGICAL       :: KLAR, OKLAR
R = 2.0
AREA = 3.141592654*R**2
I = 0
I = I + 1
KLAR = .FALSE.
OKLAR = .TRUE.
END
```

I detta enkla program sättes cirkelns radie `R` till 2 enheter och dess yta `AREA` beräknas med den välkända formeln $a = \pi \times r^2$. Därefter nollställes heltalet I, varefter det stegas upp med 1. De båda sista tilldelningssatserna sätter de båda logiska variablerna `KLAR` och `OKLAR` till värdena falskt respektive sant, vilka värden skrives på detta underliga sätt, där punkterna på båda sidor ingår i konstanterna. Det bör noteras att variabler i Fortran ej är automatiskt nollställda från början.

Punkter användes på flera ställen i Fortran för att definiera sådant som ej kan skrivas med den begränsade teckenrepertoar som finns i Fortran. Man kan då syntaktiskt särskilja dessa "punktbegrepp" från både Fortrans vanliga kommandon och från variabelnamn.

Vid kompilering av ovanstående program kan det inträffa att kompilatorn varnar för att variablerna `AREA, KLAR` och `OKLAR` ej användes efter att de tilldelats sina värden. Detta är en korrekt iakttagelse, och en kompilator som noterar sådant är ett gott hjälpmedel för att skriva korrekta program. I ovanstående exempel vore det naturligt att lägga in en utmatning av variablerna, men jag har ju inte hunnit till det än (se sektion 2.7).

2.5 Inbyggda matematiska funktioner

De vanliga matematiska funktionerna finns inbyggda i Fortran. Att de är inbyggda[2] betyder bland annat att de inte behöver deklareras, och ingen speciell åtgärd erfordras för att de skall länkas in i det färdiga programmet. De inkluderar funktioner som sin, cos, tan och motsvarande inversa eller hyperboliska (men ej inversa hyperboliska) samt kvadratrot, logaritm och exponentialfunktion. De finns alla specificerade i Bilaga D.3, sid 190.

I Bilaga D.2, sid 187, finns något som kallas numeriska funktioner, vilka innefattar sådant som absolutbelopp, realdel, heltalsdel och tecken. Där finns även funktioner för omvandling mellan olika precisioner, vilket vi återkommer till. De numeriska funktionerna är således av en mer maskinnära natur än de matematiska.

I Fortran 90 tillkom ett stort antal nya inbyggda funktioner, och även ett par inbyggda subrutiner, varför kommittén fann det lämpligt att gruppera dem som i Bilaga D. Bara funktioner i avsnitten 2, 3, 4 och 5 fanns med i Fortran 77 (och ej ens alla dom).

Vi ger nu ett enkelt exempel på användning av några av de inbyggda funktionerna.

```
IMPLICIT NONE
INTEGER       :: I, J, K
REAL      :: AREA, R, X, Y
AREA = 1.0
R = SQRT(AREA/3.141592654)
I = INT(-3.141592654)
J = FLOOR(-3.141592654)
K = CEILING(-3.141592654)
X = REAL(I*J*K)
Y = SIN(LOG10(ABS(X)))
Y = ACOS(Y)
END
```

I ovanstående lilla program beräknas först radien på en cirkel med ytan en enhet, varefter tre olika heltalsdelar av -pi beräknas, först den vanliga med `INT` som avrundar (trunkerar) mot noll och gav -3, sedan golvfunktionen `FLOOR` som avrundar nedåt mot - oändligheten och gav -4, och slutligen takfunktionen `CEILING` som avrundar uppåt mot + oändligheten och gav -3.

Därefter beräknas `X` som flyttalet svarande mot produkten av dessa tre heltal (-36) och man bildar `Y` som $\sin(^{10}\log|x|)$, varefter `Y` blir arcus cosinus av sig själv.

För arcustangenten finns två funktioner, dels den vanliga `ATAN(X)` som svarar mot $\arctan x$, dels den ovanliga `ATAN2(Y,X)`. Den senare har den uppgiften att den skall kunna klara av även de punkter där tangenten blir oändlig, nämligen

[2] **Inbyggd.** Ovanstående definitioner av matematisk och numerisk är begränsade till Fortran 90, och bör ej användas för övrigt. Den gamla standarden talade bara om inbyggda funktioner och utnyttjade därvid det engelska ordet intrinsic, vilket betyder inre och inneboende. Med inneboende avses naturligtvis inte exempelvis en inneboende student (eng. lodger, am. roomer) utan mera bildligt, som i uttrycken inneboende kvalitet eller inre värde. Ordet inbyggd (eng. built in) blir för svagt på engelska, det gäller att betona att de inbyggda funktionerna är integrerade i systemet. På svenska täcker ordet inbyggd väl även denna betydelse.

udda multipler av $\pi/2$. Resultatet av `ATAN2` kan tolkas som principalvärdet för argumentet till det från noll skilda komplexa talet x + iy. Det är funktionen $\arctan(y/x)$ och ligger i intervallet $-\pi <$ `ATAN2(Y,X)` $<= \pi$. Om y är positivt blir resultatet positivt, om y är noll blir resultatet noll om $x > 0$ och π om $x < 0$. Om y är negativt blir resultatet negativt. Om x är noll blir resultatet $\pi/2$ eller $-\pi/2$, beroende på tecknet på y. Om både x och y är noll blir resultatet odefinierat.

Samtliga trigonometriska funktioner arbetar med radianer.

2.6 Kommentarer

En kommentar[3] inleds med utropstecken ! och varar raden ut. Den kan starta redan i positionen längst till vänster (kolumn 1), och är då nästan i överensstämmelse med Fortran 77. I Fortran 77 inleddes dock kommentarer i stället med bokstaven `C` eller asterisk `*`. I Fortran 90 kan en kommentar även följa på samma rad som en sats. Kommentarer kan rymma nationella tecken, som de svenska tecknen åääöéü ÅÄÖÉÜ. Allmänt gäller att en kommentar finns i programlistningen, men den påverkar ej programmet vid körning. Från och med Fortran 77 kan även helt blanka rader användas som kommentarer (för att förbättra programmets utseende).

Ett kommando med en liknande uppgift som en kommentar är den programsats som kan inleda ett program (egentligen ett huvudprogram). Den har utseendet

```
PROGRAM namn
```

och ger namnet `namn` på programmet. Detta programnamn utnyttjas av vissa operativsystem, om någon programsats ej är med får oftast programmet automatiskt namnet `MAIN` eller `main`. Programsatsen är frivillig, men jag rekommenderar den.

2.7 Enkel in- och utmatning

In- och utmatning under Fortran är mycket kraftfull. Vi kommer nu till att börja med enbart behandla de enklaste aspekterna av dem och begränsa oss till inmatning från tangentbordet och utmatning till skärmen. Senare tillkommer filhantering och speciella egenskaper i samband med utmatning på papper.

Inmatning sker med `READ` och utmatning med `WRITE`. Dessa operationer måste knytas till vissa fysiska enheter, det är lämpligt att använda * för dessa. Man kan även använda de historiska numren 5 för in-enheten och 6 för ut-enheten. Ett exempel följer.

```
PROGRAM INUT_RI
IMPLICIT NONE
REAL    :: A
INTEGER :: I
WRITE(*,*) ' Ge värdena på flyttalet A och heltalet I'
READ(*,*)  A, I
```

[3]Fortran är det första programspråk som inkluderar kommentarer!

```
WRITE(*,*) A, I
WRITE(*,*) ' A = ', A, ' I = ', I
END
```

Den första stjärnan i parentesen ovan talar om att standardenheten skall användas, den andra att list-styrd in- respektive utmatning skall användas. List-styrd innebär att data tolkas i enlighet med vilka tal som skall matas in eller ut, matar man ut ett flyttal skrivs det också ut som ett flyttal. Den första skrivsatsen ovan skriver ut texten inom apostroferna, dvs en uppmaning att ge värdet av ett flyttal `A` och av ett heltal `I`. Man ger värdena med mellanslag, komma eller vagnretur mellan värdena, och avslutar med en vagnretur. Observera att om heltalet är stort, till exempel 7 283 550 så får det inte skrivas så, man får inte ha några blanka inuti talen. Man kan ge flyttal som heltal, heltal med decimaler, eller heltal med decimaler och exponent. Man kan således ge det som 13 eller -5 eller 157.67 eller `2.38E7` eller `4.E-8` eller `.2E5` (de båda sista varianterna är faktiskt tillåtna och tolkas som `4.0E-8` respektive `0.2E5`). Däremot är det inte tillåtet att utelämna siffrorna både före och efter decimalpunkten! Den följande skrivsatsen skriver ut värdena, den därpå följande talar även om vilka värden som skrivits ut.

Vi övergår nu till de båda datatyperna logisk och textsträng.

```
PROGRAM INUT_LC
IMPLICIT NONE
LOGICAL                 :: B
CHARACTER(LEN=8)        :: T
WRITE(*,*) ' Ge den logiska variabeln B '
READ(*,*) B
WRITE(*,*) ' Den logiska variabeln B är ', B
WRITE(*,*)
WRITE(*,*) ' Ge textsträngsvariabeln T (högst 8 tecken)'
WRITE(*,*) ' Observera att textsträngen måste ges inom'
WRITE(*,*) ' apostrofer eller citattecken'
READ(*,*) T
WRITE(*,*) ' Textsträngen T är ', T
WRITE(*,*)
WRITE(*,*) ' Ge textsträngsvariabeln T (högst 8 tecken)'
WRITE(*,*) ' Observera att textsträngen nu skall ges'
WRITE(*,*) ' utan extra tecken'
READ(*,'(A8)') T
WRITE(*,*) ' Textsträngen T är ', T
END
```

När man skall mata in att en logisk variabel skall vara sann kan man välja en av representationerna `T` eller `.T.` eller `.TRUE.` När man skall mata in att en logisk variabel skall vara falsk kan man välja en av representationerna `F` eller `.F.` eller `.FALSE.` List-styrd utmatning av en logisk variabel sker med ett ensamt `T` eller `F`. Vid list-styrd inmatning av en textsträng måste den omges med apostrofer ' eller citattecken ". Man kan valfritt välja om man vill ange en textsträng inom apostrofer eller citattecken, om texten skall innehålla ett av dessa tecken är det praktiskt att omge texten med det andra tecknet, i annat fall måste det tecken som skall skrivas ut dubbelskrivas.

Det är oftast opraktiskt att behöva omge textsträngar med apostrofer, man måste ju då svara `'JA'` i stället för `JA` på en JA/NEJ fråga. Man kommer ifrån det kravet genom att i stället för list-styrd inmatning använda format-styrd inmatning. Detta sker i den sista delen av ovanstående exempel, där den andra stjärnan utbytts mot '(A8)', vilket talar om för systemet att åtta tecken förväntas. Jag kommer i kapitel 6 utbreda mig mycket om format, allt för mycket enligt många, men det är ett mycket kraftfullt hjälpmedel, främst för att erhålla en prydlig utmatning.

2.8 Vektorer

Eftersom datorer är så snabba kan de behandla väldigt många element på en kort tid. Det vore då opraktiskt om varje variabel måste ha ett eget namn, man använder därför ofta indexerade variabler. Vi skall i detta avsnitt bara titta på variabler med ett index, kallade vektorer, för att i nästa kapitel titta på matriser och allmännare fält. Utrymme för vektorn $a_i, i = 1, 2, ..., 20$, reserveras med satsen

```
DIMENSION A(20)
```

I stället för ordet `DIMENSION` kan man använda någon av deklarationerna `CHARACTER, INTEGER, LOGICAL, REAL.`

Fältet får då angiven typ. I annat fall användes den implicita typen. Man kan även typdeklarera variabeln på vanligt sätt. Även de senare behandlade datatyperna `COMPLEX` och `DOUBLE PRECISION` kan naturligtvis bilda vektorer.

Från och med Fortran 77 behöver fältet ej längre börja i position 1, man anger då exempelvis

```
DIMENSION C(-7:5)
```

för vektorn `C(-7), C(-6), ..., C(-1), C(0), C(1), ..., C(5).`

Många gamla program utnyttjar restriktionen att index måste börja på ett, varför det matematiska indexet ofta har kanat ett steg. Många gamla programmerare lever likaså kvar i den föreställningen att index börjar på ett.

I deklarationen av dimensionen kan normalt endast konstanter användas. I funktioner och subrutiner kan i vissa sammanhang vanliga heltalsvariabler, eller en asterisk, användas. Även till detta återkommer jag i nästa kapitel.

Deklarationen är ganska ändrad i Fortran 90 jämfört med tidigare (men de gamla möjligheterna finns kvar). I stället för att ge dimensioneringen som ett kommando ger man den nu som ett attribut. Det fanns tidigare följande möjligheter att deklarera en flyttalsvektor med 20 element:

```
DIMENSION A(20)       ! Metod 1, med implicit
                      ! typdeklaration

REAL A(20)            ! Metod 2 (vanligast)

REAL A                ! Metod 3, med explicit
DIMENSION A(20)       ! typdeklation
```

Nu tillkommer den nya möjligheten, som är praktisk i det att alla egenskaper för en viss variabel samlats i en enda rad.

```
REAL, DIMENSION(20)   :: A    ! Metod 4
```

Elementen i en vektor lagras i en entydigt bestämd ordning, om vektorn är deklarerad (j:k) finns elementet (s) i position 1+(s-j). Här användes således det första elementet som referensposition.

```
DIMENSION A(-1:8), B(10:25)
A(2) = B(21)
```

Här identifierar `A(2)` det fjärde elementet i vektorn `A`, och sättes till värdet av `B(21)`, det tolfte elementet i vektorn `B`.

Vid användning av vektorer kan som index användas heltalskonstanter, heltalsvariabler, eller allmänna heltalsuttryck.

Index. I Fortran 66 (och tidigare) kunde index endast vara av en mycket enkel konstruktion, med högst en heltalsvariabel och högst två heltalskonstanter, var och en förekommande högst en gång. Om k och m är konstanter och n variabel så var de tillåtna uttrycken följande: k*n+m, k*n-m, k*n, n+k, n-k, n och k. En motsvarande restriktion införes faktiskt för vissa kommandon i HPF[4], men medför naturligtvis ingen begränsning i Fortran-delen, utan bara i den del som beskriver hur data skall lagras för hög effektivitet på parallella system!

I Fortran användes regeln ovan om uträkning av position normalt även om indexet ligger utanför de givna indexgränserna. Detta innebär att tilldelningarna `A(-2) = 17.0` och `B(9) = A(9)` är tillåtna, men kan få katastrofala följder. Den första av dessa placerar därvid flyttalet -17.0 i den minnescell som ligger närmast före den första i vektorn `A`. Den andra tilldelningen kan faktiskt tolkas som `A(8) = B(10)`, om deklarationerna av `A` och `B` är sådana att vektorerna ligger i ordning, varvid elementet närmast efter `A(8)` blir `B(10)`. Sådan ordning kan tvingas fram av programmeraren genom att använda `COMMON` och/eller `SEQUENCE`.

En del Fortran system innehåller möjlighet att slå på indexkontroll, varvid konstiga tilldelningar av ovanstående natur kan konstateras. Om tilldelningen sker med konstanter kan "felet" hittas redan vid kompileringen, om variabler användes som index kan "felet" normalt konstateras först vid exekveringen. Då måste oftast en avlusare (eng. debugger) användas.

2.9 Enkla slingor (`DO`)

När vi nu har ordnat tillgång till alla dessa tal som kan beskrivas med vektorer gäller det att finna motsvarande kommandon för att effektivt kunna utföra erforderliga beräkningar. Viktiga hjälpmedel är här slingor (`DO`-slingor) och alternativsatser (`IF`-satser, `IF`-konstruktioner och `CASE`-konstruktioner). Alternativsatserna behandlas i nästa avsnitt. En `DO`-slinga kan se ut på fyra olika sätt, först den gamla varianten

[4]HPF = High Performance Fortran är en variant av Fortran för parallell datahantering, se Koelbel [23].

```
      DO 100 index = n1, n2, n3
         ! satser
100   CONTINUE
```

Siffrorna, i detta exempel 100, får väljas godtyckligt från 1 till 99 999, men det får inte finnas mer än en sats CONTINUE med aktuellt nummer. Numret i DO-sats och före CONTINUE måste naturligtvis vara samma. Det är (tyvärr) tillåtet för flera DO-slingor att sluta på samma CONTINUE.

Det är inte absolut nödvändigt att avsluta en gammaldags DO-slinga med en CONTINUE sats. Det går även med en vanlig sats, till exempel en vanlig tilldelningssats, som då har getts motsvarande nummer till vänster. En rekommendation i Fortran 90 och ett krav i Fortran 2003 är att inte avsluta en gammaldags DO-slinga på annat sätt än med CONTINUE. Den nya och rekommenderade varianten är

```
DO index = n1, n2, n3
   ! satser
END DO
```

I båda fallen ovan gäller att storheterna n1, n2 och n3 ej får ändras under slingans utförande. De skall alltså betraktas som konstanter under exekveringen av slingan, men det är tillåtet att de är variabler eller variabeluttryck. Det viktiga är att delresultat under slingans gång, inklusive aktuellt värde av index, ej får påverka dem.

Båda dessa slingor startar med att sätta index till n1 och utföra satserna som följer. När exekveringen når CONTINUE (med rätt nummer) eller END DO hoppar exekveringen upp till DO-satsen igen och räknar upp index med n3. Om resultatet blir större än n2 avbrytes slingan med hopp till satsen efter CONTINUE eller END DO. Annars exekveras satserna ytterligare en gång. Det ovanstående förutsätter att n1 är högst lika med n2 och att n3 är positivt. Om n1 är större än n2 sker hopp direkt till satsen efter CONTINUE eller END DO.

En vanlig misstolkning av Fortran 66 var att satserna i en slinga skall utföras minst en gång. Flera kompilatorer har därför en väljare som kan sättas så att detta sker.

Om ", n3" saknas ovan så fungerar satsen exakt som om n3 vore ett. Värdet av n3 får ej vara noll.

Om n3 är negativ så gäller att ingen exekvering av satserna sker om n1 är mindre än n2. Annars blir det i princip samma sak som om n3 är positiv, det blir nu en nedräkning av n1 till n2.

I Fortran 66 gällde att index och de tre konstanterna skulle vara heltal. Detta ändrades olyckligtvis i Fortran 77 så att även flyttal är tillåtna som index och konstanter. Rekommendation i Fortran 90 är att avstå från flyttal i detta sammanhang. Kommittén har tagit bort den möjligheten i Fortran 2003.

Tidigare (till och med Fortran 66) gällde att värdet på indexet efter normalt genomlöpt slinga var odefinierat, nu (från och med Fortran 77) är det i stället "nästa värde", dvs det första underkända. Många program förutsätter tyvärr heltfelaktigt att det är det sista godkända värdet som är tillgängligt utanför slingan.

Den tredje möjligheten är att bara ha ordet DO på första raden. Man får då i princip en evig slinga, denna kan dock avbrytas med hjälp av de kommandon som beskrivs senare.

```
DO
        ! satser
END DO
```

Den fjärde möjligheten är att ha en `DO WHILE` slinga. Man får då en slinga som utföres så länge som ett visst logiskt villkor är uppfyllt. Till skillnad från parametrarna i den vanliga varianten av `DO`-slingan kan (och bör) storheter i detta villkor ändras under slingans exekvering.

```
DO WHILE (logiskt villkor)
        ! satser
END DO
```

I Fortran 90 tillkom två helt nya kommandon att användas i `DO`-slingor, nämligen `CYCLE` och `EXIT`. Dessa kan läggas in bland de vanliga satserna inne i en `DO`-slinga. Om `CYCLE` påträffas fortsätter exekveringen direkt med nästa värde på indexet respektive direkt från början. Om `EXIT` påträffas fortsätter exekveringen direkt med nästa sats efter `DO`-slingan.

En `DO`-slinga kan tilldelas namn, vilket sker genom att före `DO` ge ett namn följt av ett kolon. Dessutom bör avslutande `END DO` följas av namnet. Kommandona `CYCLE` och `EXIT` kan utnyttja namnet för att upprepa eller avsluta specificerad slinga. Om inget namn ges i `EXIT` eller `CYCLE` avses automatiskt den innersta slinga man just befinner sig i. Dessa kommandon ersätter `GO TO` till den sats som avslutade den gammaldags `DO`-slingan (vilken oftast är en `CONTINUE`-sats).

Även `IF`-konstruktioner och `CASE`-konstruktioner kan tilldelas namn i Fortran 90, liksom `FORALL`-konstruktionen i Fortran 95.

Jag har i ett föredrag [9] redogjort för problemet under Fortran 66 med att slutligt värde på kontollvariabeln `index` är odefinierad genom att visa resultatet av ett mycket enkelt program

```
      DO 20 I = 1, 2
         WRITE(6,10) I
10       FORMAT(1X,I3)
20    CONTINUE
30    WRITE(6,10) I
      STOP
      END
```

på fyra olika system för vilka jag skrivet ut det sista värdet på `I`

```
DEC System 10 med kompilatorn Fortran 10             0
DEC System 10 med kompilatorn Fortran 40             1
IBM 360/75 med kompilatorn FORTRAN IV G              2
IBM 360/75 med kompilatorn FORTRAN IV H EXTENDED     3
```

dvs fyra helt olika resutat från raden med satsnummer 30, det sista överensstämmer med standarden som förutsätter "första underkända värde".

2.10 Nya slingan (FOR ALL)

En ny sats FORALL infördes i Fortran 95 för att till skillnad från en DO-slinga kunna utföras i godtycklig ordning (och därmed parallellt om den fysiska möjligheten finns). Den hämtades från tillägget HPF = Hög-Prestanda Fortran, vilket tillägg [23] aldrig blivit någon succe. Ett par enkla exempel på FORALL-satser följer

```
FORALL (I = 1:N, J = 1:N) H(I,J) = 1.0/REAL(I+J-1)
FORALL (I = 1:N, J = 1:N, Y(I,J) .NE. 0.0) &
                X(I,J) = 1.0/Y(I,J)
FORALL (I = 1:N) A(I,I+1:N) = 4.0*ATAN(1.0)
```

Den första av dessa definierar Hilbertmatrisen av ordning N, den andra inverterar elementen i en matris med undvikande av eventuella nollor. I den tredje tilldelas alla element över huvuddiagonalen i matrisen A ett approximativt värde på π.

I dessa satser kan FORALL sägas vara en dubbelslinga som kan utföras i godtycklig ordning. Det allmänna utseendet är

```
FORALL ( v1 = l1:u1:s1, ... , vn = ln:un:sn, mask ) &
        a(e1, ... , em) = right_hand_side
```

och beräknas enligt väl definierade regler, i princip beräknas alla index först.

Dessutom finns en FORALL-konstruktion. Denna skall tolkas som om de ingående satserna i stället vore skrivna som FORALL-satser i samma ordning. Denna restriktion är väsentlig för att få ett entydigt beräkningsresultat. I en FORALL-konstruktion är anrop av funktioner och subrutiner tillåtna om dessa är rena (eng. pure). Med direktivet PURE kan man ange att så är fallet. Två enkla exempel på en FORALL-konstruktion:

```
REAL, DIMENSION(N,N) :: A, B
...
FORALL (I = 2:N-1, J = 2:N-1)
   A(I,J) = 0.25*(A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))
   B(I,J) = A(I,J)
END FORALL
```

Efter dessa satser har A och B exakt samma värden i alla inre punkter, medan B har kvar sina eventuella gamla värden på ränderna.

```
REAL, DIMENSION(N,N) :: H, B
...
FORALL (I = 1:N, J = 1:N)
H(I,J) = 1.0/REAL(I+J-1)
B(I,J) = H(I,J)**2
END FORALL
```

Först tilldelas alla elementen i Hilbertmatrisen H sina värden, därefter tilldelas matrisen B dessa värden i kvadrat.

Anm. Det har tyvärr visat sig svårt att implementera FORALL på så sätt att den blir effektivare än en vanlig DO-sats.

Övning

(2.2) Skriv en DO-slinga som successivt adderar kvadratroten ur elementen i en vektor om 100 element, men hoppar över negativa element och avslutar additionen om aktuellt element är noll. Kan denna uppgift skrivas enklare med en FORALL-konstruktion?

2.11 Alternativsatser (IF och CASE)

Alternativsatserna består av IF-satser, IF-konstruktioner och CASE-konstruktioner.

Den första av IF-satserna hör intimt samman med den satsnumrering som var nödvändig i vissa sammanhang upp till och med Fortran 77. Den går ut på att de satser man vill "hoppa" till ges ett nummer till vänster om den egentliga satsen, detta nummer skall vara mellan 1 och 99 999 (extremvärdena är tillåtna). Den första varianten av IF-satserna kallas aritmetisk, eftersom den tittar på tecknet hos den betraktade numeriska storheten (heltal eller flyttal). Om denna är negativ sker hopp till det första satsnumret, om den är noll till det andra, och om den är positiv till det tredje satsnumret.

```
        IF (X) 10, 20, 30
 10     ! Satser som utföres om X är negativt
        GO TO 100
 20     ! Satser som utföres om X är noll
        GO TO 100
 30     ! Satser som utföres om X är positivt
100     CONTINUE
        ! Satser som utföres i samtliga fall
```

I ovanstående exempel har jag även introducerat den ovillkorliga hoppsatsen GO TO, vilken gör det möjligt att helt särskilja exekveringen i de tre fallen. Om X är noll sker således först ett hopp till sats 20, där de satser som hör till detta fall utföres, varefter ett hopp sker till sats 100, där satser för samtliga fall kan följa. Många tycker illa om denna IF-sats, eftersom den inte är så där väldigt lättläst. Den har dock den fördelen att det är ett ganska vanligt fall att man skall skilja inte bara på fallen positiv och negativ, utan även måste göra något speciellt åt fallet noll. Satsen är en av dom som är aktuella att tas bort nästa årtusende.

Den andra av IF-satserna är den logiska. Den är mycket enkel till sin natur.

```
        IF (logiskt uttryck) sats
```

Om det logiska uttrycket är sant så utföres satsen, annars exekveras nästa sats direkt. Som sats i en logisk IF-sats användes exempelvis en vanlig tilldelningssats, en ovillkorlig hoppsats eller ett subrutinanrop. Däremot är följande förbjudna i detta sammanhang: DO-slinga, IF-sats eller IF-konstruktion eller CASE-konstruktion.

Naturligtvis får inte heller en END-sats finnas där, men däremot RETURN eller STOP. Dessa senare har jag dock ännu inte diskuterat.

Vi övergår nu till den allmännare IF-konstruktionen. Den ser ut som följer.

```
IF (logiskt uttryck) THEN
        ! Satser som utföres om det logiska
        ! uttrycket är sant
ELSE
        ! Satser som utföres om det logiska
        ! uttrycket är falskt
END IF
```

Här kan ELSE och efterföljande satser utelämnas, så att en förenklad variant erhålles.

```
IF (logiskt uttryck) THEN
        ! Satser som utföres om det logiska
        ! uttrycket är sant
END IF
```

Alternativt kan ELSE utbytas mot ELSE IF, så att en utvidgad variant erhålles.

```
IF (logiskt uttryck) THEN
        ! Satser som utföres om det logiska
        ! uttrycket är sant
ELSE IF (nytt logiskt uttryck) THEN
        ! Satser som utföres om det första logiska
        ! uttrycket är falskt och det andra är sant
ELSE
        ! Satser som utföres om båda de logiska
        ! uttrycken är falska
END IF
```

Notera strukturen hos den allmänna IF-konstruktionen, på första raden finns efter det logiska uttrycket enbart THEN. Den logiska IF-satsen ser likadan ut men istället för THEN finns där en exekverbar sats. De nya raderna efter THEN och både före och efter ELSE är väsentliga för att konstruktionen skall kännas igen.

IF-konstruktionen kan kapslas, på så sätt att man fortsätter med ytterligare ELSE IF många gånger, eller genom att någon eller några av satserna inne i konstruktionen utbytes mot en ny fullständig IF-konstruktion (inklusive sin avslutande END IF).

En IF-sats kan namnsättas. Då bör även motsvarande END IF följas av namnet.

Vi övergår nu till den helt nya CASE-konstruktionen. Den ser ut som följer, och väljer lämpligt fall efter ett skalärt argument av typ INTEGER, LOGICAL eller CHARACTER. Flyttalsargument är således inte tillåtna här. Ett enkelt exempel baserat på en heltalsvariabel IVAR:

```
SELECT CASE (IVAR)
        CASE (:-1)      ! Alla negativa heltal)
                WRITE (*,*) ' Negativt tal'
        CASE (0)                ! Nollfallet
                WRITE (*,*) ' Noll'
        CASE (1:9)      ! Ental
                WRITE (*,*) ' Siffran ', IVAR
```

```
        CASE (10:99)            ! Tvåsiffrigt tal
                WRITE (*,*) ' Talet ',IVAR
        CASE DEFAULT            ! Alla resterande fall
                WRITE (*,*) ' För stort tal'
END SELECT
```

Det är inte tillåtet med överlappande argument i den meningen att ett enda argument satisfierar mer än ett fall av `CASE`. Fallet `DEFAULT` behöver inte finnas med, om inget giltigt alternativ finns så fortsätter exekveringen med första sats efter `END SELECT`. Jag rekommenderar dock att alltid ha med ett `DEFAULT`, som då bör ge en felutskrift om argumentet har ett otillåtet värde.

Dessutom finns två föråldrade villkorliga hoppsatser. Dessa presenteras i avsnitt 14.7, sid 121, samt i Bilaga C.9, sid 180, dvs. styrd hoppsats och tilldelad hoppsats.

Övning

(2.3) Skriv en `CASE`-sats som utför olika beräkningar beroende på om styrvariabeln är negativ, noll, något av de första udda primtalen (3, 5, 7, 11, 13), och inget i övriga fall.

2.12 Variabler med startvärden (`DATA`)

Som jag nämnt tidigare nollställes inte variabler automatiskt från början i Fortran. I princip har det värde som föregående användare lämnade efter sig i aktuell minnescell legat kvar. Vissa kompilatorer har möjlighet till nollställning, eller ännu hellre till att sätta alla variabler till odefinierade vid start. Det finns ett kommando `DATA` i Fortran för att tilldela begynnelsevärden på variabler, denna tilldelning sker då vid inladdning av programmet. `DATA` satsen är därför ej exekverbar, det är meningslöst att hoppa tillbaks till en `DATA` sats för ny initiering. Ny initiering måste ske med omstart av programmet.

```
REAL    :: A
INTEGER :: I, J, K
DATA A /17.0/, I, J, K /1, 2, 3 /
```

Dessa satser tilldelar flyttalet `A` begynnelsevärdet 17.0 och heltalen `I, J` och `K` värdena 1, 2 respektive 3. Man kan även initiera dessa variabler ekvivalent utnyttjande de nya egenskaperna i Fortran 90, utan explicit `DATA`.

```
REAL    :: A = 17.0
INTEGER :: I = 1, J = 2, K = 3
```

För vektorer kan man likaså använda både en gammal variant liksom en ny.

```
REAL VEK1(10)
DATA VEK1 /10*0.0/
REAL, DIMENSION(1:10) :: VEK2 = (/ ( 0.0, I = 1, 10 ) /)
```

Likaså om det är fråga om olika värden som skall tilldelas kan både det gamla och det nya systemet användas.

```
REAL, DIMENSION(3) :: B
DATA B /12.0, 13.0, 14.0/

REAL, DIMENSION(3) :: B = (/ 12.0, 13.0, 14.0 /)
```

I ovanstående konstruktion skall teckenkombinationerna (/ och /) tänkas som parenteser. Avsikten var för Fortan 90 att använda de raka parenteserna [och], men jag påpekade att dessa användes för Ä och Å i svensk 7-bits kod, varför kommittén ändrade till dessa tråkiga kombinationssymboler. Nu i Fortran 2003 är [och] tillåtna (och rekommenderade) alternativ.

2.13 Konstanter (PARAMETER)

Ibland vill man i sitt program ha tillgång till variabler som i själva verket är konstanter. Sådana kallas i Fortran för parametrar, och de har den egenskapen att de ej kan ändra sitt värde under exekveringen.

```
REAL, PARAMETER       :: PI = 3.141592653589793
REAL, PARAMETER       :: PIHALF = PI/2.0
REAL, PARAMETER       :: SQRT_E = 1.648721270700128
```

Den gamla varianten skrevs

```
REAL PI, PIHALF, SQRT_E
PARAMETER (PI = 3.141592653589793)
PARAMETER (PIHALF = PI/2.0)
PARAMETER (SQRT_E = 1.648721270700128)
```

I båda fallen kan således en tidigare parameter utnyttjas vid definitionen av en senare.

2.14 Funktioner

Funktioner är en mycket viktig del i att göra ett beräkningsprogram överskådligt eller strukturerat. Genom att utnyttja en funktion kan man samla en ofta förekommande beräkning till en speciell programenhet. En funktion har ett antal indata och ger som resultat ett utdata "i funktionsnamnet". Ett naturligt exempel är den inbyggda funktionen `SIN(X)`, som har `X` som indata och returnerar motsvarande sinus-värde i sitt namn. I detta avsnitt skall vi bara behandla externa funktioner med skalära utdata. Det finns även vektorvärda funktioner, och även funktioner som ändrar en del av sina indata, liksom rekursiva funktioner. Dessa behandlas senare.

Som första exempel tittar vi på en funktion $f(x)$ som beräknar $e^{-x} \cdot \cos x$. Vi skriver då en programenhet av typ `FUNCTION` och med namnet `F`.

```
REAL FUNCTION F(X)
IMPLICIT NONE
REAL :: X
REAL, INTRINSIC :: EXP, COS
F = EXP(-X) * COS(X)
```

```
      RETURN
      END FUNCTION F
```

I ovanstående funktion är faktiskt det mesta frivilligt. I den första raden behöver ordet `REAL` ej vara med på grund av reglerna om implicit typ-deklaration, jag rekommenderar att ordet är med. Den andra raden är likaså frivillig men jag rekommenderar den varmt. Notera att den måste upprepas i varje programenhet där den skall ha verkan. Den tredje raden blir nödvändig under dessa förutsättningar. Den fjärde raden med deklaration av de inbyggda funktionerna är onödig och jag rekommenderar faktiskt att den inte får vara med. Den är bara av värde i ett mer komplicerat fall. Den femte raden är den egentliga funktionsraden, som tilldelar funktionsvärdet. Notera att det här kallas F, dvs funktionsnamnet utan argument. Den sjätte raden talar om för systemet att exekveringen av programmet är slut och att exekveringen skall återgå till anropande programenhet (ofta huvudprogrammet). Den sista raden talar om för kompilatorn att funktionen är slut.

Att ha med orden `FUNCTION F` ger kompilatorn en möjlighet att klaga om man har strukturerat sitt program fel. Kompilatorn kontrollerar nämligen då att det är en funktion, och att namnet på denna stämmer. Om raden `RETURN` saknas så gäller att Fortran 77 och Fortran 90 fungerar så att `END` återsänder exekveringen till den anropande programenheten. Under Fortran 66 och tidigare fick man kompileringsfel. Jag upprepar att `RETURN` avslutar exekveringen och `END` avslutar kompileringen. Satsen `END` måste därför ligga sist, men satsen `RETURN` behöver inte alls ligga näst sist.

Nu gäller det att skriva ett huvudprogram för att utnyttja denna funktion.

```
      PROGRAM TVA
      IMPLICIT NONE
      REAL :: X, Y
      REAL, EXTERNAL      :: F
1     READ(*,*) X
        Y = F(X)
        WRITE(*,*) X, Y
      GO TO 1
      END PROGRAM TVA
```

Om man vill köra detta exempel på en UNIX-maskin kan man antingen ha båda programmen i samma fil `prog.f90`, eller huvudprogrammet i filen `tva.f90` och funktionen i filen `f.f90`.

Man kompilerar då med

```
f90 prog.f90
```

respektive

```
f90 tva.f90 f.f90
```

och kör med

```
a.out
```

Programmet är skrivet så att det ständigt ber om mer indata. Man kan avbryta med Kontroll-z eller Kontroll-d eller ge ett sådant värde på `X` att programmet bryter på grund av spill i beräkningen av $\exp(-x)$. På Sun blev det avbrott vid -89. Ett "riktigt" program skall naturligtvis innehålla en mer användarvänlig utgång.

Nu följer ett mer meningsfullt exempel, baserad på ett exempel vid en kurs hos IBM 1967. Exemplet går ut på att skriva ett program för att beräkna kubikroten ur ett flyttal. Som bekant kan vi bara beräkna kvadratrötter ur icke-negativa flyttal (om vi håller oss till reella tal), men kubikroten blir en rent udda funktion, ty $(-2)^3 = -8$.

Beräkningen sker nedan med hjälp av omskrivning med $\exp(x)$ och $\ln(x)$. Man kan naturligtvis även använda upphöjt till med `**`, sedan man ordnat att basen (den som skall upphöjas) är positiv.

Vid kompilering av nedanstående program kommer åtminstone NAG:s kompilator att klaga på att den föråldrade aritmetiska `IF`-satsen användes. Jag tycker dock den passar utmärkt här. Man måste nämligen undvika att beräkna $\ln(0)$.

```
            PROGRAM KUBIKROT
            IMPLICIT NONE
            REAL                 :: X, Y
            REAL, EXTERNAL  :: KUBROT
 1          READ(*,*) X
            Y = KUBROT(X)
            WRITE(*,*) X, Y
            GOTO 1
            END PROGRAM KUBIKROT

            REAL FUNCTION KUBROT(X)
            IMPLICIT NONE
            REAL :: X
            LOGICAL :: SKIFTE
            SKIFTE = .FALSE.
            IF (X) 2, 1, 3
 1          KUBROT = 0.0
            RETURN
 2          SKIFTE = .TRUE.

!           Alternativ 1
                X = ABS(X)
 3              KUBROT = EXP(LOG(X)/3.0)

!           Alternativ 2
!3              KUBROT = EXP(LOG(ABS(X))/3.0)

            IF (SKIFTE) KUBROT = -KUBROT
            RETURN
            END FUNCTION KUBROT
```

Beträffande ovanstående program kan man ge två viktiga kommentarer. Den första är att Alternativ 1 är klart olämpligt, eftersom det ändrar inargumentet `X`,

dvs `X` ändras i det anropande programmet, huvudprogrammet. Man bör därför byta till Alternativ 2 (genom att ta bort utropstecknet framför den 3-an samt i stället sätta in utropstecken framför de båda satserna i Alternativ 1).

Den andra kommentaren avser att det kan vara frestande att ändra

```
LOGICAL :: SKIFTE
SKIFTE = .FALSE.
```

till

```
LOGICAL :: SKIFTE = .FALSE.
```

men då använder man en implicit `DATA` sats, vilket ger att `SKIFTE` blir initierat till falskt när programmet startas, men inte varje gång som subrutinen anropas, vilket faktiskt krävs.

Jag har ovan diskuterat den vanliga externa Fortran-funktionen samt något om de inbyggda funktionerna. Dessa båda bör deklareras med `EXTERNAL` respektive `INTRINSIC` då de användes som argument vid funktions- eller subrutinanrop.

Dessutom finns två typer av funktioner som ej kan användas som argument, nämligen satsfunktioner och interna funktioner. Till dessa lokala funktioner återkommer jag i avsnitten 10.3 respektive 10.4, sid 97.

2.15 Subrutiner

Skillnaden mellan funktion och subrutin är att subrutinen i sig inte återför något värde, och därför inte heller har någon typ. En speciell egenskap i Fortran är att en subrutin anropas med `CALL subrutinnamnet`. Liksom funktioner har subrutiner normalt ett eller flera argument, men subrutiner utan argument är tillåtna.

En subrutin utan argument skrives utan parentes, medan en funktion utan argument skrives med tommaparenteser, till exempel `FUN()`.

Man bör undvika att låta en funktion ha någon annan verkan än att returnera ett funktionsvärde, en funktion bör således inte ha som sidoeffekt att vissa av argumenten ändras.

Syftet med en subrutin kan vara flera olika, vanliga fall är

1. Beräkning av ett flertal värden, dvs ändring av en del av argumenten i anropslistan.
2. Utföra in- eller utmatning.
3. Öppna eller stänga filer.
4. Utföra andra mer datorinriktade uppgifter, till exempel att blanka skärmen.

Liksom funktioner kan subrutiner ibland behöva deklareras som `EXTERNAL` (det finns däremot bara sex inbyggda subrutiner, varför användningen av deklarationen `INTRINSIC` är sällsynt för subrutiner).

Jag ger här ett enkelt exempel på en subrutin, nämligen för lösning av en andragradsekvation med koefficienten 1 för andragradstermen (vilket garanterar att det verkligen är en andra-gradare).

```
        SUBROUTINE ANDRA( A, B, X1, X2, NANTAL)
!       LÖS EKVATIONEN X^2 + A*X + B = 0
        IMPLICIT NONE
        REAL      :: A, B, X1, X2
        INTEGER       :: NANTAL  ! Antalet reella rötter
        REAL      :: DISK        ! Diskriminanten
        DISK = A*A - 4.0*B
        IF (DISK > 0.0 ) THEN
                DISK = SQRT(DISK)
                X1 = 0.5*(-A + DISK)
                X2 = 0.5*(-A - DISK)
                NANTAL = 2
                RETURN
        ELSE IF (DISK == 0.0 ) THEN
                X1 = -0.5*A
                NANTAL = 1
                RETURN
        ELSE
                NANTAL = 0
                RETURN
        END IF
        END
```

Subrutinen har således koefficienterna i andragradsekvationen som inparametrar och antalet reella rötter, samt deras eventuella värden, som utparametrar. Även om man använder principen att allt skall deklareras, genom att ge satsen `IMPLICIT NONE` först i programenheten, så är det fortfarande mycket vanligt i Fortran att använda konventionen om att allt som börjar på `I` till `N` är heltal. Därför har antalet rötter getts som `NANTAL` i stället för `ANTAL`. För att använda denna subrutin behövs en anropande programenhet, jag ger den här i form av ett huvudprogram.

```
        PROGRAM HUVUD
!       Huvudprogram för lösning av andragradsekvationen
!       X^2 + B*X + C = 0
        IMPLICIT NONE
        REAL :: B, C, ROT1, ROT2
        INTEGER :: N
        WRITE(*,*) " Ge koefficienterna B och C"
        READ(*,*) B, C
        CALL ANDRA( B, C, ROT1, ROT2, N)
        IF ( N == 2 ) THEN
                WRITE(*,*) " De två rötterna är ", ROT1, ROT2
        ELSE IF ( N == 1) THEN
                WRITE(*,*) " Den enda roten är ", ROT1
        ELSE
                WRITE(*,*) " Inga reella rötter "
        END IF
        STOP
        END
```

I detta program har jag valt att använda olika namn på de verkliga parametrarna i huvudprogrammet, nämligen `B, C, ROT1, ROT2` och `N`, jämfört med de formella parametrarna i subrutinen `A, B, X1, X2` och `NANTAL`. Observera speciellt att den verkliga parametern `B` svarar mot den formella parametern `A` och den verkliga parametern `C` svarar mot den formella parametern `B`. Detta är tillåtet, men naturligtvis förvirrande. Det enklaste är om man har samma namn på de verkliga och formella parametrarna, det näst enklaste om man har helt olika namn. Om de ovanstående programenheterna finns på filerna `andra.f90` och `huvud_andra.f90` kompileras, länkas och körs programmet under UNIX med kommandona

```
f90 huvud_andra.f90 andra.f90 -o program
program
```

2.16 Prioritetsregler

Prioritetsreglerna för de aritmetiska operationerna är ganska självklara, de aritmetiska operationerna indelas i tre grupper med fallande prioritet:

```
**          (upphöjt till)
* och /     (multiplikation och division)
+ och -     (addition och subtraktion)
```

1. Först görs alla "upphöjt till", sedan alla multiplikationer och divisioner, slutligen alla additioner och subtraktioner
2. Parenteser respekteras
3. Operationer med samma prioritet utföres från vänster mot höger, utom vad gäller "upphöjt till", som utföres från höger mot vänster
4. Operationer på variabler av olika typ sker så att den av lägsta typ konverteras till samma typ som den med högre typ
5. En operator kan inte följa direkt efter en annan operator

Prioritetsreglerna för logiska operationer är något mer komplicerade. De logiska operationerna indelas i åtta grupper med fallande prioritet, där de fyra första avser operationer på aritmetiska storheter ingående i ett logiskt uttryck:

```
**                          (upphöjt till)
* och /                     (multiplikation och
                             division)
+ och -                     (addition och subtraktion)
.LT. .LE. .EQ. .NE. .GE. .GT.  (jämförelse)
.NOT.                       (negation)
.AND.                       (skärning)
.OR.                        (union)
.EQV. .NEQV.                (ekvivalens och
                             icke-ekvivalens)
```

Man kan med hjälp av parenteser förändra eller förtydliga prioritetsordningen. Detta rekommenderas varmt i detta fall.

2.17 Sammanfattning

I detta kapitel har de mest grundläggande kommandona i Fortran behandlats. Ett Fortran program består av ett huvudprogram och eventuella funktioner och/eller subrutiner (vi kommer senare att införa ytterligare två typer av programenheter). De variabeltyper som behandlats är bara vanliga flyttal (`REAL`), heltal (`INTEGER`) , logiska variabler (`LOGICAL`) samt ett par exempel på textsträngar (`CHARACTER`). I nästa kapitel införes det oerhört viktiga fältbegreppet, vilket är en generalisering av vektorer upp till högst sju dimensioner, och i de följande kapitlen utvidgas språket efter hand. Det som hittills har behandlats ger mycket begränsade möjligheter att skriva program i Fortran.

Övningar

(2.4) Skriv ett program för tabellering av sinus-funktionen. Skriv gärna huvudprogrammet så att det ber om det intervall för vilket funktionen önskas beräknad. Mata ut en tabell med x och $\sin x$ för $x = 0.0, 0.1, ..., 1.0$. Se till att utmatningen ser snygg ut! Provkör!

(2.5) Skriv ett program som utför en enkel kontroll av de trigonometriska funktionerna genom att beräkna den trigonometriska ettan, och skillnaden mellan denna och en exakt etta. För ett antal argument x skall således uttrycken $\max(\sin^2 x + \cos^2 x - 1)$ och $\min(\sin^2 x + \cos^2 x - 1)$ beräknas.

(2.6) Vad betyder deklarationen `LOGIC ALL`?

(2.7) Är följande deklaration korrekt? `REAL DIMENSION(1:3,2:3) :: AA`

(2.8) Är följande deklaration korrekt? `REAL REAL`

(2.9) Är följande deklaration korrekt? `COMMON :: A`

Kapitel 3

Fält

Inledning

För att lagra dimensionerade variabler användes fält. En vektor kan lagras i ett fält med rangen 1 och ett omfång som är minst lika stort som antalet element i vektorn, medan en matris kräver rangen 2 och tensorer kan ha ännu högre rang. Ett fält (eng. array) definieras att ha ett mönster (eng. shape) eller en form given av dess antal dimensioner, kallad rang (eng. rank), och omfång (eng. extent) för var och en av dessa. Rangen av ett fält har inget med den matematiska rangen för en matris att göra.

Utrymme för flyttalsvektorn a(i), i = 1, 2, ... , 20, reserveras med satsen

```
REAL, DIMENSION(20) :: A
```

och för flyttalsmatrisen b(i,j), i = 1, 2, ... , 10, j = 1, 2, ... , 10, med satsen

```
REAL, DIMENSION(10,10) :: B
```

dvs fältet har nu rangen 2.

I stället för ordet `REAL` kan man använda något av `CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER` eller `LOGICAL`, varvid fältet får angiven typ.

Från och med Fortran 77 behöver fältet ej längre börja i position 1, man anger då exempelvis

```
INTEGER, DIMENSION(-7:5) :: C
```

för heltalsvektorn `c(-7), c(-6), ..., c(-1), c(0), c(1), ..., c(5)` och motsvarande vid högre ordning. Om man vill att fältet skall börja i position ett skriver man till exempel (1:10) eller enklare (10), däremot kan kompileringsfel fås för (:10). I deklarationen av omfång (dimensionen) kan normalt endast konstanter användas (siffror eller `PARAMETER`-storheter). Vi skall senare i detta kapitel diskutera olika former av dynamisk minnesallokering och i vilka fall som dimensionsgränserna får vara variabler.

Elementen i ett fält lagras i en entydigt bestämd ordning, jämför tabell 3.1.

Vi tittar nu på ett vektor-fält och ett matris-fält.

```
REAL, DIMENSION(-1:8)  :: A
REAL, DIMENSION(10,10) :: B
A(2) = B(1,2)
```

Rang	Deklarerad dimension	Index vid anrop	Plats
1	$(j1 : k1)$	$(s1)$	$1 + (s1 - j1)$
2	$(j1 : k1, j2 : k2)$	$(s1, s2)$	$1 + (s1 - j1) + (s2 - j2) * d1$
3	$(j1 : k1, j2 : k2, j3 : k3)$	$(s1, s2, s3)$	$1 + (s1 - j1) + (s2 - j2) * d1 +$ $(s3 - j3) * d1 * d2$

Tabell 3.1: **Indexberäkning i Fortran**. Maximal rang är 7, storheten $di = ki - ji + 1$ anger omfånget (längden) av respektive dimension. Notera att den "sista" di ej ingår i någon formel för beräkning av plats.

Här identifierar `A(2)` det fjärde elementet i fältet `A`, och sättes till värdet av `B(1,2)`, det elfte elementet i fältet `B`.

Ovanstående formler innebär att ett fält `A(I,J)` lagras så att först kommer alla element svarande mot att det andra indexet har lägsta värde, sedan alla element med andra index med näst lägsta värde osv. Lagringen av en matris sker därför kolumn för kolumn, medan matematiker oftast tänker sig matriser lagrade rad för rad.

MINNESREGEL I FORTRAN: Första index varierar snabbast!

Titta på en 2*2 matris och en 4*4 matris

$$A = \begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix} \qquad B = \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix}$$

Elementen är lagrade i fältet `A` med följande ordning, nämligen 11, 21, 12, 22 och i fältet `B` med 11, 21, 31, 41, 12, 22, 32, 42, 13, 23, 33, 43, 14, 24, 34, 44. Detta innebär att den 2*2 matris som "sitter i början av B", dvs i övre vänstra hörnet, och som överenstämmer med 2*2 matrisen `A`, ej är lagrad som de första fyra elementen i fältet `B`.

I Fortran 90 bör man notera att inte alla fält är lagrade på detta enkla sätt. En möjlighet finns till exempel att som argument vid funktions- eller subrutinanrop använda en fältsektion bestående av vartannat element i ett visst fält. Då kommer "avståndet" mellan elementen att vara dubbelt så stort som normalt. Detta problem kan åtgärdas genom mellanlagring. En ytterligare komplikation uppstår vid High Performance Fortran [23] (HPF, se Appendix 9 i internetversionen), nämligen ett behov av att låta elementen i ett fält vara fördelade över de olika processorerna i ett parallellt system.

Alla variabler i Fortran är normalt lokala för varje underprogram. Man kan kommunicera mellan olika underprogram på tre olika sätt, nämligen antingen utnyttjande `COMMON` eller moduler eller argumentlistan.

Om man överför fält utnyttjande `COMMON` måste fälten ha samma utseende i alla underprogrammen. Man måste då deklarera dimensionerna med konstanter, och samma konstanter (samma värden) i alla underprogrammen. Begreppet `COMMON` anses numera föråldrat. Jag återkommer till användning av `COMMON` i kapitel 12, sid 104. Begreppet modul kan ersätta en del av funktionaliteten i `COMMON`, och behandlas i kapitel 13.4, sid 111.

Jag skall nu utförligt diskutera det viktiga begreppet överföring av fält mellan huvudprogram, subrutiner och funktioner utnyttjande argument vid anrop. Under Fortran 77 rådde vissa allvarliga begränsningar i dessa möjligheter, men eftersom många gamla program och programbibliotek utnyttjar dessa metoder är kännedom om dessa väsentlig. I de följande avsnitten diskuterar jag därför först möjligheterna i Fortran 77 och sedan de tillkommande faciliteterna i Fortran 90.

3.1 Överföring av fält som argument utnyttjande Fortran 77

Det finns ett flertal olika möjligheter att överföra fält som argument redan i Fortran 77. Jag behandlar dem här med början med den enklaste metoden för att så småningom komma fram till ganska generella metoder. Jag använder dock i nedanstående exempel i övrigt modern programmeringsteknik utnyttjande de nya möjligheterna i Fortran 90.

Den grundläggande begränsningen i Fortran 90 är att i huvudprogrammet (eller egentligen den överordnade anropande programenheten) måste dimensioneringen ha skett med konstanter och ej med variabler.

3.1.1 Fix dimensionering

Den enklaste metoden är om fältet har identisk (och konstant) dimensionering i de båda aktuella programenheterna, som i följande exempel. Då kan man ha fix dimensionering i de båda programenheterna och bara överföra fältet.

```
        PROGRAM MAIN
        REAL, DIMENSION(20,10) :: A
!          BERÄKNING
        CALL SUB(A)
!          BERÄKNING
        END

        SUBROUTINE SUB(B)
        REAL, DIMENSION(20,10) :: B
!          BERÄKNING I SUBRUTINEN
        RETURN
        END
```

Denna metod tillåter ingen som helst flexibilitet. Subrutinen måste passa exakt mot dimensioneringen i den anropande programenheten.

Alternativt kan man utnyttja `COMMON` med fix dimensionering av fälten. Detta innebär dock ingen fördel, och bör undvikas. Begreppet `COMMON` behandlas i kapitel 12.1, sid 104.

```
        PROGRAM MAIN
        REAL, DIMENSION(20,10) :: A
        COMMON /ARG/ A
!          BERÄKNING
```

```
        CALL SUB
!          BERÄKNING
        END

        SUBROUTINE SUB
        REAL, DIMENSION(20,10) :: B
        COMMON /ARG/ B
!          BERÄKNING I SUBRUTINEN
        RETURN
        END
```

3.1.2 Variabel dimensionering i subrutinen utnyttjande justerbart fält

Om man utnyttjar argumentöverföring kan man nöja sig med en verklig specifikation av dimensionerna i den anropande programenheten, medan man kan använda variabler som specifikation i det anropade underprogrammet (subrutin eller funktion). Överföringen av dimensionerna måste då ske i argumentlistan, ej via `COMMON`.

```
        PROGRAM MAIN
        REAL, DIMENSION(20,10) :: A
!          BERÄKNING
        CALL SUB(A,20,10)
!          BERÄKNING
        END

        SUBROUTINE SUB(B,N,M)
        REAL, DIMENSION(N,M) :: B
!          BERÄKNING I SUBRUTINEN
        RETURN
        END
```

Denna metod med justerbart fält är mycket användbar, eftersom den innebär att samma subrutin kan utnyttjas vid olika dimensionering hos det verkliga argumentet (vid olika anrop av subrutinen).

3.1.3 Förenklad variabel dimensionering i subrutinen

Eftersom Fortran räknar ut platsen för ett viss element utnyttjande en formel, där omfånget (längden) i den sista dimensionen ej ingår, kan subrutinen skrivas med asterisk som sista dimension, dvs vi kan få en viss förenkling genom att utnyttja vad som kallas antagen dimension (assumed-size array).

```
        PROGRAM MAIN
        REAL,DIMENSION(20,10) :: A
!          BERÄKNING
        CALL SUB(A,20)
!          BERÄKNING
        END
```

```
        SUBROUTINE SUB(B,N)
        REAL, DIMENSION(N,*) :: B
!          BERÄKNING I SUBRUTINEN
        RETURN
        END
```

Innan asterisken infördes i samband med Fortran 77 "fuskade" man i stället med att ge en etta (1) för den sista dimensioneringen. Båda dessa varianter innebär att den sista dimensioneringen är ospecificerad. Notera dessutom att om man utnyttjar möjligheten i vissa Fortan system till indexkontroll kan detta ej fungera bra vid antagen dimension. Problemet är att varken * eller 1 känner till den verkliga dimensioneringen i anropande programenhet (om inte systemet är mycket avancerat så att det automatiskt överför korrekt dimensioneringsinformation). Metoden med en etta för den sista dimensioneringen fungerar inte alls under de flesta Fortran 90 system!

3.1.4 Variabel "överdimensionering" i subrutinen

En annan metod, som även den utnyttjar att Fortran använder en väldefinierad formel för att bestämma platsen för ett fältelement, användes ofta i samband med matriser i programbibliotek. Metoden handlar dels om att använda begreppet ledande dimension, dels att låta dimensioneringen ibland vara större än vad som är matematiskt berättigat. Vi inskränker diskussionen till vektorer och matriser, motsvarande gäller vid högre ordning. Vektorer och matriser lagras i endimensionella respektive tvådimensionella fält. Dessa fält måste ha minst en dimension (respektive två dimensioner) som svarar mot den (de) matematiska, men inget hindrar att fälten har deklarerats med större dimension. För en vektor räcker det att motsvarande fält rymmer den matematiska vektorn, för en matris måste dessutom den första dimensioneringsgränsen vara känd, den så kallade ledande dimensionen.

Följande subrutin multiplicerar en matris `C` med en vektor `V`, den enda dimensioneringsinformation som måste överföras är den ledande dimension av det fält som innehåller matrisen `C`. Notera att vi här måste skilja på matematisk dimensionering av vektorer och matriser gentemot programmeringsteknisk dimensionering av fält som skall rymma dessa.

```
        SUBROUTINE MATVEK(C, V, W, N, M, LED_DIM_C)
        INTEGER :: N, M, LED_DIM_C
        REAL, DIMENSION(*) :: V, W
        REAL, DIMENSION(LED_DIM_C,*) :: C

! Beräknar W som produkten av C med V,
! där C är en N gånger M matris i fältet C
! med första dimensionen LED_DIM_C.
! Vektorn V antas ha längden M.
! Vektorn W antas ha längden N.

! Lokala variabler

        INTEGER :: I, J
```

```
      DO I = 1, N          ! Nollställning av produkten
         W(I) = 0.0
      END DO

      DO J = 1, M          ! Beräkning av produkten
         DO I = 1, N
            W(I) = W(I) + C(I,J)*V(J)
         END DO
      END DO
      RETURN
      END
```

Notera att `MATVEK` kräver dels rad- och kolumn-dimensioneringen av matrisen C (matematiska begrepp, storheterna N och M) och dels rad-dimensioneringen av fältet `C` (datatekniskt begrepp, storheten `LED_DIM_C`)). Notera även att ingen automatisk kontroll av dimensioneringen sker i subrutinen, utan måste läggas in explicit i huvudprogrammet.

Ett tillhörande anropande program kan se ut på följande sätt. Här måste således matriser och vektorer (som fält) ges explicita dimensioneringar.

```
      INTEGER IDIM, JDIM
      PARAMETER (IDIM=50, JDIM=40)
      INTEGER N, M
      REAL,     DIMENSION(IDIM,JDIM)      :: A
      REAL,     DIMENSION(JDIM) :: X
      REAL,     DIMENSION(IDIM) :: Y

      WRITE(*,*) ' Ge dimensionen för vektorn X '
      READ(*,*) M
      WRITE(*,*) ' Ge dimensionen för vektorn Y '
      READ(*,*) N
      IF ( N < 1 .OR. N > IDIM .OR. &
           M < 1 .OR. M > JDIM ) THEN
              WRITE(*,*) ' Felaktig dimensionering '
      ELSE
!   Bestämning av vektorn X och matrisen A
!   bör ske här.
              CALL MATVEK(A, X, Y, N, M, IDIM)
              WRITE(*,*) ' Vektorn Y = AX '
              WRITE(*,*) (Y(I), I = 1, N)
      END IF
      STOP
      END
```

3.1.5 "Överdimensionering" i huvudprogrammet

Med överdimensionering menar jag att man explicit använder ett större omfång (antal element i en viss dimension) än som är sakligt motiverat. På detta sätt kan ett program användas av alla storlekar på ett problem upp till och med en viss högsta gräns.

En variant av föregående metod innebär att lagringsutrymme för aktuell matris skapas i huvudprogrammet, men att matrisen aldrig användes i huvudprogrammet utan bara i subrutiner och funktioner. Metoden innebär att matrisen aldrig är tillgänglig på ett normalt sätt i huvudprogrammet, och förutsätter därför att den ej heller användes där, ej ens för utmatning. Däremot måste matrisens matematiska dimension bestämmas i huvudprogrammet, eller i en speciell subrutin, innan anrop av någon av de subrutiner som innehåller aktuell matris kan ske.

```
      PROGRAM MAIN
      INTEGER :: N
      REAL, DIMENSION(20,20) :: A
      CALL SUBN(N)
      CALL SUB1(A,N)
      CALL SUB2(A,N)
      CALL SUB3(A,N)
      END

      SUBROUTINE SUBN(N)
      INTEGER :: N
      DO
        WRITE(*,*) ' Ge aktuell dimension '
        READ (*,*) N
        IF ( N < 1 .OR. N > 20 ) THEN
              WRITE(*,*) ' Felaktig dimension '
        ELSE
              EXIT
        END IF
      END DO
      END

      SUBROUTINE SUB1(B,M)
      REAL, DIMENSION(M,M) :: B
!        BERÄKNING I SUBRUTINEN
      RETURN
      END

      SUBROUTINE SUB2(C,L)
      REAL, DIMENSION(L,L) :: C
!        BERÄKNING I SUBRUTINEN
      RETURN
      END

      SUBROUTINE SUB3(D,K)
      REAL, DIMENSION(K,K) :: D
!        BERÄKNING I SUBRUTINEN
      RETURN
      END
```

I subrutinen `SUBN` har satsen `EXIT` den funktionen att den vid ett acceptabelt

värde på dimensionen N avbryter den "eviga" slingan. Subrutinen ger då återhopp till huvudprogrammet.

Jag upprepar att matrisen `A` är lagrad på ett onormalt sätt i huvudprogrammet (jämfört med lagringen i subrutinerna), om inte `N` råkar vara just 20. Utnyttjande bara första index vid beräkningen, dvs `A(i+(j-1)*N, 1)`, kan man dock få fram rätt värde på elementet i den i-te raden och den j-te kolumnen. Däremot ger `A(i,j)` fel värde, nämligen uträknat som `A(i+(j-1)*20, 1)`.

3.2 Överföring av fält som argument utnyttjande Fortran 90

Det finns ett flertal olika tillkommande möjligheter att överföra fält som argument i Fortran 90. Endast den första av de tillkommande metoderna är dock enkel.

3.2.1 Automatiskt fält

Ett automatiskt fält, "automatic array" är mycket likt fallet med variabel dimensionering i subrutinen (avsnitt 3.1.2, sid 31). Det är dock en mycket väsentlig skillnad i att med ett automatisk fält överföres inget fält från en rutin till en annan, utan ett lokalt fält (arbetsutrymme) skapas. I Fortran 77 där dynamisk minnesallokering saknades var man i stället tvungen att överföra arbetsareor i argumentlistan, varför i nedanstående exempel även `X` måste vara med i argumentlistan (och vara allokerat i anropande programenhet).

```
PROGRAM HUVUD
INTEGER :: N, K
N = 3
K = 17
CALL SUB(N, K, 2*K+N*N)
END

SUBROUTINE SUB (I, J, K)
INTEGER :: I, J, K
REAL, DIMENSION (I, J, K) :: X
Inne i subrutinen kan fältet X användas
RETURN
END
```

Dimensioneringen för `X` hämtas från heltalen i det anropande programmet. Automatiskt fält är mycket praktiskt för arbetsareor i subrutiner och funktioner. De skapas automatiskt (dynamisk minnesallokering) och försvinner likaså automatiskt vid uthopp ur programenheten. Arbetsareor vid utnyttjande av programbibliotek var tidigare en stor administrativ börda vid programmeringen.

3.2.2 Antaget mönster hos ett fält

Antaget mönster, "assumed-shape array" är ett mycket kraftfullt och användbart begrepp. Lagringen definieras i den anropande proceduren, där även allokering

sker, och i den anropade programenheten behöver endast typ, rang och namn ges. Dock krävs ett explicit gränssnitt i den anropande programenheten för att överföra dimensioneringsinformationen. Det hela kan se ut som följer, i exemplet kan fältet `B` från huvudprogrammet användas under namnet `A` i subrutinen, där inga dimensioneringsgränser getts.

Gränssnittet har till uppgift att tala om för systemet att dimensioneringsinformationen i anropande programenhet skall överföras automatiskt till den anropade programenheten. Detta sker medelst `INTERFACE`.

```
PROGRAM HUVUD

INTERFACE
        SUBROUTINE SUB(A)
        REAL, DIMENSION (:,:,:) :: A
        END SUBROUTINE SUB
END INTERFACE

REAL, DIMENSION(1:20,1:12,-3:7) :: B
...
CALL SUB(B)
...
END PROGRAM HUVUD

SUBROUTINE SUB (A)
REAL, DIMENSION(:, :, :) :: A
...
END SUBROUTINE SUB
```

3.3 Dynamisk minneshantering

Det finns fyra olika sätt att göra dynamisk minnesallokering i Fortran 90.

I Fortran 77 kan dynamisk minnesallokering egentligen ej ske, men den simuleras ibland genom att utrymme allokeras redan i den anropande programenheten, och både fältnamn och erforderlig dimensionering finns med i anropet, justerbart fält, avsnitt 3.1.2. En förenklad variant är där den sista dimensioneringen ges med en * i deklarationen, antagen dimension, avsnitt 3.1.3.

Nu tillkommer, förutom de redan diskuterade automatiskt fält, avsnitt 3.2.1, och antaget mönster, avsnitt 3.2.2, de båda ytterligare möjligheterna allokerbart fält och användning av fält utnyttjande pekare.

3.3.1 Allokerbart fält

Den enklaste metoden att göra dynamisk minnesallokering är att använda ett allokerbart fält, "allocatable array", dvs att med de båda satserna `ALLOCATE` och `DEALLOCATE` erhålla respektive återlämna lagringsutrymme för ett fält vars typ, rang och namn (och andra attribut) tidigare har deklarerats. Notera att båda dessa satser kräver att fältnamnet ges inom parentes, vid allokering skall det dessutom förses med explicit dimensioneringsinformation.

En ytterligare förutsättning är att aktuellt fält deklarerats med attributet ALLOCATABLE, vilket talar om för systemet att dynamisk minnesallokering kommer att ske.

```
REAL, DIMENSION(:), ALLOCATABLE :: X
...
ALLOCATE(X(N:M)) ! N och M är heltalsuttryck
                 ! Ge båda gränserna eller bara
                 ! den övre och då utan kolon!
...
X(J) = Q
CALL SUB(X)
...
DEALLOCATE (X)
...
END
```

Deallokering av lokala fält förekommer automatiskt i en subrutin eller funktion (om ej attributet SAVE getts) när man når RETURN eller END.

3.3.2 Allokering av fält med pekare

Pekare diskuteras närmare i kapitel 11, men tillämpningen på allokering av fält är så pass enkel att den kan ges redan nu. Ett fält som skall allokeras med hjälp av pekare måste vid deklarationen ges attributet POINTER för pekare. Vid lämpligt tillfälle sker sedan allokering på vanligt sätt med kommandot ALLOCATE. Skillnaden mellan de båda metoderna är således att den första kräver attributet ALLOCATABLE och den andra attributet POINTER.

För att kunna deklarera med pekare i huvudprogrammet och sedan allokera ett utrymme i subrutinen krävs användning av ett gränssnitt eller INTERFACE i huvudprogrammet.

På detta sätt erhålles i nedanstående exempel en dynamisk minnesallokering av vektorn B i subrutinen, och den kan även användas som vektorn A i huvudprogrammet. Storleken på vektorn bestäms i subrutinen. Gränssnittet är i huvudsak en upprepning av deklarationen i subrutinen. Det har till uppgift att informera huvudprogrammet om att verklig dimensioneringsinformation kan erhållas ur subrutinen.

Detta är en bra metod att föra en dimension uppåt i anropskedjan. Man kan naturligtvis i stället "fuska" genom att ha en subrutin som bestämmer dimensionen, hoppa tillbaks till huvudprogrammet och där göra en allokering av ett ALLOCATABLE fält, för att därefter anropa en annan subrutin som kan utnyttja det nu skapade fältet.

```
PROGRAM HUVUD
INTERFACE
        SUBROUTINE SUB(B)
        REAL, DIMENSION (:), POINTER :: B
        END SUBROUTINE SUB
END INTERFACE

INTEGER :: N
```

```
        REAL, DIMENSION (:), POINTER :: A
!   Här har vektorn A ännu ej fått någon dimensionering!
        CALL SUB(A)
!   Använd nu vektorn A, den har nu fått dimensionering
!   och även tilldelats värden i subrutinen SUB
        N = SIZE(A)         ! N är nu dimensionen av A
        STOP
        END PROGRAM HUVUD

        SUBROUTINE SUB(B)
        REAL, DIMENSION (:), POINTER :: B
        INTEGER :: I, M
        WRITE(*,*) " Tilldela nu en dimension till vektorn"
        READ(*,*) M
        IF (M < 1 ) M = 1
        ALLOCATE (B(M))
        DO I = 1, M
           B(I) = 17.0*I + 3.0*I**2 - 1.0/I
        END DO
        RETURN
        END SUBROUTINE SUB
```

En alternativ metod att föra dimensionering uppåt i anropskedjan är användning av en modul med ett allokerat och sparat `SAVE`d fält. Ett enkelt exempel på detta förfarande för en vektor finns här.

```
MODULE DYNA
IMPLICIT NONE
REAL, DIMENSION(:), ALLOCATABLE, SAVE :: ARBETE
END MODULE DYNA

PROGRAM TEST_AV_DYNAMIK
USE DYNA
IMPLICIT NONE
PRINT *, 'MAIN'
CALL SUB1
CALL SUB2
PRINT *, 'MAIN'
END PROGRAM TEST_AV_DYNAMIK

SUBROUTINE SUB1
USE DYNA
IMPLICIT NONE
! Storleken av vektorn ARBETE bestäms här
INTEGER :: I
REAL, DIMENSION(5) :: A, B
PRINT *, 'SUB1'
        A = 1.0
B = (/ (3.0, I = 1, 5) /)
ALLOCATE ( ARBETE(SIZE(B)) )
```

```
ARBETE = A
        WRITE(*,*) A
        WRITE(*,*) B
        WRITE(*,*) ARBETE
END SUBROUTINE SUB1

SUBROUTINE SUB2
USE DYNA
IMPLICIT NONE
! Vektorn ARBETE används här
REAL :: B
REAL, DIMENSION(:), ALLOCATABLE :: A
PRINT *, 'SUB2'
ALLOCATE ( A(SIZE(ARBETE)) )
        A = 2.*ARBETE
B = SUM(ARBETE)
ARBETE = A
        WRITE(*,*) A
        WRITE(*,*) B
        WRITE(*,*) ARBETE
END SUBROUTINE SUB2
```

3.4 Fältoperationer

Redan i Fortran 77 fanns det viss möjlighet att operera på ett helt fält utan att behöva specificera dessa element för element. Detta kunde dock bara utnyttjas vid in- och utmatning. En hel matris kan exempelvis skrivas ut med följande kommando,

```
REAL, DIMENSION(10,10) :: MATRIS
WRITE(*,*) MATRIS
```

vilket är mycket effektivare, både vad gäller tidsåtgång och erforderligt lagringsutrymme (om skrivningen i stället sker till fil), jämfört med det explicita anropet av varje element.

```
REAL, DIMENSION(10,10) :: MATRIS
DO I = 1, 10
   DO J = 1, 10
      WRITE(*,*) MATRIS(I,J)
   END DO
END DO
```

eller, med en implicit slinga för kolumnerna, varvid varje rad av matrsen skrives ut på en rad på "papperet". I ovanstående exempel kommer i stället varje element på egen rad.

```
REAL, DIMENSION(10,10) :: MATRIS
DO I = 1, 10
   WRITE(*,*) (MATRIS(I,J), J = 1, 10)
END DO
```

Ovanstående kan enklare skrivas

```
REAL, DIMENSION(10,10) :: MATRIS
DO I = 1, 10
   WRITE(*,*) MATRIS(I,:)
END DO
```

3.4.1 Enkla fältoperationer

Ett fält (eng. array) definieras att ha ett mönster (eng. shape) given av dess antal dimensioner, rang (eng. rank) och omfång (eng. extent) för var och en av dessa. Två fält överensstämmer om de har samma mönster. Man kan då utföra fältuttryck (eng. array assignment). Operationer sker normalt på element för element bas! De elementära funktionerna fungerar således på detta sätt, element för element, i Fortran.

```
REAL, DIMENSION(5,20)    :: X, Y
REAL, DIMENSION(-2:2,20) :: Z
:
Z = 4.0*SIN(Y)*SQRT(X)
```

Vi kanske här vill skydda oss för negativa element i `X`. Detta sker genom

```
WHERE ( X >= 0.0 )
        Z = 4.0*SIN(Y)*SQRT(X)
ELSEWHERE
        Z = 0.0
END WHERE
```

Notera att `ELSEWHERE` måste vara i ett ord! *Det är faktiskt det enda Fortran-ord som man tycker är naturligt att dela upp i två ord som den uppdelningen icke är tillåten för.*

Med fältsektion menas en del av ett fält. Om fältet `A` är deklarerat med

```
REAL, DIMENSION(-4:0,7) :: A
```

så väljer `A(-3,:)` ut hela den andra raden, medan `A(0:-4:-2, 1:7:2)` väljer i omvänd ordning ut vartannat element i varannan kolumn. Liksom variabler kan bilda fält, så kan även konstanter det.

```
REAL, DIMENSION(6) :: B
REAL, DIMENSION(2,3) :: C
B = (/ 1, 1, 2, 3, 5, 8 /)
C = RESHAPE( B, (/ 2,3 /) )
```

där det första argumentet till den inbyggda funktionen `RESHAPE` ger värdena och det andra argumentet ger det nya mönstret. Ytterligare två argument finns som möjliga till denna funktion.

I följande mycket enkla exempel visar jag hur man kan tilldela matriser med satser av typ `B = A`, dvs utnyttja den inbyggda matrismultiplikationen `MATMUL` (denna multiplicerar inte element för element utan efter reglerna för matrismultiplikation) och fältadderaren `SUM` (som adderar samtliga element i fältet), samt utnyttja fältsektioner (i exemplet av typ vektorer).

```
PROGRAM FALT
        IMPLICIT NONE
        INTEGER                 :: I, J
        REAL, DIMENSION (4,4)     :: A, B, C, D, E
        DO I = 1, 4         ! Beräkna en test matris.
                DO J = 1, 4
                        A(I,J) = (I-1.2)**J
                END DO
        END DO
        B = A*A           ! Element för element multiplikation.
        CALL SKRIV(A,4)
        CALL SKRIV(B,4)
        C = MATMUL(A,B)   ! Inbyggd matris-multiplikation.
        DO I = 1, 4       ! Explicit matris-multiplikation
                          ! utnyttjande fältsektion för
                          ! den inre slingan.
                DO J = 1, 4
                        D(I,J) = SUM( A(I,:)*B(:,J) )
                END DO
        END DO
        CALL SKRIV(C,4)
        CALL SKRIV(D,4)
        E = C - D   ! Jämförelse av de två metoderna.
        CALL SKRIV(E,4)
END PROGRAM FALT

SUBROUTINE SKRIV(FALT,N)
        Generell rutin för utskrift av en flyttalsmatris.
        IMPLICIT NONE
        INTEGER                 :: N, I
        REAL, DIMENSION (N,N)   :: FALT
        DO I = 1, N
                WRITE(*,'(5E15.6)') FALT(I,:)
        ! Samtliga värden i matrisens i:te rad skrives ut,
        ! med (högst) fem värden per utskriftsrad).
        END DO
        WRITE(*,*)                ! Skriv en blank rad.
        END SUBROUTINE SKRIV
END SUBROUTINE SKRIV
```

Övning

(3.1) Skriv en rutin för lösning av ett linjärt ekvationssystem, utnyttjande Gausselimination med partiell pivotering.

Kapitel 4

Funktioner och subrutiner

Inledning

I detta kapitel skall vi gå igenom de regler som gäller för funktioner och subrutiner i Fortran 90. Vi skall härvid skilja på inbyggda, externa och lokala funktioner och subrutiner.

4.1 Inbyggda funktioner och subrutiner

De inbyggda funktionerna och subrutinerna kallas i Fortrans engelska språkbruk för "intrinsic", vilket är ett starkare uttryck än det ganska intetsägande built in". Min engelsk-svenska ordbok översätter "intrinsic" med inre eller inneboende. Det utmärkande för de inbyggda funktionerna är att någon speciell åtgärd ej behöver vidtas för att länka in dem. De behöver ej heller deklareras, utom då de användes som argument vid anrop av andra funktioner eller subrutiner, varvid de skall deklareras att vara just `INTRINSIC`. För ett exempel på detta se avsnitt 4.2.1, Funktioner som argument.

4.1.1 Funktioner

Bland de inbyggda funktionerna finns främst de elementära matematiska funktionerna (typ sinus), samt några funktioner som kallas numeriska (typ absolutbelopp). En kort diskussion av dessa fanns redan i avsnitt 2.5, en fullständig presentation finns i Bilaga D, sid 187. Dessutom finns där presentation av de många övriga inbyggda funktionerna.

De inbyggda funktionerna är generiska, dvs datatypen hos argumentet anger vilken rutin som skall väljas. Under Fortran 66 var man tvungen att skriva `SIN(X)` om `X` var `REAL, DSIN(X)` om `X` var `DOUBLE PRECISION` och slutligen `CSIN(X)` om `X` var `COMPLEX`. Från Fortran 77 fungerar `SIN(X)` i alla tre fallen. Om funktionsnamnet användes som argument vid funktions- eller subrutinanrop måste däremot det explicita namnet anges.

4.1.2 Subrutiner

De inbyggda subrutinerna kom först med Fortran 90 och var bara fem stycken, två för tidsangivelser, en för bitkopiering och två för pseudoslumptal. Med Fortran 95 kom ytterligare en inbyggd subrutin, nämligen för CPU-tid. Dessa presenteras fullständigt i slutet av Bilaga D.21, sid 204.

4.2 Externa funktioner

De externa funktionerna och subrutinerna är de man närmast tänker på när man talar om funktioner och subrutiner i Fortran. De bildar egna programenheter, som kan kompileras separat. Data kan överföras med argument eller utnyttjande ett `COMMON` block, eller utnyttjande externa filer. Den viktigaste metoden är att data överföres via argument.

4.2.1 Funktioner

Liksom i matematiken så är i Fortran en funktion något som har ett eller flera argument och som returnerar ett funktionsvärde. Från Fortran 90 kan funktioner vara rekursiva och/eller fältvärda. Det finns även vissa funktioner som saknar argument.

Vanliga enkla funktioner diskuterades redan i avsnitt 2.14, sid 21. Vi skall därför här diskutera funktioner utan argument, fältvärda funktioner och rekursiva funktioner. Vidare skall vi ta upp ävsikten"hos olika argument, frivilliga argument och argument med nyckelord. Användning av funktioner som argument skall likaså diskuteras. Vi börjar med det senare.

Funktioner som argument

För att illustrera användningen av funktioner som argument vid anropet av en annan funktion väljer vi att titta på ett program för bestämning av nollstället till en funktion f(x) av en reell variabel, utnyttjande intervallhalveringsmetoden. För att förenkla programmet förutsätter vi att det betraktade intervallet är $[a, b]$, och att $a < b, f(a) > 0, f(b) < 0$, samt att $\epsilon > 0$.

```
        REAL FUNCTION NOLL(F, A, B, EPS)
        IMPLICIT NONE
! Beräknar nollställe till en funktion f(x) om
! a < b och f(a) > 0 och f(b) < 0 och eps > 0.
        REAL :: F, A, B, EPS
        REAL :: V, H, MITT
        V = A
        H = B
        DO
                IF ( H - V >= EPS ) THEN
                        MITT = 0.5*(V+H)
                        IF ( F(MITT) > 0.0 ) THEN
                                V = MITT
                        ELSE
                                H = MITT
```

```
            END IF
            CYCLE     ! Ny iteration behövs.
        END IF
        EXIT    ! Konvergens har erhållits.
END DO
NOLL = 0.5*(V+H)
RETURN
END FUNCTION NOLL
```

Funktionen ovan fungerar så att först testas om vänster- och högerpunkterna ligger tillräckligt nära varandra, i så fall utförs inte `IF`-satsen och den "eviga" `DO`-slingan avbrytes med `EXIT`. Ett slutligt nollställe beräknas som medelvärdet av vänster- och högerpunkterna. Om punkterna ej ligger tillräckligt nära beräknas en mittpunkt, som väljes som ny vänster- eller högerpunkt, och en ny iteration begäres med `CYCLE`. Funktionen täcker som synes inte alla fall, och svarar inte alltid så tidigt som möjligt. Anledningen är att vi sökt göra den så enkel som möjligt för att illustrera hur funktioner med funktioner som argument fungerar. Ett par utvidgningar föreslås i övningarna nedan.

Den funktion `F(X)` som användes som argument till funktionen `NOLL` måste också finnas, och då varken som intern funktion eller som satsfunktion, utan som en extern funktion, eller som en inbyggd funktion. Ett exempel på en extern funktion ges nedan, med namnet `G`.

```
REAL FUNCTION G(X)
IMPLICIT NONE
REAL :: X
G = COS(X) - LOG(X)
RETURN
END FUNCTION G
```

För att använda dessa funktioner kräves även ett huvudprogram. Härvid är det väsentligt, på grund av Fortrans separatkompilering, att informera systemet om att funktionen `G(X)` är en extern funktion, vilket sker med satsen `EXTERNAL G`.

```
PROGRAM NOLLSTAELLE
IMPLICIT NONE
REAL :: NOLL, A, B, G, EPS, Y
EXTERNAL G
A = 1.0
B = 2.0
EPS = 2.0E-7
Y = NOLL(G, A, B, EPS)
WRITE(*,*) Y
STOP
END PROGRAM NOLLSTAELLE
```

Om funktionen i stället är en inbyggd funktion användes satsen `INTRINSIC` funktion. I exemplet ovan passar det bra att byta ut satsen `EXTERNAL G` mot `INTRINSIC COS` och satsen `Y = NOLL(G, A, B, EPS)` mot `Y = NOLL(COS, A, B, EPS)`, varvid programmet räknar ut att $\pi/2$ är ett nollställe till cosinus. I detta fall behövs naturligtvis inte den externa funktionen `G(X)`. Notera även att den inbyggda funktionen ej skall typdeklareras.

Övningar

(4.1) Modifiera funktionen `NOLL` till att göra ett direkt uthopp med rätt nollställe som resultat om `F(MITT)` råkar bli exakt noll vid beräkningen.

(4.2) Modifiera funktionen `NOLL` till att klara även andra teckenkombinationer på $f(a)$ och $f(b)$.

(4.3) Modifiera funktionen `NOLL` till att använda en mer avancerad algoritm än intervallhalvering, till exempel regula falsi.

Funktioner utan argument

En funktion kan faktiskt helt sakna argument. Skillnaden mot en subrutin är då främst att anropet sker med funktionsnamnet, och ej med `CALL` subrutinnamnet. Vanliga exempel på funktioner som fungerar bra utan argument är sådana som returnerar aktuell tidpunkt eller ett slumptal. De skrives med en tom argumentlista, men parentesen måste vara med både i funktionsdefinitionen och i funktionsanropet.

Som exempel har jag skrivet en enkel funktion utan argument. För att få ut något resultat ur den har jag valt att använda tidsrutinen i Fortran 90 (se Bilaga D). I heltalsvektorn `VALUES` lagras år, datum och tidpunkt, vilka jag summerar med fältadderaren `SUM` för att få något som påminner om ett slumptal. Anropet av tidsrutinen sker med nyckelordsargument, vilket förklaras senare.

```
PROGRAM TEST_INGET_ARGUMENT
IMPLICIT NONE
REAL :: NOLL_ARG, Y
Y = NOLL_ARG()
WRITE(*,*) Y
STOP
END PROGRAM TEST_INGET_ARGUMENT

REAL FUNCTION NOLL_ARG()
IMPLICIT NONE
INTEGER, DIMENSION(8) :: VALUES
CALL DATE_AND_TIME(VALUES=VALUES)
NOLL_ARG = SUM(VALUES)
RETURN
END FUNCTION NOLL_ARG
```

Rekursiva funktioner

En helt ny möjlighet i Fortran 90 är rekursion. Notera att det krävs att man i funktionsdeklarationen ger en helt ny egenskap `RESULT (utvariabel)`. Denna resultat-variabel eller utvariabel användes inuti funktionen för att lagra funktionens värde. Vid själva anropet av funktionen, både externt och internt, användes däremot i stället det yttre eller gamlafunktionsnamnet. Användaren kan därför vid anrop av den rekursiva funktionen strunta i att det finns en resultat-variabel. Här följer två exempel, dels rekursiv beräkning av fakulteten, dels rekursiv beräkning av Fibonacci-talen. Den senare är mycket ineffektiv.

```
RECURSIVE FUNCTION FAKULTET(N) RESULT (FAK_RESULTAT)
IMPLICIT NONE
INTEGER, INTENT(IN)   :: N
INTEGER                 :: FAK_RESULTAT
IF ( N <= 1 ) THEN
        FAK_RESULTAT = 1
ELSE
        FAK_RESULTAT = N * FAKULTET(N-1)
END IF
END FUNCTION FAKULTET

RECURSIVE FUNCTION FIBONACCI(N) RESULT (FIBO_RESULTAT)
IMPLICIT NONE
INTEGER, INTENT(IN)   :: N
INTEGER                 :: FIBO_RESULTAT
IF ( N <= 2 ) THEN
        FIBO_RESULTAT = 1
ELSE
        FIBO_RESULTAT = FIBONACCI(N-1) + FIBONACCI(N-2)
END IF
END FUNCTION FIBONACCI
```

Anledningen till att ovanstående beräkning av Fibonacci-talen blir så ineffektiv är att anrop med ett visst värde på N genererar två anrop av rutinen, som i sin tur generar fyra, osv. Gamla värden (anrop) återutnyttjas ej!

En intressantare användning av rekursiv teknik är beräkning av exponentialfunktionen av en matris. I stället för det omedelbara uttrycket med successiv multiplikation med en matris kan man använda en rekursiv metod där man plockar ut 2-potenser för att optimera beräkningen. Rekursion är vidare utmärkt för att koda adaptiva algoritmer, se Övning (4.5) nedan.

En annan viktig användning av det nya begreppet resultat-variabel är vid fält-värda funktioner, då det är lätt att deklarera denna variabel att lagra funktionens värde(n). Det är faktiskt kombinationen rekursivitet och fält som tvingat fram det nya begreppet.

I programmet ovan användes det nya begreppet "avsikt" eller `INTENT`, vilket förklaras i avsnitt 4.2.1, sid 48.

Övningar

(4.4) Skriv en rutin för beräkning av tribonacci-talen. Dessa bildas som Fibonacci-talen, men man utgår från tre tal (alla 1 vid starten) och adderar hela tiden de tre senaste talen för att få nästa. Provkör och beräkna `TRIBONACCI(15)`. Notera att beräkningstiden växer mycket snabbt med argumentet.

(4.5) Skriv en adaptiv rutin för kvadratur, dvs beräkning av den bestämda integralen över ett visst intervall.

Fältvärda funktioner

En helt ny möjlighet i Fortran 90 är fältvärda funktioner. Notera att man även här måste utnyttja den nya egenskapen `RESULT (utvariabel)`, se avsnittet om rekursiva funktioner i 4.2.1, sid 45. Denna användes inuti funktionen för att på ett enkelt sätt deklarera funktionens typ. Vid själva anropet av funktionen, både externt och internt, användes däremot i stället det yttre eller gamlaff-unktionsnamnet. Användaren kan därför vid anrop av den fältvärda funktionen strunta i att det finns en resultat-variabel.

Fältvärda funktioner kräver den tidigare diskuterade nyheten, nämligen ett explicit gränssnitt (`INTERFACE`) i den anropande programenheten. Gränssnittet består i princip av deklarationerna i funktionen, se exemplet nedan. Gränssnitt kräves vid flera olika tillfällen och är faktiskt det svåraste i Fortran 90 (om man inte redan kan Ada). Se vidare avsnittet 10.2, sid 97.

Nedanstående enkla funktion ger som resultat en vektor med de första åtta potenserna av det skalära argumentet.

```
            PROGRAM TEST_FAELTVAERD_FUNKTION
            IMPLICIT NONE
            INTERFACE   ! Detta är gränssnittet för den
! fältvärda funktionen POTENS(X).
                  FUNCTION POTENS(X) RESULT(FAELT)
                  REAL :: X
                  REAL, DIMENSION(8) :: FAELT
                  END FUNCTION POTENS
            END INTERFACE
            REAL :: X
            REAL, DIMENSION(8) :: Y
            X = 2.0
            Y = POTENS(X)
            WRITE(*,*) Y
            STOP
            END PROGRAM TEST_FAELTVAERD_FUNKTION

            FUNCTION POTENS(X) RESULT(FAELT)
            IMPLICIT NONE
            REAL :: X
            REAL, DIMENSION(8) :: FAELT
            INTEGER :: I
            DO I = 1, 8
               FAELT(I) = X**I
            END DO
            RETURN
            END FUNCTION POTENS
```

Vi kan ganska enkelt justera ovanstående funktion till att klara godtyckliga potenser, där ordningen ges endast som indata i huvudprogrammet. Dimensionen överföres i nästa exempel automatiskt med argumentet, en flyttalsvektor, och dimensionen för resultatet sättes lika med denna dimension. Det går naturligtvis minst lika bra att överföra dimensionen med ett vanligt heltalsargument.

Det är inte nödvändigt att här låta argumentet vara en flyttalsvektor. Man kan även klara sig med ett skalärt argument, men då måste man använda pekare vid deklarationen av fältet, se sektion 3.3.2, sid 37.

```
PROGRAM NY_TEST_FAELTVAERD_FUNKTION
IMPLICIT NONE
INTERFACE
      FUNCTION POTENS(X) RESULT(FAELT)
      REAL, DIMENSION(:) :: X
      REAL, DIMENSION(SIZE(X)) :: FAELT
      END FUNCTION POTENS
END INTERFACE
REAL, DIMENSION(:), ALLOCATABLE :: X
REAL, DIMENSION(:), ALLOCATABLE :: Y
INTEGER :: DIM
WRITE(*,*) " Ge ordning "
READ(*,*) DIM
ALLOCATE(X(DIM))
ALLOCATE(Y(DIM))
X = 2.0         ! Detta är en vektoroperation.
Y = POTENS(X)
WRITE(*,*) Y
STOP
END PROGRAM NY_TEST_FAELTVAERD_FUNKTION

FUNCTION POTENS(X) RESULT(FAELT)
IMPLICIT NONE
REAL, DIMENSION(:) :: X
REAL, DIMENSION(SIZE(X)) :: FAELT
INTEGER :: I
DO I = 1, SIZE(X)
    FAELT(I) = X(I)**I
END DO
RETURN
END FUNCTION POTENS
```

Avsikten hos olika funktionsargument

I variabeldeklarationer till subprogram har för argumenten tillkommet möjligheten att ange avsikten hos en variabel, dvs om den är invariabel, utvariabel, eller båda samtidigt. Detta anges med `INTENT` som kan vara `IN, OUT` eller `INOUT`. Om `IN` gäller så kan det verkliga argumentet vid anropet vara ett uttryck som `X+Y` eller `SIN(X)` eller en konstant som 37, eftersom ett värde skall överföras till subprogrammet men ej åter till anropande enhet. Variabeln får i detta fall ej tilldelas något nytt värde i subprogrammet. Om `OUT` gäller så måste däremot det verkliga argumentet vara en variabel. Vid inträde i subprogrammet anses då variabeln som odefinierad. Det tredje fallet täcker båda möjligheterna, ett värde in, ett annat (eller eventuellt samma) ut. Även då måste naturligtvis det verkliga argumentet vara en variabel. Implementeringen av `INTENT` är dock ännu ej fullständig, dvs alla system gör ännu ej alla tester som är tänkbara. Sanno-

likheten för att tester sker ökar om man har med ett fullständigt `INTERFACE` i den anropande programenheten, som i nedanstående exempel. Ett speciellt avsnitt om avsikt eller `INTENT` finns i avsnitt 10.1, sid 96. Om argumentet har ett pekar-attribut `POINTER` så får `INTENT` ej sättas.

```
        PROGRAM TEST_AV_AVSIKT
        IMPLICIT NONE
        INTERFACE
             FUNCTION FUN(X,Y,Z) RESULT(UT)
             REAL :: UT
             REAL, INTENT(IN)    :: X
             REAL, INTENT(OUT)   :: Y
             REAL, INTENT(INOUT) :: Z
             END FUNCTION FUN
        END INTERFACE
        REAL :: Y, Z
        Z = 12.0
        WRITE(*,*) Z
        Y = FUN(1.0,Y,Z)
        WRITE(*,*) Y, Z
        STOP
        END PROGRAM TEST_AV_AVSIKT

        FUNCTION FUN(X,Y,Z) RESULT (UT)
! Varning! Denna funktion har sido-effekter, vilket är
! olämpligt. Sidoeffekterna är här för att illustrera
! avsikt eller INTENT.
           IMPLICIT NONE
           REAL, INTENT(IN)    :: X
           REAL, INTENT(OUT)   :: Y
           REAL, INTENT(INOUT) :: Z
           REAL :: UT
           Y = SIN(X) + COS(Z)
           Z = SIN(X) - COS(Z)
           UT = Y + Z
           RETURN
           END FUNCTION FUN
```

Om man ovan exempelvis byter ut `Y = FUN(1.0,Y,Z)` mot anropet `Y = FUN(1.0,Y,3.0)` klagar NAG:s Fortran 90 kompilator på att det ej getts något ut-argument på plats 3.

Frivilliga argument och argument utnyttjande nyckelord

Funktioner kan anropas med nyckelordsargument och de kan utnyttja frivilliga eller underförstådda argument, dvs en del argument kan ges med nyckelord i stället för med position, och en del behöver ej ges alls utan utnyttjar i stället ett skönsvärde (eng. default value).

Användningen av nyckelord och underförstådda argument är ett av de fall då ett explicit gränssnitt `INTERFACE` erfordras. Jag ger därför här ett fullständigt exempel. Som nyckelord användes de formella parametrarna i gränssnittet, vilka

ej behöver ha samma namn som de i den verkliga subrutinen. Dessa nyckelord skall ej deklareras i anropande programenhet.

```
IMPLICIT NONE
INTERFACE
   FUNCTION SUMMA (A, B, N) RESULT (UT)
      REAL                         :: UT
      INTEGER, INTENT (IN)         :: N
      REAL, INTENT(OUT)            :: A
      REAL, INTENT(IN), OPTIONAL :: B
   END FUNCTION SUMMA
END INTERFACE

REAL :: X, Y
Y = SUMMA(X,100.0,20)       ! Normalt anrop
WRITE(*,*) X, Y
Y = SUMMA(B=10.0,N=50,A=X)  ! Anrop med nyckelord
WRITE(*,*) X, Y
Y = SUMMA(B=10.0,A=X,N=100) ! Anrop med nyckelord
WRITE(*,*) X, Y
Y = SUMMA(N=100,A=X)        ! Anrop med nyckelord
                            ! och ett skönsvärde.
WRITE(*,*) X, Y
END

FUNCTION SUMMA(A,B,N) RESULT (UT)
IMPLICIT NONE
REAL :: A, UT
REAL, OPTIONAL, INTENT (IN) :: B
INTEGER :: N
REAL :: TEMP_B
IF (PRESENT(B)) THEN
        TEMP_B = B
ELSE
        TEMP_B = 20.0
END IF
A = TEMP_B + N
UT = A + TEMP_B + N
RETURN
END
```

Notera att `IMPLICIT NONE` för huvudprogrammet ej verkar i funktionen `SUMMA`, varför denna har kompletterats med det kommandot och deklaration av de ingående variablerna. Den inbyggda funktionen `PRESENT` är sann om dess argument finns med i anropet, och falsk om det inte finns med. I det senare fallet användes i stället skönsvärdet (eng. the default value).

Körning på UNIX sker med

```
f90 program.f90
a.out
   1.2000000E+02     2.4000000E+02
```

```
60.0000000     1.2000000E+02
  1.1000000E+02      2.2000000E+02
  1.2000000E+02      2.4000000E+02
```

Gränssnittet `INTERFACE` placeras lämpligen i en modul, så att användaren inte behöver bry sig. Gränssnitten blir ett naturligt komplement till rutinbiblioteken, Fortran 90 söker automatiskt efter moduler i aktuell filkatalog, eventuella filkataloger i I-listan, samt `/usr/local/lib/f90`. Begreppet I-lista förklaras i Bilaga 6. Om man glömmer `INTERFACE` eller har ett felaktigt sådant erhålles ofta felet "Segmentation error". Detta fel kan dock även erhållas vid ett i princip korrekt `INTERFACE`, men då man glömt att allokera något av de ingående fälten i den anropande programenheten.

Notera att om en utmatningsvariabel anges som `OPTIONAL` och `INTENT (OUT)` så måste den vara med i anropslistan om programmet vid exekveringen lägger ut ett värde på denna variabel. Man måste därför i sitt program använda test med `PRESENT` av aktuell variabel, och endast om den är med i anropet använda den för tilldelning, för att på så sätt få den önskade valfriheten om man bara ibland vill ha ut en viss variabel.

Generiska funktioner

Detta begrepp innefattar dels de inbyggda funktionerna, vilkas generiska egenskaper har diskuterats redan i avsnitt 4.1.1, sid 42, dels möjligheten att skriva egna generiska funktioner och subrutiner.

Grundprincipen för de inbyggda generiska funktionerna tål dock att upprepas, dvs att datatypen hos argumentet anger vilken rutin som skall väljas. Under Fortran 66 var man tvungen att skriva `SIN(X)` om `X` var `REAL, DSIN(X)` om `X` var `DOUBLE PRECISION` och slutligen `CSIN(X)` om `X` var `COMPLEX` för att få resultat av rätt typ och noggrannhet. Från Fortran 77 fungerar `SIN(X)` i alla tre fallen.

Jag ger här ett fullständigt exempel på en generisk subrutin, nämligen en rutin `SWAP(A,B)` som byter plats på värdena hos variablerna `A` och `B`, utnyttjande olika underliggande rutiner beroende på om de båda argumenten är av typ `REAL, INTEGER` eller `CHARACTER`.

Subrutinen finns således i detta fall i tre varianter, nämligen för flyttal, heltal och textsträng (med längden ett). De är skrivna på vanligt sätt och heter `SWAP_R, SWAP_I` respektive `SWAP_C`. Samtliga är dessutom med i huvudprogrammets `INTERFACE` med sina deklarationer, men detta gränssnitt har namnet `SWAP`. När man i huvudprogrammet vill utnyttja någon av subrutinerna använder man enbart namnet `SWAP` och argument av viss typ. Systemet väljer då bland de tre tillgängliga funktionerna ut den som svarar mot argumenten i anropet. Om dessa båda argument är av olika typ blir det ett fel.

Nedanstående fungerar utmärkt, men användaren är inte så glad åt att behöva släpa med all information om `SWAP` och dess olika varianter i sitt program. Lösningen är att flytta över allt som rör `SWAP` till en modul, vilken sedan kan användas i huvudprogrammet med satsen `USE` modulnamnet. Vi återkommer till detta i kapitel 13, sid 109.

```
PROGRAM SWAP_HUVUD
IMPLICIT NONE
INTEGER   :: I, J, K, L
REAL      :: A, B, X, Y
CHARACTER :: C, D, E, F
INTERFACE SWAP
        SUBROUTINE SWAP_R(A,B)
        REAL, INTENT (INOUT)      :: A, B
        END SUBROUTINE SWAP_R
        SUBROUTINE SWAP_I(A,B)
        INTEGER, INTENT (INOUT)   :: A, B
        END SUBROUTINE SWAP_I
        SUBROUTINE SWAP_C(A,B)
        CHARACTER, INTENT (INOUT) :: A, B
        END SUBROUTINE SWAP_C
END INTERFACE

I = 1   ; J = 2    ;   K = 100  ; L = 200
A = 7.1 ; B = 10.9 ;   X = 11.1 ; Y = 17.0
C = 'a' ; D = 'b'  ;   E = '1'  ; F = '"'

WRITE (*,*) I, J, K, L, A, B, X, Y, C, D, E, F
CALL SWAP(I,J) ; CALL SWAP(K,L)
CALL SWAP(A,B) ; CALL SWAP(X,Y)
CALL SWAP(C,D) ; CALL SWAP(E,F)
WRITE (*,*) I, J, K, L, A, B, X, Y, C, D, E, F
END

SUBROUTINE SWAP_R(A,B)
IMPLICIT NONE
REAL, INTENT (INOUT)          :: A, B
REAL                          :: TEMP
        TEMP = A ; A = B ; B = TEMP
END SUBROUTINE SWAP_R

SUBROUTINE SWAP_I(A,B)
IMPLICIT NONE
INTEGER, INTENT (INOUT)       :: A, B
INTEGER                       :: TEMP
        TEMP = A ; A = B ; B = TEMP
END SUBROUTINE SWAP_I

SUBROUTINE SWAP_C(A,B)
IMPLICIT NONE
CHARACTER, INTENT (INOUT)     :: A, B
CHARACTER                     :: TEMP
        TEMP = A ; A = B ; B = TEMP
END SUBROUTINE SWAP_C
```

4.2.2 Subrutiner

Vanliga enkla subrutiner diskuterades redan i avsnitt 2.15, sid 24. De mer komplicerade möjligheterna hos funktioner gäller även subrutiner, skillnaden är att subrutinnamnet inte återför något värde i sig. Vi skall här nämna de båda specialfallen subrutiner utan argument och rekursiva subrutiner, liksom avsikten hos olika argument, frivilliga argument och argument med nyckelord.

4.3 Lokala funktioner

Eftersom de externa funktionerna och subrutinerna automatiskt får globalt giltiga namn finns det behov av funktioner som bara är tillgängliga i den programenhet där de behövs. I Fortran 77 och tidigare tillämpades för detta de så kallade satsfunktionerna. Detta begrepp har generaliserats i Fortran 90 till interna funktioner och även interna subrutiner.

För de lokala funktionerna och subrutinerna gäller att de ej kan användas som argument vid anrop av externa funktioner eller subrutiner (eller andra lokala).

4.3.1 Satsfunktioner

Satsfunktioner kom mycket tidigt i Fortran, men är inte så väldigt populära hos programmerarna. De är mycket praktiska och lättanvända, satsfunktioner placeras i aktuell programenhet mellan deklarationerna och de exekverbara satserna, och ser ut som tilldelningssatsen inne i en extern funktion. Observera att trots att de måste kunna skrivas på en logisk rad så kan de vara ganska avancerade, eftersom de kan anropa såväl inbyggda funktioner som tidigare definierade satsfunktioner. Ett enkelt exempel följer.

```
        IMPLICIT NONE
        REAL :: X, Y, PI, COT, NCOT
! Efter deklarationer kan satsfunktioner placeras.
        Y(X) = X + 2.0*X - 3.0*X**2
        COT(X) = 1.0/TAN(X)
        NCOT(X) = COT(PI*X)/PI
! Nu följer de exekverbara satserna.
        PI = 3.141592654
        READ(*,*) X
        WRITE(*,*) X, Y(X), COT(X), NCOT(X)
        END
```

Här är satsen `PI = 3.14159264` den första exekverbara satsen, men den borde nog hellre ha placerats bland deklarationerna utnyttjande `PARAMETER` satsen eller attributet. Denna form ger dock en god information om att satsfunktionerna utnyttjar argument (här `X`) och konstanter (här `PI`) som tilldelas senare, men satsfunktioner som definierats tidigare! De kan naturligtvis även använda konstanter och variabler som initierats redan bland deklarationerna.

Fördelarna gentemot en extern funktion är dels att lokala variabler delas med den aktuella programenheten, dels att funktionsnamnet blir lokalt (vilket innebär minskad risk för namnkonflikt).

4.3.2 Interna funktioner och subrutiner

Interna funktioner och subrutiner är en modernisering av de gamla satsfunktionerna. I en intern funktion eller subrutin kan mer komplicerade satser skrivas, ej enbart i form av en logisk rad. De placeras i slutet av aktuell programenhet (dock ej funktion), omedelbart före `END` placeras satsen `CONTAINS` följd av interna funktioner och subrutiner. Liksom hos satsfunktioner kan lokala konstanter och variabler delas med programenheten.

Ett enkelt exempel följer, nämligen en omskrivning av satsfunktionerna ovan. Notera att de tre interna funktionerna `Y, COT` och `NCOT` nu ej får deklareras på rad 3, den med `REAL :: X, PI`.

```
PROGRAM TEST_AV_INTERN
        IMPLICIT NONE
        REAL :: X, PI
! Nu följer de exekverbara satserna.
        PI = 3.141592654
        READ(*,*) X
        WRITE(*,*) X, Y(X), COT(X), NCOT(X)
CONTAINS
! Här placeras interna funktioner och subrutiner.
        FUNCTION Y(X)
                REAL :: Y
                REAL, INTENT(IN) :: X
                Y = X + 2.0*X - 3.0*X**2
        END FUNCTION Y
        FUNCTION COT(X)
                REAL :: COT
                REAL, INTENT(IN) :: X
                COT = 1.0/TAN(X)
        END FUNCTION COT
        FUNCTION NCOT(X)
                REAL :: NCOT
                REAL, INTENT(IN) :: X
                NCOT = COT(PI*X)/PI
        END FUNCTION NCOT
END PROGRAM TEST_AV_INTERN
```

I verkligheten kan de olika interna funktionerna och subrutinerna vara mycket komplicerade enheter med alla satser i Fortran, de kan dock ej i sin tur innehålla interna funktioner eller subrutiner.

OBS! Var försiktig vid deklaration av interna funktioner och subrutiner. De deklareras i samband med att de införes, de skall ej ges någon "egendeklaration.

Ett eventuellt `IMPLICIT NONE` i den överordnade programenheten gäller automatiskt även alla interna funktioner och subrutiner, liksom naturligtvis för eventuella satsfunktioner.

Övningar

(4.6) Skriv en rutin för beräkning av integralen av en funktion. Använd nyckelordsargument och underförstådda argument så att

- Om ingen vänster ändpunkt `A` ges så användes värdet 0.
- Om ingen höger ändpunkt `B` ges så användes värdet 1.
- Om ingen tolerans `TOL` ges så användes värdet 0,001 för det absoluta felet.

(4.7) Skriv det gränssnitt (`INTERFACE`) som behövs i anropande rutin för att kunna använda ovanstående integrationsrutin.

Kapitel 5

Text (CHARACTER)

Inledning

I detta kapitel ges en kortfattad genomgång av de viktigaste kommandona vid texthantering.

5.1 Nutida texthantering

I och med Fortran 77 infördes en speciell datatyp `CHARACTER` för texthantering. Denna datatyp innehåller i sin enklaste form utrymme för en bokstav eller siffra eller annat tecken. Följande enkla program skriver således ut texten kul.

```
CHARACTER*1 A, B, C
A = 'k'
B = 'u'
C = 'l'
WRITE(*,*) A, B, C
END
```

Ovanstående utnyttjar Fortran 77 deklationen `CHARACTER*1 A` för att specificiera att variabeln `A` är en textsträngsvariabel med längden 1. I Fortran 95 är motsvarande deklaration i stället `CHARACTER(LEN=1) :: A`.

En intressant variant är att använda konkatenering (summation) av tecken, vilket sker med symbolen två snedstreck `//`. Följande modifierade program har därför samma resultat som det föregående.

```
CHARACTER(LEN=1) :: A, B, C
A = 'k'
B = 'u'
C = 'l'
WRITE(*,*) A // B // C
END
```

Konkateneringen blir mer värdefull då vi väljer att sätta det nya resultatet i en längre variabel, vilken jag kallar för textsträng. Andra svenska böcker föredrar ordet teckensträng.

```
CHARACTER(LEN=1) :: A, B, C
CHARACTER(LEN=3) :: D
A = 'k'
B = 'u'
C = 'l'
D = A // B // C
WRITE(*,*) D
END
```

Vi kan nu börja med en verklig textmanipulering, till exempel genom att i stället omforma textsträngen kul till ull. Vi utgår nu i stället från den direkta tilldelningen av tre bokstäver till textsträngsvariabeln `D`, och placerar den andra bokstaven först och därefter den tredje bokstaven på den andra platsen.

```
CHARACTER(LEN=3) :: D
D = 'kul'
D(1:1) = D(2:2)
D(2:2) = D(3:3)
WRITE(*,*) D
END
```

Vi kan även omforma textsträngen kul till ull på ett "tuffare" sätt, genom att kopiera de båda sista bokstäverna till de båda första positionerna. Detta förfarande är tillåtet och entydigt även i detta överlappande fall i Fortran 90, men ej i Fortran 77.

```
CHARACTER(LEN=3) :: D
D = 'kul'
D(1:2) = D(2:3)
WRITE(*,*) D
END
```

En annan nyhet är att en tom sträng nu är tillåten, dvs "ingenting" mellan de båda "blipparna" är tillåtet och tolkas som en tom textsträng, vilket kan vara bra vid iterativa processer.

5.2 Inbyggda funktioner

En viktig uppgift vid texthantering är sortering i bokstavsordning. Under Fortran kräves att siffrorna och bokstäverna ligger i ordning så att 1 är mindre än 3, `c` är mindre än `p`, och `D` är mindre än `Z`. Däremot är inget föreskrivet om relationen mellan siffror, gemener och versaler. Under den vanliga sorteringsordningen ASCII (American Standard Code for Information Interchange) är det ordningen siffror, versaler och gemener, medan under IBM:s system EBCDIC är det i stället gemener, versaler och siffror. Hos EBCDIC ligger specialtecknen först men i ASCII ligger specialtecken både först och sist samt på båda sidor om versalerna. Detta senare passar hyfsat men ej exakt med vårt utnyttjande av vissa specialtecken för de svenska bokstäverna.

Jämförelse sker utnyttjande de gamla symbolerna

```
.LT.    .LE.    .EQ.    .NE.    .GT.    .GE.
```

eller motsvarande nya och mer matematiska

```
<       <=      ==      /=      >       >=
```

Notera att eftersom likhetstecknet = användes som tilldelningstecken (motsvarande := i Pascal) så måste vi välja ett annat tecken för likhet, nämligen `.EQ.` eller = =. Kompilatorer är ofta skrivna så att de påstår att symbolen `.LT.` har använts på fel sätt, även om man i själva verket använt symbolen <.

Observera att vi även har tillgång till ett antal logiska operatorer, nämligen `.NOT. .AND. .OR.` samt de mer ovanliga `.EQV.` och `.NEQV.` De definieras i nedanstående tabell, men de har tyvärr ingen modernare beteckning än "punktnamnen".

```
   A            T     T     F     F
   B            T     F     T     F
.NOT. A         F     F     T     T
A .AND. B       T     F     F     F
A .OR. B        T     T     T     F
A .EQV. B       T     F     F     T
A .NEQV. B      F     T     T     F
```

I Fortran 77 infördes ett alternativ till den maskinberoende sorteringsordning som erhålles med jämförelser av typ

```
IF (CHAR1 .LT. CHAR2 ) THEN
```

nämligen de fyra logiska funktionerna `LGE, LGT, LLE` och `LLT`, vilka är baserade på sorteringsordning enligt ASCII och som argument har två textsträngar. Ovanstående jämförelse skrives då

```
IF ( LLT(CHAR1, CHAR2) ) THEN
```

varvid en maskinoberoende sortering erhålles.

Fyra andra funktioner i Fortran 77 ger det nummer som svarar mot ett visst tecken, och tvärtom. Funktionen `ICHAR(A)` ger numret för tecknet i textsträngsvariabeln `A`, medan `CHAR(I)` ger tecknet som svarar mot numret `I`, utgående från intern (maskinberoende) representation. De båda funktionerna `IACHAR(A)` och `ACHAR(I)` arbetar däremot med ASCII-koden.

I Fortran 90 tillkom ytterligare ett antal användbara funktioner för texthantering. Dessa finns, tillsammans med ovan nämnda, uppräknade i Bilaga D.4, sid 192, och D.5, sid 193. De båda `ADJUSTL` och `ADJUSTR` vänster- respektive högerjusterar en sträng (dvs flyttar eventuella inledande respektive avslutande blanka men behåller längden), medan `TRIM` kortar av strängen genom att ta bort avslutande blanka. Längden av en sträng erhålles med `LEN`, medan `LEN_TRIM` ger längden utan avslutande blanka. Observera att `LEN` kan tillämpas på en deklarerad sträng som ej har tilldelats ett värde, i det fallet blir naturligtvis `LEN_TRIM` noll. Funktionen `REPEAT` användes för att upprepa en sträng, medan funktionerna `INDEX, SCAN` och `VERIFY` ger positionen för ett visst element, se Bilaga D för detaljer.

5.3 Forntida texthantering

Som tidigare nämnts så infördes `CHARACTER` i och med Fortran 77. Även tidigare fanns behov av att skriva ut text som lagrats i variabler. Detta skedde då genom att man lagrade text i vanliga heltalsvariabler `INTEGER` (eller i flyttalsvariabler `REAL`). Ett exempel följer, men det kan ej köras med NAG:s Fortran 90 system, eftersom detta ej innehåller tilldelning av textsträngar med Hollerithkonstruktionen `nHtext`, där `n` är en siffra med antalet tecken (bokstäver) som ryms i variabeln. Denna möjlighet, som motsvara den nutida varianten med textsträng 'text', försvann med Fortran 77, men många implementationer innehåller den som en utvidgning. Exemplet fungerade därför med Sun:s Fortran 77 och gav utskriften `Karl Johan`.

```
            INTEGER A, B, C
            A = 4HKarl
            B = 4H Joh
            C = 4Han
            WRITE(*,10) A, B, C
10          FORMAT(3A4)
            END
```

Metoden var inte speciellt flyttbar (portabel) eftersom antalet bokstäver som ryms i ett ord varierar mellan olika datorsystem. Vid utmatningen ovan måste ett explicit `FORMAT` användas för att få textutmatning, annars presenteras innehållet i de tre heltalsvariablerna `A, B` och `C` som tre heltal, dvs om `WRITE(*,*)` användes i stället för `WRITE(*,10)`. Mer om Format följer i nästa kapitel.

Man kan tilldela nya värden till ovanstående variabler med satser som

```
      IF (X) 20, 30, 40
20    C = A
        GO TO 50
30    C = 4HBo
        GO TO 50
40    C = B
50    CONTINUE
```

Det är dock mycket väsentligt här att man bara gör text-tilldelningar mellan variabler av samma datatyp, annars sker en omvandling mellan olika numeriska typer, varvid texten blir helt oläslig. I nedanstående fall erhålles på Sun utmatningen `Karl JohnÂÜ@`, eftersom jag nu infört flyttalsvariabeln `D`, som sedan satts lika med heltalsvariabeln `C`. Satsen `D = C` kommer därför att medföra en automatisk konvertering av bitmönstret i `C`.

```
            INTEGER A, B, C
            REAL D
            A = 4HKarl
            B = 4H Joh
            C = 4Han
            D = C
            WRITE(*,10) A, B, D
10          FORMAT(3A4)
            END
```

Däremot fungerar det om samtliga variabler är flyttal av samma precision, men ej om jag blandar enkel och dubbel precision.

5.4 Några in- och utmatningsproblem

Om textformatet `An` inte stämmer med `CHARACTER(LEN=n)` kan litet förvillande resultat erhållas. Notera att `n` i `An` inte får vara en variabel, utan måste vara en eller flera siffror.

Inläsning av `CHARACTER(LEN=1)` med `A5`

```
ABCDE     blir      E   (dvs sista tecknet vinner)
```

Inläsning av `CHARACTER(LEN=5)` med `A5`

```
ABCDE     blir  ABCDE
A         blir      A
A C       blir  A C
```

Inläsning av `CHARACTER(LEN=5)` med `A1`

```
ABCDE     blir  A     (dvs första tecknet vinner)
```

5.5 Textsträngar med variabel längd

Ett speciellt tillägg till standarden finns för detta, se ISO/IEC 1539-2 [21].

Den fullständiga texten till standarden, liksom implementering i standard Fortran 90 och två exempel, finns tillgängliga elektroniskt, se webb-versionens Kapitel 6, Sektion 6.5.

Programmen i exemplen definierar ord såsom åtskilda av blank (mellanslag), komma, punkt, utropstecken, frågetecken eller postslut (radslut). De bör därför kunna fungera även på svenska texter.

Ett liknande ämne är funktioner med textsträng av antagen längd som resultat, men detta begrepp försvinner snart (föråldrat i Fortran 2003).

Kapitel 6

Avancerad in- och utmatning (FORMAT)

Inledning

En mycket använd egenskap hos Fortran är att man kan föreskriva väldigt väl hur utskriften skall se ut vad gäller nya rader och antalet blanka mellan de olika posterna. Det är dock inte så lätt att skriva programmet så att det gör vad man vill, ofta krävs litet av trial and error.

Alla variabler är lagrade i datorn med 1 och 0, men representerar heltal, flyttal, logiska variabler eller bokstäver och andra skrivtecken.

Vid in/utmatning måste därför information finnas om vilken variabeltyp det är frågan om!

1. Vid liststyrd in/utmatning avgör typerna i listan detta, perfekt för inmatning från terminal (särskilt från skärm). Det är denna typ som vi hitintills har utnyttjat.

2. Vid format-styrd in/utmatning får programmeraren full kontroll! Ger mycket snygg utmatning. Oftast besvärlig vid inmatning från skärm. Formatstyrd inmatning är dock bekväm vid `CHARACTER`, vid liststyrd inmatning måste nämligen textsträngen omges med apostrofer, det slipper man vid format-styrd inmatning.

3. Vid oformaterad in/utmatning används bit-mönstret direkt. Perfekt för mellanlagring, tar litet utrymme och litet CPU-resurser. Ger för människan oläslig utskrift.

Notera dock att det använda bit-mönstret vid oformaterad in/utmatning kan skilja sig mellan olika datorfabrikat, se respektive systeminformation, som oftast även innehåller rutiner för omvandling mellan olika system. Även det använda programspråket kan påverka hur lagringen sker, Fortran och C använder olika representationer.

6.1 Formatstyrd utmatning

Formatstyrd utmatning innebär att användaren explicit ger ett format för varje variabel. Det är mycket jobb och ganska bökigt att utnyttja `FORMAT`, men det ger stora möjligheter. Numera utnyttjas `FORMAT` nästan bara för utmatning, för inmatning från terminal är liststyrd inmatning perfekt, medan formatstyrd var perfekt vid inmatning med hålkort.

Följande värden på flyttalet `A`, dubbelprecisionstalet `D`, komplexa talet `C`, heltalet `I`, logiska variabeln `L` samt tecknet `X`,

```
A = 1.0        D = 2.D0       C = (3.0 , 4.0)
I = 25         L = .TRUE.     X = 'X'
```

ger med utskriftskommandot

```
      WRITE(*,30) A, D, C, I, L, X
   30 FORMAT('1 A = ',F8.3,' D = ',F20.16,/
     ,'  C = ',2F8.3,' I = ',I3,' L = ',L1,' X = ',A)
```

först ny sida och sedan

```
A =    1.000 D =   2.0000000000000000
C =    3.000   4.000 I =  25 L = T X = X
```

Här är en förklaring på sin plats. Utskriften av talen i listan sker utnyttjande formatet med nummer 30, vilket dels innehåller information om vilken text som skall skrivas mellan de olika talen, dels hur de olika talen skall skrivas ut. Texten mellan de olika talen, samt eventuellt inledande och avslutande text, skrivs direkt i formatet utnyttjande vanliga apostrofer '.

- Som nämnts i avsnitt 2.7, sid 11, om in- och utmatning, fungerar både vanliga apostrofer `'` och citat-tecken `"` i detta sammanhang. Om man vill skriva en apostrof inne i texten är det lämpligt att innesluta texten i citat-tecken, eller tvärtom.

- Eftersom citat-tecken `"` infördes i standarden först med Fortran 90 var man tidigare tvungen att skriva ut eventuell apostrof på annat sätt, nämligen genom att dubbelskriva den. Denna metod kan fortfarande användas, och även på citat-tecknet.

- Eftersom apostrof-tecken `'` infördes i standarden först med Fortran 77 var man tidigare tvungen att skriva ut text på annat sätt, nämligen genom att utnyttja Hollerith-konstruktionen `nHtext`, där talet `n` anger antalet tecken (bokstäver, blanka och andra skrivtecken) i texten. Denna konstruktion finns fortfarande kvar i Fortran 90, men överväges att tas bort i nästa revidering av standarden. Problemet med Hollerith-konstruktionen är att man måste räkna antalet tecken i texten, och resultatet måste vara rätt, annars blir det kompileringsfel eller konstiga utmatningsfel.

- Herman Hollerith (1860 - 1929) var uppfinnaren av hålkortet!

Hur de olika talen skall skrivas ut anges med formatbokstäver. Dessa bokstäver finns listade i Bilaga A.8, sid 161.

I ovanstående exempel skrivs det första talet `A` ut med formatet `F8.3`, vilket betyder ett flyttal med totalt åtta positioner, varav tre för decimalerna. Notera att utrymme behövs för inledande blank, tecken, heltalssiffra samt decimalkomma, varför `F7.3` är tänkbart, men ej `F5.3`. Det andra talet `D` skrives ut med formatet `F20.16`, vilket betyder ett flyttal med totalt 20 positioner, varav 16 för decimalerna. Notera att utrymme även här behövs för inledande blank, tecken, heltalssiffra samt decimalkomma. Det komplexa talet `C` skrivs ut som två tal, realdelen och imaginärdelen.

Heltalsutmatning betecknas med `I` och antalet tecken i fältet. Motsvarande gäller logiska variabler och textsträngsvariabler.

En mindre variant på samma utmatning är

```
   WRITE(*,40) A, D, C, I, L, X
40 FORMAT('1 A = ',E18.3,' D = ',D28.16,/
  ,'  C = ',2E18.3,' I = ',I1,' L = ',L1,' X = ',A4)
```

med resultatet

```
A =           0.100E+01 D =              0.2000000000000000D+01
C =           0.300E+01         0.400E+01 I =  * L = T X =    X
```

Här är utrymmet för litet för heltalet 25, vilket därför markeras med en stjärna. Notera vidare att vi nu använder formatbokstäverna `E` och `D` för att beteckna flyttal med exponent. Fortfarande så anger den sista siffran antalet decimaler, men nu måste vi notera att exponentdelen tar fyra positioner. Både `E` och `D` betyder 10 upphöjt till, `D` är bara en indikation på att dubbel precision använts.

Man kan byta ut `E` som markering för utmatning på exponentform mot `ES`, och får då en vetenskaplig (Scientific) form, med utmatning av en siffra skild från noll före decimaltecknet. Om man i stället byter ut `E` mot `EN` får man en teknisk (ENgineering) form med en till tre siffror före decimaltecknet, och en exponent jämnt delbar med tre. Om det utmatade värdet är noll erhålles samma utmatning från `ES` och `EN` som från `E`.

Vid användning av vanlig formatstyrd in/utmatning kan man under vissa operativsystem, förutom att ange en fullständig specifikation, alternativt ge bara respektive `FORMAT`-bokstav. Betydelsen, dvs antalet positioner i fältet, är då systemberoende.

6.2 Styrtecken

Första tecknet i varje post (rad) vid format-styrd utmatning ger ett kontrolltecken till radskrivaren, ibland kallade ASA-tecken efter en tidigare benämning på den amerikanska standardiseringskommissionen ANSI. Dessa har följande betydelse:

```
blank     Ny rad
+         Ej ny rad (överskrivning,
          finns ej på alla skrivare)
```

```
0         Dubbel radframmatning
1         Ny sida
```

På en del skrivare används ytterligare tecken, siffrorna är ofta vertikala tabulatorer.

OBS: Utmatning med `FORMAT(I5)` eller `F7.3` eller `1PE8.2` kunde medföra utmatning av en förfärlig massa papper med nästan inget tryck på, dvs "KARTONGBYTE". Ge därför alltid en explicit blank först i alla `FORMAT` avsedda för utmatning (om ej annat styrtecken önskas). Erhålles med `1X` eller `1H` eller `' '`. Använd därför aldrig samma `FORMAT` för både in- och utmatning.

Under UNIX är det litet bökigt att att utnyttja ovanstående styrtecken, det sker på olika sätt hos olika leverantörer. **Styrtecknen är nu på väg ut ur standarden, finns inte i Fortran 2003!** Av kompatibiltetsskäl negligeras därför det första tecknet, varför det fortfarande gäller att lämpligen ha en blank i position 1.

6.3 Liststyrd utmatning

Den liststyrda utmatningen är mycket bra vid utmatning av resultaten från enkla beräkningar, där layouten inte är så viktig. Man skriver bara

```
WRITE(*,*) utdata
```

där utdata dels är de variabler som skall matas ut, men även kan vara kompletterande text inom apostrofer eller citat-tecken. Se även en något utförligare diskussion i avsnitt 3.7 Enkel in- och utmatning samt nedanstående avsnitt om liststyrd inmatning.

Användning av Hollerith-konstruktionen vid liststyrd utmatning skall undvikas, den fungerar dock under en del Fortran 77 system.

```
WRITE(*,*) 32H Mycket olämpligt utmatningssätt
```

Följande tre utmatningssätt är tillåtna, men det första av dem är både jobbigt och föråldrat.

```
      WRITE(*,60)
60    FORMAT(25H Olämpligt utmatningssätt)

      WRITE(*,70)
70    FORMAT(' Lämpligt utmatningssätt')

      WRITE(*,*) ' Lämpligt utmatningssätt'
```

6.4 Oformaterad utmatning

Den oformaterade utmatningen är mycket bra vid utmatning av mellanresultat till en fil, för senare inmatning antingen från samma program, eller från ett annat.

Den innebär att in/utmatning sker i det format som variablerna lagras i datorn, dvs i princip som binära siffror och packade på ett sådant sätt att den

lagrade informationen är nästan oläslig på skärm eller papper. Denna lagringsmetod har **tre stora fördelar**:

- den är *snabb* eftersom ingen omformatering behöver ske
- den tar *minimalt med utrymme* eftersom ingen extra information behöver lagras
- *ingen noggrannhetsförlust*, eftersom ingen konvertering mellan binära och decimala flyttal behövs

Oformaterad lagring rekommenderas för all mellanlagring som avser samma maskintyp. Kommunikation mellan olika maskintyper kräver dock oftast att formaterad lagring används, ibland finns speciella konverteringsrutiner så att oformaterad överföring kan utnyttjas.

För att skriva ut oformaterat ger man först ett kommando för att öppna en fil för oformaterad utmatning, se vidare i nästa kapitel. Vi väljer att låta den oformaterade utmatningen ske på enhetsnummer 1 och på en ny fil med namnet utfil.

```
OPEN(1, FILE='utfil', STATUS='NEW', FORM = 'UNFORMATTED')
```

Själva utmatningen sker med en skrivsats som innehåller enhetsnumret inom parentesen, men inget ytterligare. Listan över de olika variablerna skrives på vanligt sätt.

```
WRITE(1) A, B, C, D
```

När man är färdig med utmatningen bör man stänga filen. Vissa system stänger automatiskt alla filer när programmet avslutas på normalt sätt, men ej alltid vid avbrott på grund av fel.

```
CLOSE(1)
```

Vid en eventuell inläsning av dessa data är det väsentligt att de variabler till vilka de inlästa talen tilldelas är av samma typ (och längd) som de skrivna. Man måste naturligtvis ha enheten öppnad innan man kan läsa från den.

```
READ(1) E, F, G, H
```

6.5 Formaterad utmatning av fält

En mycket viktig egenskap i Fortran är att in- och utmatning av hela fält kan ske mycket enkelt, utan att varje enskilt element behöver specificeras individuellt. I nedanstående exempel kan således hela vektorn `A` skrivas ut genom att enbart namnet `A` ges i en ut-lista. Alternativt kan ut-listan specificera de element som skall skrivas ut, antingen genom explicit uppräkning eller med en implicit `DO`-slinga. Slutligen så kan godtyckliga uttryck ges direkt i skrivsatsen.

Som framgår av nedanstående exempel kan man ange antalet element som skall matas ut med en faktor (konstant) framför aktuellt format.

```
     INTEGER :: I
     REAL, DIMENSION(10)   :: A

     WRITE(*,20) A       ! Utskrift av alla 10 elementen
20   FORMAT(10F10.3)
     WRITE(*,30) A(1), A(2), A(7) ! Utskrift av 3 element
30   FORMAT(3F10.3)

     WRITE(*,50) (A(I), I = 1,9,2)
50   FORMAT(5F10.3)
     WRITE(*,60) A(1)*A(2)+I, SQRT(A(3))
60   FORMAT(2F10.3)
```

Konstruktionen `(A(I), I = nr1, nr2, nr3)` är en implicit `DO`-slinga, och den fungerar nästan som en vanlig explicit `DO`-slinga

```
DO I = nr1, nr2, nr3
   WRITE(*,50) A(I)
END DO
```

men med den skillnaden att den explicita slingan skriver ut ett tal per rad, medan den implicita i detta fall skriver fem tal per rad. Skillnaden beror på att varje värde på styrvariabeln `I` ger upphov till ett nytt anrop av utskriftsrutinen, varvid formatet användes på nytt från början. Man kan säga att varje anrop av `WRITE` skriver ut minst en ny rad (post) och att på motsvarande sätt varje anrop av `READ` läser in minst en rad. Konstruktionen med en implicit `DO`-slinga kan bara användas i `WRITE` och `READ` satser.

En viktig egenskap hos format är att man kan ge för många"utan att det ställer till någon skada. I fallet ovan kan formaten med nummer 30, 50 och 60 samtliga ersättas med det längsta, nämligen det med nummer 20. Om antalet format-poster är för stort jämfört med antalet utmatningsposter så användes inte de överblivna, om det i stället är för litet så börjar utmatningen om från början (jämför dock nedan) av formatet. Exempel på möjliga formatkonstruktioner med motsvarande ut-lista är följande,

```
10F12.3                A
4(I5,F10.2)            (I(J),A(J), J = 1,4)
```

Den senare skriver ett heltal `I(1)`, ett flyttal `A(1)`, ett ytterligare heltal `I(2)`, ett flyttal `A(2)`, osv. Formaten kan kapslas

```
4(I5,3F10.2)           (I(J),A(J),B(J),C(J), J = 1, 4)
```

Om formaten tar slutbörjar det om från början (men på en ny rad)

```
(10I8)                 (I(J), J = 1, 1000)
```

eller från den vänsterparentes som svarar mot den näst sista högerparentesen (eller dess repetitionsfaktor, om sådan finns)

```
(2I5, 3(I2,2(I1,I3)), 2(2F8.2,I2))
                      ^         ^
                      ^         Näst sista högerparentes
                      Repetitionsfaktor till
                      denna parentes
```

I detta exempel matas således först 17 heltal ut, dvs (2+3(1+2*2)), därefter följer två uppsättningar med vardera två flyttal och ett heltal, därefter följer ny rad med två uppsättningar med vardera två flyttal och ett heltal, följt av ny rad med två uppsättningar med vardera två flyttal och ett heltal, osv, så länge som utdata räcker till.

Det kan även inträffa att man vill ge för fåtal att skrivas ut, som i nedanstående exempel.

```
      WRITE(*,10) A, B, C
10    FORMAT(' A = ', F6.3, ' B = ', F6.3, ' C = ', F6.3)
```

Detta ger en trevlig utmatning

```
A =  1.123 B = -2.345 C = 3.456
```

men om man bara har två tal att skriva ut erhålles den trista utskriften

```
A =  1.123 B = -2.345 C =
```

dvs `C =` står där i sin ensamhet. Om man i stället sätter in kolon efter varje post i formatet och således använder

```
10    FORMAT(' A = ', F6.3,:, ' B = ', F6.3,:, ' C = ', F6.3)
```

så blir utskriften den önskade

```
A =  1.123 B = -2.345
```

Användning av kolon i formatbeskrivningen innebär att systemet kollar om det finns något mer värde att skriva ut, om så är fallet fortsätter utmatningen som om ingenting hänt, om så inte är fallet avbrytes utmatningen. Flera kolon är således tillåtna, det är till och med möjligt att inleda med ett kolon för att alstra en blank rad, när utlistan inte innehåller något element.

6.6 Avancerad inmatning

6.6.1 Formatstyrd inmatning

Formatstyrd inmatning innebär att för varje element i in-listan användes ett specificerat format, t ex `FORMAT(F10.3, E16.6, I7, L3, A8)`. Detta innebär att i detta fall det första flyttalet skall finnas i inmatningsradens tio första positioner, det andra i de sexton följande positionerna, heltalet i de därpå följande sju positionerna, det logiska elementet i de följande tre, och slutligen texten i de följande åtta positionerna.

- Denna inmatningsmetod var den allmänna under hålkortstiden, men har nu bara ett historiskt intresse. Två problem bör dock noteras, nämligen att om ett datafält var felplacerat mellan de angivna fälten så blev resultatet också fel, men även om data var inom rätt fält kunde resultatet noll erhållas. Detta senare gällde vid format som `E16.6`, där det bara är de tolv sista positionerna som skall innehålla ett tal av formen `0.123456E+01`. Om man gett värdet `123` i de tre första positionerna, som enligt formatbeskrivningen skall vara blanka, blir inmatningsresultatet noll.
- Notera att även om man angett formatet som `E16.6` kan man låta inmatningsvärdet ges som fixpunkt, t ex 1.23, om det är tillräckligt långt till höger.
- Notera vidare att man vid fixpunktsformat som `F10.3` med decimal ej alltid behövde ge decimalpunkten explicit.

6.6.2 Liststyrd inmatning

Liststyrd in/utmatning innebär att ett standardformat utnyttjas för varje datatyp.

```
REAL A
DOUBLE PRECISION D
COMPLEX C
INTEGER I
LOGICAL L
CHARACTER*1 X

READ(5,*) A, D, C, I, L, X
```

på indata

```
1. 2. 3. 4 T 'Z'
```

ger med

```
WRITE(6,*) A, D, C, I, L, X
```

utskriften

```
1.000000, 2.0000000000000000
(3.000000, 0.000000E+00), 4, TZ
```

men samma lässats på indata

```
1. 2. (3., 4.) 5 .F. 'Y'
```

ger utskriften

```
1.000000, 2.0000000000000000
(3.000000, 4.000000), 5, FY
```

och på indata

```
1. 2. 3. 4. X 'Z'
```

erhålles

```
                          ^
?Illegal character in data
```

eftersom X inte är ett logiskt värde, och slutligen med

```
1. 2. 3. 4. T 'T'
```

erhålles

```
1.000000, 2.0000000000000000
(3.000000, 0.000000E+00), 4, TT
```

Regler för liststyrd inmatning.

1. Komplexa värden måste ges inom parentes, annars blir det ofta fel, till exempel får realdelen rätt värde, imaginärdelen blir noll och nästa storhet får kanske nästa värde i listan (det som eventuellt var tänkt för imaginärdelen).

2. Textsträngar (`CHARACTER`) måste inneslutas inom apostrofer.

3. Variabelvärden åtskiljes av blank, komma eller postslut (radslut).

4. Två komma direkt efter varandra innebär att motsvarande variabel överhoppas.

5. Samma värde kan tilldelas flera variabler i följd om det föregås av en multiplikator, t ex 15*7.0 för femton flyttals-sjuor, eller 13* för tretton överhoppade variabler.

6. Datorn väntar på samtliga värden i listan (vagnretur räcker ej för att få slut, i nödfall kan tecknet / tillgripas för att markera slut på inläsningen).

7. Vid liststyrd inmatning av oktala värden (`O`-format) skall det oktala värdet föregås av citattecknet ".

Vid liststyrd utmatning (fritt format, `FMT = *`) gäller standard-längder på fälten, i princip verkar det vara så att ett tillräckligt antal positioner utnyttjas. Man får ett komma och en blank mellan varje variabel, utom före en textsträngsvariabel (motsvarande `A`-format).

6.7 Diverse in- och utmatning

Bra tillägg till `READ` och `WRITE` är de speciella utgångarna vid fel eller filslut.

```
READ(5,...,ERR=snr1, END=snr2)
WRITE(5,...,ERR=snr3)
```

Vid paritetsfel och en del andra fel (implementationsberoende) flyttas exekveringen till sats snr1 respektive sats snr3.

På DEC-20 fungerade ERR även då typen på indata ej stämde med variabelns datatyp, den gav då en möjlighet att hoppa tillbaks för ett nytt försök. Ibland ger även filslut återhopp till snr1 (om END=snr2 ej getts).

Vid filslut gäller att END=snr2 ger att exekveringen flyttas till snr2. Ett filslut erhålles med kommandot Kontroll-z eller Kontroll-d. Bra för att avsluta en inläsning.

Numera anses dessa tillägg ej längre vara så lämpliga, eftersom de utnyttjar hopp till ett annat ställe i programmet, dvs de ger ofta en oöverskådlig (ostrukturerad) kod. En mer strukturerad ersättning har därför införts i form av en statusvariabel (av typ heltal) med namnet IOSTAT = HELTALSVARIABEL i läs- eller skrivsatsens parentes-del. Denna variabel är noll om inget fel inträffat, positiv vid fel, och negativ vid filslut.

Den gamle Fortran-programmeraren är van vid att numrera sina Formatsatser. Det ser emellertid inte så snyggt ut i ren Fortran 90, där satsnummer inte längre bör användes annat än i undantagsfall. Redan i Fortran 77 fanns, en dock sällan utnyttjad, möjlighet att i stället för ett numrerat format ha en formatvariabel, av typ CHARACTER, som sättes in direkt i respektive skrivsats.

Nedan visar jag tre olika sätt att ge denna tilldelning. De har alla sina för- och nackdelar.

```
PROGRAM FORMAT
IMPLICIT NONE
REAL                           :: X
CHARACTER (LEN=11)             :: FORM1
CHARACTER (LEN=*), PARAMETER :: FORM2 = "( F12.3,A )"
FORM1 = "( F12.3,A )"
x = 12.0
PRINT FORM1, X,  ' HELLO '
WRITE (*, FORM2) 2*X,  ' HEJ '
WRITE (*, "( F12.3,A )") 3*X,  ' HEJSAN '
END
```

I PRINT-satsen använder jag en textsträngsvariabel FORM1 med längden 11 som tilldelas sitt värde i en explicit tilldelningssats. Nackdelen med denna metod är främst att man måste manuellt räkna antalet tecken, om man anger ett för litet antal ger de flasta kompilatorer inget kompileringsfel, utan felavbrottet kommer först vid exekveringen.

I den första WRITE-satsen använder jag nu i stället en textsträngskonstant FORM2. Fördelen är att genom att jag använder PARAMETER ej behöver ge någon explicit längd på konstanten, utan kan ange LEN = *. Nackdelen är att en konstant ej kan ges ett nytt värde. I avsnitt 17.2, sid 146, ger jag ett mycket enkelt exempel på vad som kan kallas ett dynamiskt format.

I den andra WRITE-satsen använder jag däremot direkt en textsträng. Nackdelen är att strängen ej kan återanvändas.

Övningar

(6.1) Vad utför satsen `WRITE(*, "( HEJ )")` ?

(6.2) Vad utför följande satser?

```
CHARACTER (LEN=9) :: FILIP
FILIP = '(1PG14.6)'
WRITE(*,FILIP) 0.001, 1.0, 1000000.
```

Kapitel 7

Filer

Inledning

Filer är mycket användbara vid programmering. Förutom att de olika programmen oftast finns på filer är det vanligt med indata på fil och att utdata placeras på en fil, dels som lagring och dels inför utskrift på papper. En ytterligare viktig användning är för mellanlagring av resultat, som kanske inte i sin helhet kan rymmas i tillgängligt primärminne.

Vid utmatning på papper är det naturligtvis viktigt att det hela är läsbart, därför bör formaterad eller liststyrd utmatning användas. Vid mellanlagring bör man i stället se till effektiviteten och därför använda oformaterad utmatning, vilket dels drar litet datorresurser och dels ger en kompakt lagring. Vid behov av att läsa mellanresultat "med ögonen" kan man enkelt skriva ett program som läser den oformaterade filen och skriver den formaterat antingen direkt på skärmen eller på en annan fil för senare utmatning på papper.

7.1 Externa filer

Då man använder en extern fil måste man välja ett nummer (logisk enhet) mellan 0 och 99[1], men naturligtvis ej 5 eller 6, vilka är de vanliga enhetsnumren för in- respektive utmatning. Man skall under UNIX ej heller använda nummer 0, eftersom detta utnyttjas för *standard error*, dvs felutmatning.

För att kunna använda filer måste de först öppnas och sedan stängas. Flera Fortransystem (till exempel Sun och DEC Fortran 77 samt Cray under både Fortran 77 och 90) öppnar dock vid behov automatiskt filer som svarar mot Fortrans nummer, till exempel en fil `fort.7` vid utmatning på logisk enhet 7, samt stänger alla öppna filer vid normalt programslut. Andra system (till exempel NAG Fortran 90 på Sun och DEC) kräver att filen har öppnats explicit med kommandot `OPEN` innan den kan användas, de ger annars en felutskrift när filen behövs.

Om man skall skapa en fil för vanlig utmatning är det mycket enkelt, efter att ha valt ett enhetsnummer och ett filnamn använder man kommandot `OPEN` och får tillgång till att skriva på filen. När man är klar bör man stänga filen med kommandot `CLOSE`.

[1] Många implementationer klarar högre nummer än 99.

```
      NOUT = 20
      OPEN(NOUT,FILE='UTFIL.TXT')
      WRITE(NOUT,*) lista1
      WRITE(NOUT,10) lista2
10    FORMAT(     ) ! Här måste en formatspecifikation finnas.
      CLOSE(NOUT)
```

Man kan nu läsa filen. Även nu måste man först öppna den med kommandot `OPEN` och man bör stänga filen med kommandot `CLOSE`. Man behöver inte använda samma enhetsnummer, men man måste använda samma filnamn.

```
      NIN = 30
      OPEN(NIN,FILE='UTFIL.TXT')
      READ(NIN,*) lista1
      READ(NIN,10) lista2
10    FORMAT(     ) ! Här måste en formatspecifikation finnas.
                    ! Den måste passa med hur filen skrivits.
      CLOSE(NIN)
```

7.1.1 Vanliga filer

Vanliga filer är rent sekventiella filer, dvs man skriver post efter post, för att sedan spola tillbaksöch läsa post efter post. Möjligheterna att hoppa över en post, gå tillbaks en post eller att skriva över en post med nya värden är mycket begränsade. Man kan bara backa en post med kommandot `BACKSPACE` och backa till början av filen med `REWIND`. Båda dessa kommandon skall följas av respektive enhetsnummer. Man kan hoppa över poster vid inläsning med hjälp av `READ` satser där man inte tar hand om några inlästa värden.

Vanliga filer kan antingen vara formaterade eller oformaterade, normalt är formaterade.

Kommandot `OPEN`

När man skall skapa en fil väljer man ett enhetsnummer och ett filnamn, och med kommandot `OPEN` öppnar man filen. Detta kommando kan ges ett stort antal attribut `UNIT, FILE, STATUS, ACCESS, FORM, RECL, BLANK, ERR, IOSTAT, ACTION, POSITION, DELIM` och `PAD`.

- `UNIT` anger vilket enhetsnummer som skall användas. Kan ges som enbart talet (och då först bland attributen) eller med `UNIT=talet`.
- `FILE` anger filens namn. Kan ges som `FILE=VAR`, där textsträngsvariabeln `VAR` innehåller filnamnet, eller enklare som `FILE='filnamn'`.
- `STATUS` kan vara `'OLD', 'NEW', 'SCRATCH', 'REPLACE'` eller `'UNKNOWN'`, vilket anger att det är en gammal fil, en ny fil, en tillfällig fil, eller att typen är okänd, dvs oftast att det är en ny eller gammal fil. Attributet `'REPLACE'` innebär att om en fil med angivet namn ej finns skapas en sådan och ges attributet `'OLD'`. Attributet `'REPLACE'` innebär att om en fil med angivet namn finns så tas denna bort och en ny med angivet namn skapas och ges attributet `'OLD'`. Fallet `'UNKNOWN'` är installationsberoende, men normalt

har det samma effekt som `'REPLACE'`. Om något attribut ej ges blir det automatiskt `'UNKNOWN'`.

- `ACCESS = 'SEQUENTIAL'` anger att det är fråga om en vanlig sekventiell fil och `ACCESS = 'DIRECT'` anger att det är en direktaccess-fil, jfr sektion 7.1.2, sid 77. Om attributet ej ges blir det automatiskt en sekventiell fil.

- `FORM = 'FORMATTED'` anger att det är fråga om en formaterad fil medan däremot `FORM = 'UNFORMATTED'` anger att det är en oformaterad fil. Om attributet ej ges blir det automatiskt en formaterad fil vid sekventiell fil, men en oformaterad fil vid direktaccess-fil. Observera att list-styrda filer i detta sammanhang betraktas som formaterade.

- `RECL = längd` anger postlängden i direktaccess-filer, se sektion 7.1.2, sid 77.

- `BLANK = 'ZERO'` anger att blanktecken i tal som läses skall tolkas som nollor medan däremot `BLANK = 'NULL'` anger att blanktecken i tal som läses skall nonchaleras, enbart blanktecken tolkas dock som noll. Om attributet ej getts användes skönsvärdet `BLANK = 'NULL'`.

- `ERR = satsnr` anger var exekveringen skall fortsätta vid fel vid utförande av `OPEN`-satsen.

- `IOSTAT = variabel` får ett positivt värde om fel inträffar vid utförande av `OPEN`-satsen, och ett negativt värde om enbart ett filslut påträffas, och ett annat negativt värde om enbart ett postslut påträffas. Variabeln blir naturligtvis noll om inget fel inträffar.

- `ACTION = 'READ'` anger att filen endast får läsas, `'WRITE'` att den ej får läsas, medan däremot `'READWRITE'` anger att båda operationerna är tillåtna.

- `POSITION = 'ASIS'` anger att filen skall användas i sin nuvarande position, `'REWIND'` att den skall användas från början, medan däremot om `'APPEND'` anges att den skall användas från slutet (t ex skriva in mer information på slutet).

- `DELIM = 'APOSTROPHE'` anger att apostrofer ' användes för att avgränsa textsträngar vid list-styrd formatering eller vid utnyttjande av `NAMELIST`. Om däremot `'QUOTE'` angetts betyder det att citat-tecken " användes, i båda fallen dubbelskrives eventuellt tecken i textsträngen av respektive slag. Däremot anger standardfallet `'NONE'` att ingen avgränsning görs. Detta attribut kan endast användas vid formaterade filer, och nonchaleras vid normal formaterad inmatning.

- `PAD = 'YES'` anger att om vid formaterad inmatning en post innehåller för få element kompletteras med blanka, om däremot `'NO'` angetts betyder det att antalet element i inmatningslista och format-specifikationen måste överensstämma. Detta attribut kan endast användas vid formaterade filer och formaterad in- och utmatning. Om attributet ej getts gäller `'YES'`. OBS: För Fortran 77 gällde i princip däremot `PAD = 'NO'`.

Exempel på OPEN-sats:

```
NOUT = 20
OPEN(NOUT, FILE='UTFIL.TXT', STATUS='NEW', &
ACCESS='DIRECT', FORM='FORMATTED')
```

När man är klar med användningen av en fil bör man stänga filen med kommandot CLOSE. Detta kommando kan ges ett mindre antal attribut, nämligen UNIT, STATUS, ERR och IOSTAT.

STATUS kan här vara 'KEEP' eller 'DELETE', vilket anger att filen skall sparas eller tas bort vid stängningen. En fil som öppnats som 'SCRATCH' kan ej sparas.

Exempel på CLOSE-sats:

```
CLOSE(NOUT, STATUS = 'DELETE')
```

Kommandot INQUIRE för filer och filnamn

Ett ytterligare kommando i detta sammanhang är INQUIRE-satsen, vilken kan användas för att se om en viss fil finns, antingen via filnamnet eller via enhetsnumret.

```
INQUIRE(UNIT = nummer, lista)
```

eller

```
INQUIRE(FILE = filnamn, lista)
```

I lista anges exempelvis EXIST = logisk_variabel, varvid den logiska variabeln får värdet sant om respektive fil finns, annars får den värdet falsk. Här följer övriga möjliga parametrar:

- ERR = satsnr anger var exekveringen skall fortsätta vid fel vid utförande av INQUIRE-satsen.
- IOSTAT = variabel får ett positivt värde om fel inträffar vid utförande av INQUIRE-satsen, och ett negativt värde om enbart ett filslut påträffas, och ett annat negativt värde om enbart ett postslut påträffas. Variabeln blir naturligtvis noll om inget fel inträffar.
- OPENED = logisk_variabel, varvid den logiska variabeln får värdet sant om respektive fil eller enhet är öppnad, annars får den värdet falsk.
- NUMBER = heltals_variabel, varvid heltalsvariabeln får värdet av enhetsnumret, eller -1 om ingen enhet är kopplad till filnamnet.
- NAME = textsträngs_variabel, varvid variabeln tilldelas filnamnet, eventuellt kvalificerat (utvidgat). Det kan användas i en följande OPEN-sats.
- NAMED = logisk_variabel, varvid den logiska variabeln får värdet sant om respektive fil har ett namn, annars får den värdet falsk.
- ACCESS = textsträngs_variabel, varvid variabeln tilldelas något av värdena 'SEQUENTIAL', 'DIRECT' eller 'UNDEFINED', det senare om enheten ej är kopplad.

- SEQUENTIAL = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om filen kan öppnas för sekventiell access.
- DIRECT = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om filen kan öppnas för direkt-access.
- FORM = textsträngs_variabel, varvid variabeln tilldelas ett av värdena 'FORMATTED', 'UNFORMATTED' eller 'UNDEFINED' beroende på vilket slags fil som är kopplad, sista alternativet om ingen fil är kopplad.
- FORMATTED = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om filen kan öppnas för formaterad access.
- UNFORMATTED = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om filen kan öppnas för oformaterad access.
- RECL = heltals_variabel, varvid heltalsvariabeln får värdet av maximala postlängden tillåten för filen. Längden är antalet tecken vid enbart formaterad teststräng, och systemberoende i alla andra fall.
- NEXTREC = heltals_variabel, varvid heltalsvariabeln får värdet av senaste lästa eller skrivna posten plus 1.
- BLANK = textsträngs_variabel, varvid variabeln tilldelas värdet 'NULL', 'ZERO' eller 'UNKNOWN' beroende på om blanka tecken i numeriska fält skall nonchaleras, tolkas som noll, eller om det antingen saknas koppling eller om kopplingen inte är för formaterad in/ut-matning.
- POSITION = textsträngs_variabel, varvid variabeln kan tilldelas värdet 'REWIND', 'APPEND', 'ASIS' eller 'UNDEFINED' beroende på aktuell position (i första hand från motsvarande OPEN-sats). Om det saknas koppling, eller om kopplingen avser direkt-access, gäller fallet 'UNDEFINED'.
- ACTION = textsträngs_variabel, varvid variabeln tilldelas något av värdena 'READ', 'WRITE', 'READWRITE' eller 'UNDEFINED' beroende på aktuellt fall (i första hand från motsvarande OPEN-sats). Om det saknas koppling gäller fallet 'UNDEFINED'.
- READ = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om läsning är tillåten.
- WRITE = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om skrivning är tillåten.
- READWRITE = textsträngs_variabel, varvid variabeln tilldelas värdet 'YES', 'NO' eller 'UNKNOWN' beroende på om både läsning och skrivning är tillåten.
- DELIM = textsträngs_variabel, varvid variabeln tilldelas ett av värdena 'APOSTROPHE', 'QUOTE' eller 'NONE' efter vad som specificerats i motsvarande OPEN-sats. Om det saknas koppling, eller om den inte avser formaterad in/ut-matning, gäller i stället 'UNDEFINED'.

- `PAD = textsträngs_variabel`, varvid variabeln tilldelas värdet `'YES'` eller `'NO'` efter vad som specificerats i motsvarande `OPEN`-sats.

Kommandot `INQUIRE` för list-längd

En ytterligare variant av `INQUIRE`-satsen kan användas för att bestämma längden av utmatningslistan vid oformaterad utmatning.

```
INQUIRE(IOLENGTH = laengd) lista
```

Här är `lista` en utmatningslista, till exempel `A(1:N)`. Man får då i `laengd` reda på längden i systemberoende enheter för motsvarande oformaterade utmatning. Detta kan vara värdefullt i samband med postlängden `RECL` i direktaccessfiler.

7.1.2 Direktaccess-filer

Direktaccess-filer är utmärkta då man vill skriva poster i en viss ordning, och ändra vissa poster, och läsa enbart vissa poster. Möjligheterna att hoppa över en post, gå tillbaks en post eller att skriva över en post med nya värden är mycket stora.

Direktaccess-filer användes exempelvis på databaser, som ofta består av ett visst antal fixa poster. Ett bra exempel är en telefonkatalog med ett visst antal tecken för efternamn, förnamn, titel, adress, postnummer, adressort och telefonnummer. Numera är dock de flesta databaser relationella, men om man vill koppla en databas till ett Fortranprogram kan man behöva exportera databasen till en direktaccess-fil i Fortran.

Notera att både vanliga filer (sekventiella) och direktaccess-filer oftast lagras på skivminnen, vilka ofta kallas för direkt access minnen, men då användes direkt access i betydelsen on line.

Direktaccess-filer kan antingen vara formaterade eller oformaterade, normalt är oformaterade.

Vid användning av direktaccess-filer tillkommer några attribut till `READ` och `WRITE`-satserna, jämför avsnitt 6.7, sid 69.

- `UNIT` anger som vanligt vilket enhetsnummer som skall användas. Kan ges som enbart talet (och då först bland attributen) eller med `UNIT`-talet.
- `FMT = nummer` eller format får vara med, om det saknas är överföringen oformaterad. Om det är andra element kan det ges utan `FMT` = (om även enhetsnummret getts utan `UNIT` =). Vid direktaccess får `FMT = *`, dvs liststyrt format, ej användas.
- `END` får ej ingå. Direktaccess-filer har nämligen inget formellt filslut.
- `ERR` får ingå. De fel som avkännes är implementationsberoende.
- `REC = nummer` måste vara med, innehåller numret på den post man vill läsa eller skriva, kallas ibland postnummer".
- `IOSTAT = heltalsvariabel`. Denna variabel är noll om inget fel inträffat, positiv vid fel, och negativ vid filslut.

På direktaccess-filer får man inte använda något av `BACKSPACE, ENDFILE` eller `REWIND`. Däremot kan man naturligtvis simulera `BACKSPACE` med att minska postnumret med ett och `REWIND` med att sätta det till ett.

Jag använde direktaccess-filer mycket under min tid på Försvarets Forskningsanstalt. Under varje postnummer (`REC=nr`) lagrade jag rutnätet och alla tillhörande storheter vid en viss tidpunkt för lösningen av en hyperbolisk differentialekvation. Med ett program kunde jag sedan välja ut lämplig tidpunkt (postnummer) och plotta ut önskade storheter vid denna tidpunkt. Med att annat program kunde jag plotta valda storheter som funktion av tiden. Direktaccessfilen kunde även användas för omstart (fortsättning av föregående körning).

Jag ger här ett mycket enkelt exempel på användning av direktaccess-filer, nämligen en databas med telefonnummer över personalen vid Nationellt Superdatorcentrum för många år sedan (samtliga har nu slutat).

Det första programmet skapar databasen, vilken jag valt vara formaterad och med längden 120 tecken för varje post. Inmatningen av data har jag gjort på enklast möjliga sätt, varefter jag skriver ut den i omvänd ordning. De första 50 positionerna i fältet på direktaccess-filen placeras högerjusterade i `X`, varför jag vänsterjusterar med `ADJUSTL` före utmatningen.

```
      PROGRAM DIREKT_ACCESS
      IMPLICIT NONE
      INTEGER, PARAMETER :: NDIR = 20
      INTEGER            :: NR
      CHARACTER*50 :: X
      OPEN(NDIR, FILE='direkt.acc', STATUS='UNKNOWN', &
           ACCESS='DIRECT', FORM='FORMATTED', RECL=120)
      WRITE(NDIR,10,REC=1) "Bo Einarsson;013-281432"
      WRITE(NDIR,10,REC=2) "Bo Sjögren;013-282625"
      WRITE(NDIR,10,REC=3) "Mats S Andersson;013-282568"
      WRITE(NDIR,10,REC=4) "Ulla Bjuhr-Jönsson;013-282618"
      WRITE(NDIR,10,REC=5) "Larsgunnar Nilsson;013-281107"
      DO NR = 5, 1, -1
         READ(NDIR,10,REC=NR) X
         WRITE(*,*) ADJUSTL(X)
      END DO
      CLOSE(NDIR)
      STOP
 10   FORMAT(A50)
      END PROGRAM DIREKT_ACCESS
```

Det andra programmet använder databasen, man ger önskat postnummer och får ut önskad individ. Notera att man inte behöver läsa in hela databasen, utan bara den post som man är intresserad av.

```
      PROGRAM LAES_DIREKT_ACCESS
      IMPLICIT NONE
      INTEGER, PARAMETER :: NDIR = 20
      INTEGER            :: NR
      CHARACTER*50 :: X
      OPEN(NDIR, FILE='direkt.acc', STATUS='OLD', &
```

```
         ACCESS='DIRECT', FORM='FORMATTED', RECL=120)
      DO
          WRITE(*,'(A)',ADVANCE='NO') ' Ge nr = '
          READ(*,*) NR
          IF (NR < 1 ) EXIT
          READ(NDIR,10,REC=NR) X
          WRITE(*,*) ADJUSTL(X)
      END DO
      CLOSE(NDIR)
      STOP
 10   FORMAT(A50)
      END PROGRAM LAES_DIREKT_ACCESS
```

Begreppet ADVANCE behandlas i kapitel 14.4, sid 120.

7.2 Interna filer

Interna filer kan användas i de fall man först vill läsa en post med ett format och sedan läsa den (fortfarande samma post) med ett annat format. Vad man gör då är att skriva på en fil i primärminnet (intern fil) i stället för en fil på sekundärminnet (skivminnet, extern fil).

Ett exempel ges nedan på detta (något krångliga) förfarande, kört både på den gamla DEC-20 under operativsystemet TOPS-20 och på en modern UNIX-maskin. Vi skapar här den interna filen genom att deklarera KORT som en textsträng med 80 tecken. Vanlig inläsning utnyttjande textformat sker till denna interna fil i sats nummer 15 nedan, vanlig utskrift av den sker i sats 22, medan den interna filen återanvändes för inläsning i satserna 28 och 50, utnyttjande heltals respektive flyttalsformat.

```
      PROGRAM INTERN
C     DETTA PROGRAM DEMONSTRERAR ANVÄNDNINGEN AV INTERNA
C     FILER. NOTERA ATT IBLAND BETRAKTAS BLANKA SOM NOLLOR,
C     DVS PÅ TOPS-20 ANVÄNDES BZ. VID VANLIG
C     HELTALSINMATNING UNDER TOPS-20 GÄLLDE DÄREMOT BN.
C     UNDER UNIX GÄLLER BN.
      CHARACTER KORT*80
      INTEGER HELTAL
      REAL FLYTAL
10    WRITE(6,11)
11    FORMAT(' IN: ')
15    READ(5,20,END=90) KORT
20    FORMAT(A80)
22    WRITE(6,25) KORT
25    FORMAT(' UT: ',A80)
28    READ(KORT,30,ERR=50) HELTAL
30    FORMAT(I5)
      WRITE(6,40) HELTAL
40    FORMAT(' HELTAL = ',I5)
      GOTO 10
50    READ(KORT,60,ERR=80) FLYTAL
```

```
60    FORMAT(F20.5)
      WRITE(6,70) FLYTAL
70    FORMAT(' FLYTTAL = ',F15.8)
      GOTO 10
80    CONTINUE
      GOTO 10
90    STOP
      END
```

Programmet ovan läser in en post (rad, hålkort med 80 tecken) och skriver ut den oförändrad. Därefter kollar programmet om det står ett heltal i de första fem positionerna, i så fall skrivs heltalet ut och programmet begär ny post. Om programmet ej funnit heltal kollar det i stället om det är ett flyttal i de första 20 positionerna, skriver ut det och begär nästa post. Man slutar genom att ge slutttecknet Kontroll-z.

Ovanstående princip kan användas om man inte vet exakt vilken typ av data som kommer. Körning av detta program gav på DEC-20 under TOPS-20 respektive på Sun SPARC under Sun OS 4.1.2.

```
DEC                                 Sun (vid skillnad)
@EX INTERN                          f77 intern.f
FORTRAN: INTERN                     intern.f
INTERN                              MAIN intern:
LINK:     Loading                   a.out
LNKXCT    INTERN execution
IN:
ABCDEFGHIJKLMNOPQRSTUVWXYZÅÄÖÉÜ
UT: ABCDEFGHIJKLMNOPQRSTUVWXYZÅÄÖÉÜ

IN:
12345
UT: 12345
HELTAL = 12345
IN:
1
UT: 1
HELTAL = 10000                      HELTAL =     1
IN:
1 1 1
UT: 1 1 1
HELTAL = 10101                      HELTAL =   111
IN:
12.56
UT: 12.56
FLYTTAL =     12.55999994           FLYTTAL =     12.56000042
IN:
9.1
UT: 9.1
FLYTTAL =      9.10000002           FLYTTAL =      9.10000038
IN:
```

```
  9.1
UT:   9.1
FLYTTAL =      9.10000002       FLYTTAL =      9.10000038
IN:
kontroll-Z
                                Suspended
```

Övning:
Kör detta program på ditt system och se vad som händer!

Kapitel 8

Formen hos ett Fortran-program

Inledning

Ett Fortran-program består av en eller flera programenheter, däribland exakt ett huvudprogram. De olika programenheterna diskuteras närmare i kapitel 13.

Tidigare skrevs alla Fortran-program i ett hålkortsbaserat system där kolumnerna var mycket viktiga, och den verkliga programtexten (källkoden) strängt taget bara fick finnas i kolumnerna 7 till 72. En ny form för källkoden har därför skapats.

8.1 Fix form

I det gamla systemet är kolumnerna 1 till 5 reserverade för eventuellt satsnummer, kolumn 6 är reserverad för eventuellt fortsättningstecken, kolumnerna 7 till 72 användes för den verkliga programtexten, och kolumnerna 73 till 80 är reserverade för kommentarer (hålkortsnummer). Under vissa system har en utvidgning av fältet för den verkliga programtexten skett, till bland annat 80 och 120. Denna gamla form kallas numera fix form, och normalt låter man namnet på motsvarande källkodsfil heta `filnamn.f` eller `FILNAMN.FOR`.

Ett vanligt fel är att man låter den verkliga programtexten börja för tidigt (före kolumn 7), vilket ger att kompilatorn oftast noterar ett fel, eller låter den fortsätta för långt åt höger (efter kolumn 72), vilket kompilatorn oftast ej noterar som fel.

8.2 Fri form

I det nya systemet är kolumnerna ej reserverade. Denna nya form kallas fri form, och normalt låter man namnet på motsvarande källkodsfil heta `filnamn.f90` eller `FILNAMN.F90`. En rad under fri form får innehålla upp till 132 tecken (inklusive eventuella blanka).

En avgörande skillnad är att under fix form är blanka ej signifikanta (utom i textsträngar), men i fri form är de det.

8.3 Fortsättningsrader

Ibland behövs mer än en rad för en sats. Detta löses olika i fix form respektive i fri form. Vi börjar med fix form, där ett godtyckligt tecken i kolumn 6 (dock ej blank eller noll) markerar att den raden är en fortsättning av den föregående.

```
     PRINT *, ' Detta är en lång utmatningsrads första del',
    *' och detta är dess andra del!'
```

Under fix form tillåtes maximalt 19 fortsättningsrader. Under fri form markerar man på den rad som skall fortsättas att en fortsättningsrad följer, i stället för i början av "nästa rad" som i fix form.

```
PRINT *, ' Detta är en lång utmatningsrads första del', &
' och detta är dess andra del!'
```

dvs man fortsätter numera en rad med att ge ett "ampersand" eller "och son", dvs tecknet &, på slutet av den gamla raden. Under fri form tillåtes maximalt 39 fortsättningsrader.

Ibland kan det hända att en viss identifierare eller ett visst numeriskt uttryck inte får plats på raden. Man kan då under fri form avbryta var som helst med tecknet & och sedan på nästa rad ge ett nytt & som första icke-blanka tecken. Man fortsätter sedan direkt från detta & utan blank. Tecknet & fungerar således som ett slags avstavningstecken.

```
PI = 3.141592653589793
```

kan således skrivas helt ekvivalent

```
PI = 3.14159265&
       &3589793
```

Notera även att det är det sista & på en rad som fungerar som fortsättningstecken, det går därför alldeles utmärkt att skriva en textsträng innehållande &.

Det kan ibland vara önskvärt att göra tvärtom, nämligen att ha flera satser på samma rad. Detta sker med hjälp av semikolon, samt med utropstecken före eventuell avslutande kommentar på raden. Det var däremot ej möjligt att ha flera logiska satser på samma rad under Fortran 77 och tidigare.

```
A = 0.0 ; B = 1.0 ; C = 2.0      ! Initiering
```

Ovanstående gäller den nya fria formen. Även i den gamla kolumnorienterade formen får man numera använda semikolon och utropstecken mellan kolumnerna 7 och 72, men man kan där ej fortsätta med & och ej skriva kommandon i kolumn 1 till 6 eller 73 till 80.

Notera att under den nya fria formen så är blanka signifikanta, dvs mitt favoritexempel

```
DO 25 I = 1. 25
```

ger kompileringsfel eftersom kompilatorn inte hittar något komma mellan undre och övre gräns, medan den komprimerade versionen

```
DO25I = 1.25
```

ger samma resultat som den okomprimerade under Fortran 77 eller under den gamla fixa formen av Fortran 90, nämligen att variabeln `DO25I` tilldelas värdet 1,25.

8.4 Kommentarer

Under fix form inledes kommentarer med ! eller `C` eller * i kolumn 1. Från Fortran 77 är dessutom en helt blank rad att betrakta som en (tom) kommentar, den gav tidigare kompileringsfel. Notera att ! kom först med Fortran 90.

Under fri form inledes kommentarer med ! var som helst på raden och avslutas med radslut. Notera att `C` eller * ej är tillåtna som kommentartecken under den fria formen.

Observera att kommentarrader ej kan fortsättas eftersom även tecknet & där behandlas som tillhörigt kommentaren. Däremot kan man fritt lägga in kommentarrader även bland fortsättningsrader. Man kan naturligtvis "fortsätta" kommentarrader genom att ge ett ! i kolumn 1 på nästa rad, men då blir det formellt en ny kommentar.

Ovanstående gäller den nya fria formen. Utropstecken i någon av kolumnerna 1 till 5 eller 7 till 72 betyder naturligtvis kommentar även i den gamla fixa formen, men i kolumn 6 betyder både ! och ; som tidigare fortsättningsrad. Notera att under Fortran 77 är däremot utropstecken inte tillåtet i kolumn 2 till 5 (om det inte finns en kommentarsymbol redan i kolumn 1).

Kommentarer är ett inkompatibilitetsproblem mellan Fortran 77 och fri form Fortran 90, men ej mellan fix och fri form Fortran 90, eftersom utropstecknet ! är tillåtet för att inleda en kommentar i båda. Lyckligtvis var det även tillåtet i såväl Sun Fortran 77 och DEC Fortran 77 (både DEC Station Ultrix och VAX/VMS) som i CRAY CF77. Man kan numera vara ganska säker på att ! i kolumn 1 fungerar som kommentartecken även vid användning av Fortran 77.

Övningar

(8.1) Vad betyder följande rad?

```
A = 0.0 ; B = 3.0 ! First variables ; C = 17.0 ; D = 33.0
```

(8.2) Är följande rader korrekt Fortran 90?

```
Y = SIN(MAX(X1,X2)) * EXP( -COS(X3)**I ) - TAN(AT&
        & AN(X4))
```

8.5 Gemensam form

Ibland kan det vara önskvärt att ha en programenhet i Fortran skriven på ett sådant sätt att den kan användas både under Fortran 77, Fortran 90 fix form och Fortran 90 fri form. En förutsättning för att samma källkod med säkerhet skall kunna utnyttjas är att programenheten verkligen följer standarden. Programenheten kan användas i alla tre fallen om man dels använder det nya

fortsättningstecknet `&` sist på den gamla raden (men i position 73 eller senare så att det ej stör i Fortran 77) och dels väljer som det nästan godtyckliga tecknet i kolumn 6 just `&` för att få fortsättning enligt Fortran 77. Ett inledande & nonchaleras "i princip" av Fortran 90.

```
      program TEST                          ! kolumn 73
!                                                |
      write(*,*)                                 &
     &     ' test '
      end
!     |
! kolumn 6
```

Detta är ej strikt standard Fortran 77 eftersom varken `&` eller `!` finns i den standardiserade tecken-uppsättningen. Däremot passar ovanstående konstruktion perfekt för programsegment som med satsen `INCLUDE` (se närmare Bilaga B.1, sid 164) ibland skall inkluderas i Fortran 90 programenheter under den gamla fixa formen, ibland under den nya fria formen. Programsegmentet skall då skrivas som om blanka vore signifikanta.

Kapitel 9

Ytterligare datatyper

Inledning

I Fortran 77 finns de båda datatyperna `COMPLEX` för komplexa tal och `DOUBLE PRECISION` för dubbel precision hos vanliga flyttal. Dessa båda datatyper saknas i de flesta andra programspråk, men de finns naturligtvis kvar i Fortran 90. I en del implementationer av Fortran 77 finns även kombinationen komplex dubbelprecision, men denna har aldrig standardiserats.

9.1 Komplexa tal (`COMPLEX`)

Ett komplext tal karakteriseras av att det består av en realdel och en imaginärdel. I Fortran begränsar man sig till det fallet att dessa båda delar är flyttal (dvs ej heltal). Man deklarerar motsvarande variabler med

```
COMPLEX :: A, B, C
```

Vid in- och utmatning betraktas komplexa tal som två reella tal, varför eventuell format specifikation måste ges för två tal. Vid liststyrd inmatning skall de båda delarna ges med parentes omkring samt med komma och/eller blank emellan.

Ett enkelt exempel på ett program med komplexa tal följer här.

```
      PROGRAM KOMPLEX
      REAL :: A, B, D
      COMPLEX :: C, C1, C2
      A = 3.0
      B = 5.0
      C1 = CMPLX(A,B)         ! Bilda ett komplext tal
      C2 = (4.0, 8.0)         ! Bilda ett komplext tal
      WRITE(*,*) ' C1 = ', C1
      WRITE(*,10) C2
10    FORMAT('  C2 = ',F10.3,E12.3)
      A = REAL(C1)        ! Ta realdelen
      B = AIMAG(C1)       ! Ta imaginärdelen
      C = CONJG(C1)       ! Komplexkonjugera
```

```
      D = ABS(C1)         ! Ta beloppet
      WRITE(*,20) A, B, C, D
 20   FORMAT('  Realdelen av C1 = ', F10.3,/, &
             '  Imaginärdelen av C1 = ', F12.3,/, &
             '  Konjugerade C1 = ', 2F10.3,/, &
             '  Absolutbeloppet av C1 = ', F10.3)
      WRITE(*,*) 'Ge nya värden på C1 i list-styrt format'
      READ(*,*) C1
      WRITE(*,*) 'Ge nya värden på C2 i formaterad form'
      READ(*,'(2F10.3)') C2
      WRITE(*,*) ' C1 = ', C1
      WRITE(*,10) C2
      END PROGRAM KOMPLEX
```

Körning av ovanstående program komplex.f90 på en Sun följer. Notera att vid list-styrd inmatning måste jag ge det komplexa talet inom parenteser, och vid formaterad inmatning måste jag placera de båda delarna i rätt kolumner (dvs 1-10 för realdelen och 11-20 för imaginärdelen), eftersom jag använt formatet 2F10.3.

```
f90 komplex.f90
a.out
  C1 =  (  3.0000000,  5.0000000)
  C2 =      4.000   0.800E+01
 Realdelen av C1 =  3.000
 Imaginärdelen av C1 =    5.000
 Konjugerade C1 = 3.000 -5.000
 Absolutbeloppet av C1 =  5.831
 Ge nya värden på C1 i list-styrt format
(12.0 , 25.0)                          Inmatade värden
 Ge nya värden på C2 i formaterad form
   7.6     8.3                         Inmatade värden
  C1 =  ( 12.0000000, 25.0000000)
  C2 =      7.600   0.830E+01
```

9.2 Dubbel precision (DOUBLE PRECISION)

På en normal dator finns numera oftast flyttalssystem enligt IEEE 754 [20], vilket innebär att enkel precision har cirka 7 decimala siffrors noggrannhet, men i dubbel precision cirka 16 decimala siffror. På en dator som Cray gäller i stället 13 siffror i enkel precision och 28 i dubbel precision.

Vid flera andra programspråk kan man välja om man vill arbeta i enkel eller dubbel precision för hela programmet, för Fortran gäller att man har full frihet att välja vilka satser som skall utföras i enkel precision och vilka som skall utföras i dubbel precision. Man måste härvid deklarera vilka variabler som skall lagras i dubbel precision med deklarationen

```
DOUBLE PRECISION :: lista
```

Vid användning av dubbel precision är det mycket viktigt att komma ihåg att låta alla aktuella variabler vara i dubbel precision, annars förlorar man

lätt noggrannhet, till exempel om man räknar ut skillnaden `D - E`, där `D` är en variabel i dubbel precision och `E` är en variabel i enkel precision, så blir resultatet av följande steg

```
DOUBLE PRECISION :: D
REAL :: E
...
E = 0.1
D = D - E
```

att eftersom `E` lagrar en tiondel binärt i enkel precision, dvs med ett avrundningsfel svarande mot enkel precision, att även `D` får ett avrundningsfel svarande mot enkel precision. Det rätta sättet att skriva är i detta fall, om man vill minska med en tiondel,

```
DOUBLE PRECISION :: D, E
...
E = 0.1D0
D = D - E
```

eller enklare

```
DOUBLE PRECISION :: D
...
D = D - 0.1D0
```

I den första alternativlösningen är det inte absolut nödvändigt att ha med `D0` i `0.1D0` för att instruera systemet att tiondelen skall beräknas i dubbel precision, eftersom en intelligent kompilator känner av detta, men det är säkrast. I den andra alternativlösningen är det mer nödvändigt eftersom det där är ett sammansatt uttryck. För sådana gäller att uttrycken automatiskt omvandlas till dubbel precision i enlighet med vissa specifika men krångliga regler, varför det finns risk att konverteringen till dubbel precision sker efter uträkningen av tiondelen i stället för vid uträkningen.

Ett enkelt exempel på ett meningslöst program utnyttjande dubbel precision följer.

```
PROGRAM DUBBEL
DOUBLE PRECISION :: D, E, F
REAL :: X, Y, Z
INTEGER :: K
...
Y = 5.2 - 0.7*F      ! Enkel precision
E = 5.2 - 0.7*F      ! Enkel precision
D = 5.2D0 - 0.7D0*F  ! Dubbel precision (säker)
X = D*F              ! Minskad precision
F = K - 20           ! Heltal till dubbel precision
E = D + DBLE(K-20)   ! Dubbel precision (säker)
IF ( F < 10.0D0 ) THEN
        F = F + 0.3D0
END IF
...
END PROGRAM DUBBEL
```

Raden `5.2 - 0.7*F` är faktiskt intressant. De båda enkelprecisionstalen 5.2 och 0.7 utvidgas till dubbel precision på grund av faktorn `F` i dubbel precision (men värdet av konstanterna blir eventuellt motsvarande enkel precision) och beräkningen av uttrycket sker i dubbel precision, varefter resultatet lagras i enkelprecisionsvariabeln Y respektive dubbelprecisionsvariabeln `E`.

Det finns således en viss frihet för kompilatorskrivaren att låta kompilatorn förbättra det program som användaren skrivet, genom att konvertera konstanter av typ decimaltal till dubbel precision i blandade uttryck. Om explicit mellanlagring, i en variabel deklarerad som enkel precision, skall ske är det dock förbjudet att efteråt använda en högre noggrannhet på motsvarande storhet.

Vid konvertering av ett program från enkel till dubbel precision kan det räcka med att glömma konvertera en enda variabel för att nästan hela förbättringen i noggrannhet skall gå förlorad.

9.3 Det nya precisionsbegreppet

Problemet med tidigare versioner av Fortran var att enkel precision på en dator kunde svara mot en högre noggrannhet än dubbel precision på en annan dator, och att den icke standardiserade datatypen `DOUBLE PRECISION COMPLEX` eller `COMPLEX*16` ej fanns på alla system. Denna senare datatyp finns det ett stort behov av på normala maskiner med enbart cirka 7 siffrors noggrannhet i enkel precision.

I Fortran 90 finns dels standardfunktioner för att undersöka vilken precision som gäller (se Bilaga D.8, sid 194, där exempelvis `PRECISION(X)` ger antalet signifikanta siffror hos tal av samma slag som variabeln `X`), och dels möjligheten att vid deklarationen av en variabel ange hur många signifikanta siffror som minst skall rymmas i flyttal av denna typ (eller detta slag, jämför nedan, avsnitt 9.4). De båda vanliga precisionerna enkel precision (`SP`) och dubbel precision (`DP`) på ett IEEE 754 [20] baserat system deklareras med

```
INTEGER, PARAMETER  :: SP = SELECTED_REAL_KIND(6,37)
INTEGER, PARAMETER  :: DP = SELECTED_REAL_KIND(15,307)
REAL(KIND=SP)       :: enkelprecisionsvariabler
REAL(KIND=DP)       :: dubbelprecisionsvariabler
```

Om man vill arbeta med exempelvis minst 14 decimala siffrors noggrannhet och minst en decimal exponent om + 300 så väljer man

```
INTEGER, PARAMETER  :: AP = SELECTED_REAL_KIND(14,300)
REAL(KIND=AP)       :: arbetsprecisionsvariabler
```

Tyvärr måste man nu ge alla flyttalskonstanter med tillägget `_AP`, till exempel

```
REAL(KIND=AP)     :: PI
PI = 3.141592653589793_AP
```

medan för de inbyggda funktionerna gäller att dessa är generiska, det vill säga de känner automatiskt av vilken datatyp (slag) som argument har och väljer då själva vilket slag som resultatet skall ha (oftast samma som argumentet).

Med denna metod får man i praktiken dubbel precision på datorer baserade på IEEE 754 [20], men enkel precision på datorer som Cray eller datorer baserade

på Digital Equipments Alpha-processor, dvs i samtliga fall en verklig precision om ungefär 15 signifikanta siffror.

Den icke-standardiserade datatypen `DOUBLE PRECISION COMPLEX` eller med ett annat skrivsätt `COMPLEX*16` erhålles helt enkelt genom att i stället skriva deklarationen som till exempel `COMPLEX(KIND=2)` eller `COMPLEX(KIND=16)`. Detta kan ske på de system som har detta slag definierat på ett lämpligt sätt. Se till exempel kapitel 14.9, sid 125, varav framgår att detta gäller en del vanliga system. I de fall man inte känner till slags-parameterns aktuella värden ger man i stället följande deklarationer och tilldelningar.

```
INTEGER, PARAMETER  :: AP = SELECTED_REAL_KIND(14,300)
COMPLEX(KIND=AP)    :: AA
...
AA = (0.1_AP, 0.2_AP)
```

9.4 Olika precision eller slag (KIND)

Eftersom det finns variabler av olika typer så finns det nu också en inbyggd funktion som känner av aktuell typ, nämligen funktionen `KIND(X)` som känner av typen hos `X`. Denna funktion användes även för deltyper eller slag".

```
KIND(0)        Heltal
KIND(0.0)      Flyttal
KIND(.FALSE.)  Logisk variabel
KIND("A")      Textsträng
```

Det finns en inbyggd funktion `SELECTED_REAL_KIND` som ger det slag av typen `REAL` vars representation har (åtminstone) en viss precision och ett visst exponentutrymme. Till exempel så ger funktionen `SELECTED_REAL_KIND(8,70)` det slag av `REAL` som har åtminstone 8 decimala siffrors noggrannhet och som tillåter belopp mellan 10-70 och 10+70.

För heltal heter den motsvarande funktionen `SELECTED_INT_KIND` och har naturligtvis bara ett argument. Med till exempel ett val `SELECTED_INT_KIND(5)` tillåtes alla heltal mellan -99 999 och +99 999. Slaget hos en typ kan tilldelas ett namn.

```
INTEGER, PARAMETER  :: K5 = SELECTED_INT_KIND(5)
```

Detta slag av heltal användes sedan i konstanter enligt

```
-12345_K5
+1_K5
2_K5
```

dvs på ett ganska onaturligt sätt, efter värdet följer ett understrykningstecken samt slagets namn. Däremot kan variabler av det nya heltals-slaget deklareras på ett trevligare sätt

```
INTEGER (KIND=K5)   :: IVAR
```

Motsvarande gäller för flyttalsvariabler, om vi först inför en hög precision `LONG` med

```
INTEGER, PARAMETER  :: LONG = SELECTED_REAL_KIND(15,99)
```

så får vi ett flyttals-slag med minst femton decimala siffrors noggrannhet och med minst exponent-området från 10^{-99} till 10^{+99}. Motsvarande konstanter erhålles med

```
2.6_LONG
12.34567890123456E30_LONG
```

och variablerna deklareras med

```
REAL (KIND=LONG)        :: LASSE
```

De gamla typomvandlingarna `INT, REAL` och `CMPLX` har utvidgats. Med funktionen

```
INT(X, KIND = K5)
```

omvandlas ett flyttal `X` till ett heltal av slaget `K5`, medan om `Z` är ett komplext flyttal så ger

```
REAL(Z, KIND(Z))
```

omvandling till ett reellt flyttal av samma slag som `Z` (dvs realdelen av `Z`).

Dubbel precision finns inte inbyggd i "nya" Fortran 90 på något annat sätt än i "gamla" Fortran 77, men det förutsättes att kompilatorn innehåller stöd för att utnyttja i hårdvaran eventuellt inbyggd dubbel eller fyrdubbel precision, så att man själv enkelt definierar ett lämpligt slag av `REAL` som `DP` eller `QP`. Man kan naturligtvis även använda det gamla begreppet `DOUBLE PRECISION`.

Anledningen till att man krånglat till det hela på detta sättet är dels att man inte vill ha alltför många obligatoriska precisioner (enkel, dubbel, fyrdubbel; eventuellt för de båda fallen `REAL` och `COMPLEX`), dels att det gamla begreppet `DOUBLE PRECISION` inte innebar någon viss specificerad maskinnoggrannhet. Nu kan man "enkeltspecificera både precisionen och exponentområdet. Ytterligare information om slag ges i Appendix 6 i internetversionen, där de olika datatypernas normala slag på både DEC (DEC Station Ultrix) och Sun samt Cray, liksom PC ges vid NAG:s system för Fortran 90. Se även kapitel 14.9, sid 125, här i boken.

Övningar

(9.1) Deklarera en flyttalstyp som svarar mot dubbel precision på IBM och enkel precision på Cray.

(9.2) Deklarera några variabler av ovanstående flyttalstyp.

(9.3) Deklarera några konstanter av ovanstående flyttalstyp.

9.5 Flera precisioner med kompileringskommandon

Mina tester tyder på att de flesta datorer följer IEEE 754 [20] korrekt (enkel och dubbel precision, högre precision är ännu ej standardiserad).

Avrundningskonstanten

Vi titter åter på beräkningen av avrundningskonstanten μ i kapitel 1, programmet `KAHAN`. Detta program, som är skrivet i normal enkel precision, finns på filen `kahan.f90`. Jag har nedan kört det med de tre olika möjliga precisionerna på en dator med en mycket enkel omställning av precisionen, först i enkel, sedan i dubbel och slutligen i fyrdubbel precision.

```
% f90 kahan.f90
% a.out
  my =   5.9604645E-08
% f90 -r8 kahan.f90
% a.out
  my =   1.110223024625157E-016
% f90 -r16 kahan.f90
% a.out
  my =   9.62964972193617926527988971292463 7E-0035
%
```

Hos flera leverantörer finns det några enkla kommandon för att informera kompilatorn (vid kompileringstillfället) om vilken precision som önskas. Detta sker på Sun med det krångliga kommandot `-xtypemap`, och följande kombinationer är tillåtna:

```
-xtypemap=real:32,double:64                    Default
-xtypemap=real:64,double:64                    Båda lika
-xtypemap=real:64,double:128                   Båda dubblade
```

Även för heltalen finns två alternativ, `integer:32` eller `integer:64`. Notera att endast variabler specificerade med `REAL :: VAR` påverkas, inte de specificerade med `REAL (KIND=ngt) :: VAR`, och med motsvarande regel för heltalen. Det är inte möjligt att använda `real:128` eller `double:32`. Man kan således inte på Sun "skrämma upp" precisionen från enkel till fyrdubbel.

9.6 Flera precisioner på en gång

Tre precisioner på en gång

Vi tittar på summation framlänges och baklänges, programmet `SUMMA`. Detta program finns på filen `summa.f90`.

Programmets matematiska/numeriska uppgift är att visa att summation med början på de minsta storheterna ger det mest noggranna resultatet. I ett svep klarar detta program alla tre möjliga precisionerna.

```
      PROGRAM SUMMA
!     Summation fram och baklänges av 1/n^2
!     Kompilerar inte om den begärda precisionen QUAD ej finns!
      IMPLICIT NONE
      INTEGER :: N, NMAX
      CHARACTER :: SVAR
      REAL :: SUM_SP, SLASK
      DOUBLE PRECISION :: SUM_DP
```

```
      INTEGER, PARAMETER :: QUAD = SELECTED_REAL_KIND(20,1000)
      REAL(KIND=QUAD) :: SUM_QUAD

      SUM_SP = 0.0
      SUM_DP = 0.0D0
      SUM_QUAD = 0.0_QUAD

 1    WRITE(*,'(A)', ADVANCE='NO') 'Hur många tal', &
       & ' vill Du summera? '
      READ(*,*) SLASK
      NMAX = NINT(SLASK)
      IF (NMAX .LT. 10) THEN
         WRITE(*,*) 'För få tal!'
         GO TO 1
      ELSE IF (NMAX .GT. 100000000) THEN
         WRITE(*,*) 'För många tal!'
         GO TO 1
      ENDIF
 2    WRITE(*,'(A)', ADVANCE='NO') &
     & 'Vill Du summera framlänges (F) eller baklänges (B)? '
      READ(*,*) SVAR
      SELECT CASE (SVAR)
        CASE('F','f')
          DO N = 1, NMAX
            SUM_SP = SUM_SP + 1.0/REAL(N)/REAL(N)
            SUM_DP = SUM_DP + 1.0D0/DBLE(N)/DBLE(N)
            SUM_QUAD = SUM_QUAD + &
                & 1.0_QUAD/REAL(N,KIND=QUAD)/REAL(N,KIND=QUAD)
          END DO
          WRITE(*,*) 'Summation framlänges av ',NMAX,' tal.'
        CASE('B','b')
          DO N = NMAX, 1, -1
            SUM_SP = SUM_SP + 1.0/REAL(N)/REAL(N)
            SUM_DP = SUM_DP + 1.0D0/DBLE(N)/DBLE(N)
            SUM_QUAD = SUM_QUAD + &
                &1.0_QUAD/REAL(N,KIND=QUAD)/REAL(N,KIND=QUAD)
          END DO
          WRITE(*,*) 'Summation baklänges av ',NMAX,' tal.'
        CASE DEFAULT
          WRITE(*,*) 'Svara F eller B!'
          GO TO 2
      END SELECT
      WRITE(*,*) 'Summan i enkel precision      = ', SUM_SP
      WRITE(*,*) 'Summan i dubbel precision     = ', SUM_DP
      WRITE(*,*) 'Summan i fyr-dubbel precision = ', SUM_QUAD
      END PROGRAM SUMMA
```

Enkel precision och dubbel precision deklareras som vanligt med `REAL` respektive `DOUBLE PRECISION`, medan för den fyrdubbla precisionen utnyttjas möjligheten att införa en egen precision, vilken jag ger namnet `QUAD`. Definitio-

nen utnyttjar funktionen `SELECTED_REAL_KIND` för att erhålla ett lämpligt slag (`KIND`-nummer).

```
% f90 summa.f90
% a.out
Hur många tal vill Du summera? 1e6
Vill Du summera framlänges (F) eller baklänges (B)? f
 Summation framlänges av      1000000  tal.
 Summan i enkel precision      =    1.644725
 Summan i dubbel precision     =    1.64493306684877
 Summan i fyr-dubbel precision =
                  1.64493306684872643630574849997952
% a.out
Hur många tal vill Du summera? 1e6
Vill Du summera framlänges (F) eller baklänges (B)? b
 Summation baklänges av      1000000  tal.
 Summan i enkel precision      =    1.644933
 Summan i dubbel precision     =    1.64493306684873
 Summan i fyr-dubbel precision =
                  1.64493306684872643630574849997939
%
```

Vi känner igen från numerisk analys [14, avsnitt 2.7] att man bör summera med början av talen med de minsta beloppen för att få högsta möjliga noggrannhet i resultatet.

9.7 Egna datatyper

Fortran har tidigare inte tillåtet användardefinierade datatyper. Detta blir nu möjligt.

```
TYPE stab_medlem
        CHARACTER(LEN=20) :: foernamn, efternamn
        INTEGER           :: id, avdelning
END TYPE
```

vilket kan utnyttjas för att beskriva en individ. En kombination av individer kan likaså bildas.

```
TYPE(stab_medlem), DIMENSION(100) :: stab
```

Individer kan refereras som `stab(nummer)` och ett visst fält kan refereras som `stab(nummer)%foernamn`. Man kan även kapsla definitioner

```
TYPE bolag
        CHARACTER(LEN=20)   :: bolagsnamn
        TYPE(stab_medlem), DIMENSION(100) :: stab
END TYPE
...
TYPE(bolag), DIMENSION(10) :: flera_bolag
```

Ett numeriskt mer intressant exempel är en gles matris `A` med högst hundra element skilda från noll, som kan deklareras med hjälp av

```
TYPE NONZERO
        REAL VALUE
        INTEGER ROW, COLUMN
END TYPE
```

och

```
TYPE (NONZERO) :: A(100)
```

Man får då värdet av `A(10)` genom att skriva `A(10)%VALUE`. Tilldelning kan ske med exempelvis `A(15) = NONZERO(17.0,3,7)`.

För att kunna använda användardefinierade datatyper i de föråldrade begreppenm `COMMON` eller `EQUIVALENCE`, eller för att se till att två likadana datatyper verkligen betraktas som samma datatyp användes kommandot `SEQUENCE`, i det senare fallet får dessutom ingen variabel vara deklarerad `PRIVATE`. Anledningen till dessa extraregler är att normalt är inte ordningen för komponenterna i en struktur fastställd. Kommandot `SEQUENCE` medför att komponenterna lagras i samma ordning som de uppräknas. Det skall alltid stå direkt efter `TYPE` kommandot. Följande exempel är hämtat från Überhuber och Meditz [34] och något modifierat. De påpekar att `SEQUENCE` inte är något egentligt attribut, eftersom det måste avse alla komponenterna.

```
TYPE BOK
   SEQUENCE
   CHARACTER (LEN = 50) :: FOERFATTARE, TITEL, FOERLAG
   INTEGER :: UTGIVNINGSAAR, ISBN_NUMMER
   REAL :: PRIS
END TYPE BOK
```

Kapitel 10

Avancerad användning av subrutiner och funktioner

Inledning

Det finns nu inte bara "vanliga" subrutiner och funktioner, utan dels finns flera varianter, som olika typer av lokala funktioner och subrutiner, dels ett antal attribut till de ingående argumenten. Inte minst intressant är den nya möjligheten med rekursiva programenheter.

10.1 Avsikt (`INTENT`)

De flesta funktioner och subrutiner har ett antal argument. Dessa kan delas in i tre kategorier:

1. Variabler som överför värden från den anropande programenheten till den anropade programenheten, men inte tvärtom. Dessa variabler kallas in-variabler och har således avsikten "in". De kan betecknas i Fortran 90 med `INTENT(IN)`.

2. Variabler som överför värden från den anropade programenheten till den anropande programenheten, men inte tvärtom. Dessa variabler kallas ut-variabler och har således avsikten "ut". De kan betecknas i Fortran 90 med `INTENT(OUT)`.

3. Variabler som både överför värden från den anropande programenheten till den anropade programenheten, liksom tvärtom. Dessa variabler kallas in/ut-variabler och har således avsikterna "in" och "ut". De kan betecknas i Fortran 90 med `INTENT(INOUT)`.

Genom att ge dessa attribut i koden ges systemet en möjlighet att kontrollera att de båda programenheterna samverkar på rätt sätt. Detta innebär att vid anropet måste en ut-variabel vara en variabel som kan ta emot ett värde, till exempel `A` eller `B(1,J)` men inte `SIN(A)` eller `C + 3.0`. Å andra sidan bör vid anropet en in-variabel redan ha tilldelats ett värde.

För en variabel som är både in och ut måste båda villkoren ovan vara uppfyllda.

Inuti den anropade programenheten bör värdet på en in-variabel användas, men inte ändras. För en utvariabel gäller att den vid inhoppet saknar värde, och således inte bör användas före den första tilldelningen.

För allt detta gäller att systemet har en möjlighet att kontrollera dessa olika villkor, i begränsad omfattning vid kompileringen (statisk kontroll) och i full omfattning vid exekvering (dynamisk kontroll). I praktiken så utför de nuvarande systemen ännu inte så mycket av dessa kontroller. Enligt standarden är det inget krav på att systemet utför dem! Det är dock större chans att kontrollerna sker om man inte bara använder `INTENT` utan även gränssnitt `INTERFACE`.

Vid dessa kontroller utnyttjas i stor omfattning tekniken med dataflödesanalys, se avsnittet 15.1, sid 128.

10.2 Gränssnitt (`INTERFACE`)

Vid ett par tillfällen saknar gamla Fortran möjlighet att kontrollera allt det som finns i nya Fortran, eftersom Fortran av tradition är baserat på separatkompilering av varje programenhet. För att råda bot på detta utnyttjas i stor omfattning tekniken med gränssnitt eller `INTERFACE`.

Med hjälp av ett gränssnitt överföres information mellan olika programenheter.

Eftersom de olika programenheterna i ett Fortran-program behandlas helt självständigt måste all information om argumenten i princip föras över manuellt. I Fortran 77 skedde detta med otympliga argumentlistor. I Fortran 90 kan i stället ett gränssnitt kallat `INTERFACE` användas. Detta måste användas vid

a) moduler som använder exempelvis egna datatyper,

b) anrop med nyckelordsargument eller underförstådda argument,

c) egna generiska rutiner,

d) fält med antaget mönster,

e) fält deklarerade med pekare,

f) vid definition av ny betydelse hos en `OPERATOR`,

g) om avsikt `INTENT` skall få verkan vid NAG-kompilatorn.

h) vid användning av subrutiner eller funktioner som argument (om man använder `IMPLICIT NONE`)

10.3 Satsfunktioner

Satsfunktion är en funktion som är lokal till en viss programenhet, och som finnes efter deklarationerna och före de exekverbara satserna. Den har en mycket enkel form, och fanns redan i FORTRAN I. En satsfunktion kan ej användas som argument! Se vidare avsnittet 4.3.1, sid 53.

En generellare form är den interna funktionen efter `CONTAINS`, se nästa avsnitt.

10.4 Interna funktioner

En intern funktion är en funktion som är lokal till en viss programenhet, och som finnes efter en `CONTAINS` sats. Alla variabler i den överordnade programenheten är direkt tillgängliga. En intern funktion kan ej användas utanför den direkt överordnade programenheten. Detta kan vara bra för att undvika namnkonflikter mellan funktioner och subrutiner från olika bibliotek. Interna funktioner är generellare än satsfunktioner. En intern funktion kan ej användas som argument! Det finns även interna subrutiner.

10.5 De nya inbyggda funktionerna

De inbyggda funktionerna är en mycket viktig del av Fortran. Som tidigare nämnts är de generiska, dvs de känner av argumentets datatyp och returnerar ett resultat baserat på denna typ, normalt även av denna typ. I vissa sammanhang (användning av funktion som argument) måste dock det specifika namnet användas (t ex `DSIN` istället för `SIN` för att ta sinus för ett dubbelprecisionsflyttal). Se vidare Bilaga D, sid 187.

10.6 Rekursiva funktioner

En helt ny möjlighet i Fortran 90 är rekursion. Notera att det krävs att man i funktionsdeklarationen ger en helt ny egenskap `RESULT (utvariabel)`. Denna resultat-variabel eller utvariabel användes inuti funktionen för att lagra funktionens värde. Vid själva anropet av funktionen, både externt och internt, användes däremot i stället det yttre eller gamlafunktionsnamnet. Användaren kan därför vid anrop av den rekursiva funktionen strunta i att det finns en resultat-variabel. Två enkla exempel gavs i kapitel 4, avsnittet **Rekursiva funktioner** på sid 45.

En annan viktig användning av det nya begreppet resultat-variabel är vid fältvärda funktioner, då det är lätt att deklarera denna variabel att lagra funktionens värde(n). Det är faktiskt kombinationen rekursivitet och fält som tvingat fram det nya begreppet.

10.7 Underförstådda argument och nyckelordsargument

Rutiner kan anropas med nyckelordsargument och kan utnyttja underförstådda argument, dvs en del argument kan ges med nyckelord i stället för med position, och en del behöver ej ges alls utan utnyttjar i stället ett standardvärde.

Användningen av nyckelord och underförstådda argument är inte riktigt så enkel som det borde vara. Det är ett av de fall då ett explicit gränssnitt `INTERFACE` erfordras. Jag har gett ett liknande exempel, utnyttjande en funktion, redan i kapitel 4, avsnittet om **Frivilliga argument** på sid 49. Nu tittar jag i stället på en subrutin.

Som nyckelord användes de formella parametrarna i gränssnittet, vilka ej behöver ha samma namn som de i den verkliga subrutinen. Dessa skall ej deklareras i anropande programenhet (utom i gränssnittet).

```
IMPLICIT NONE
INTERFACE
        SUBROUTINE SOLVE (A, B, N)
                INTEGER, INTENT (IN)        :: N
                REAL, INTENT(OUT)           :: A
                REAL, INTENT(IN), OPTIONAL  :: B
        END SUBROUTINE SOLVE
END INTERFACE

REAL X
CALL SOLVE(B=10.0,N=50,A=X)
WRITE(*,*) X
CALL SOLVE(B=10.0,N=100,A=X)
WRITE(*,*) X
CALL SOLVE(N=100,A=X)
WRITE(*,*) X
END

SUBROUTINE SOLVE(A,B,N)
IMPLICIT NONE
REAL, OPTIONAL, INTENT (IN) :: B
REAL :: A, TEMP_B
INTEGER :: N
IF (PRESENT(B)) THEN
        TEMP_B = B
ELSE
        TEMP_B = 20.0
END IF
A = TEMP_B + N
RETURN
END
```

Notera att `IMPLICIT NONE` för huvudprogrammet ej verkar i subrutinen `SOLVE`, varför denna har kompletterats med det kommandot och deklaration av de ingående variablerna. Körning på Sun-datorn ger

```
 60.0000000
  1.1000000E+02
  1.2000000E+02
```

Gränssnittet `INTERFACE` placeras lämpligen i en modul. Gränssnitten blir ett naturligt komplement till rutinbiblioteken, Fortran 90 söker under UNIX automatiskt efter moduler i aktuell filkatalog, eventuella filkataloger i I-listan, samt i något som liknar `/usr/local/lib/f90`. Begreppet I-lista är ett begrepp i UNIX för filkataloger, i detta fall kataloger med moduler.

Notera att om en utmatningsvariabel anges som både `OPTIONAL` och `INTENT (OUT)` så måste den vara med i anropslistan om programmet vid exekveringen lägger ut ett värde på denna variabel. Man måste därför i sitt program använda test med `PRESENT` av aktuell variabel, och endast om den är med i anropet använda den för tilldelning, för att på så sätt få den önskade valfriheten om man bara ibland vill ha ut en viss variabel.

Kapitel 11

Pekare

Inledning

Pekare har införts i Fortran 90, men inte på det vanliga sättet, som i de flesta andra språk med en egen datatyp, utan i stället som ett attribut till de andra datatyperna. Anledningen till det nya sättet är främst att pekare som en egen datatyp ökar risken för felaktig användning. En variabel med pekarattribut kan användas dels som en vanlig variabel, dels på ett antal nya sätt. Pekare i Fortran 90 är således inte minnesadresser som i flera andra programspråk eller i vissa Fortran-varianter, utan är snarare ett extra namn (alias).

På grund av skillnaden mellan pekare i Fortran 90 och Cray:s pekare i utvidgad Fortran 77 (CF77), vilken senare är mer lik en konventionell pekare, har Cray infört något som de kallar "Cray pointer" och "Cray character pointer" i sin implementering av Fortran 90. För ytterligare information hänvisas till referensmanualer från Cray. Detta är inte standard Fortran 90!

Den ökade säkerheten erhålls genom att inte bara varje variabel som skall användas som pekare måste ges attributet `POINTER`, utan att dessutom alla variabler som skall pekas på måste ges attributet `TARGET`. Detta krav gäller dock inte pekare, som kan peka på varandra utan något `TARGET` attribut. Ett exempel förklarar hur det fungerar.

```
REAL, TARGET      :: B(10,10)
REAL, POINTER     :: A(:,:)
A => B
```

Ovan har matrisen `B` deklarerats fullständigt, dvs dimensionen har angetts explicit. Dessutom har det angetts att den kan vara mål för en pekare. Matrisen `A`, som skall användas som pekare, måste deklareras som en matris, dvs ges rätt antal dimensioner (rang), men omfånget för dessa bestäms först vid tilldelningen (egentligen är det inte en tilldelning utan en pekar-associering), vilken sker med `=>`. Observera att pekar-tilldelningen inte innebär att data i matrisen `B` kopieras över till matrisen `A` (vilket skulle tagit relativt stora datorresurser i anspråk), utan det är en adress som genereras. Att "flytta" data med pekare är således mycket effektivt. Alternativt kan en pekare associeras med kommandot `ALLOCATE` och av-associeras med `DEALLOCATE`, i vårt fall till exempel

```
ALLOCATE (A(5,5))
```

```
DEALLOCATE (A)
```

Det finns en inbyggd funktion `ASSOCIATED` för att undersöka om en pekare är associerad (och även om den är associerad med ett visst mål) och ett kommando `NULLIFY` för att avsluta associeringen.

```
IF ( ASSOCIATED (A)   ) WRITE(6,*) ' A associerad'
IF ( ASSOCIATED (A,B) ) WRITE(6,*) ' A associerad med B'
NULLIFY (A)
```

Notera vidare att en pekare i Fortran 90 har både typ och rang, och att dessa måste överensstämma med motsvarande hos målet. Detta ökar säkerheten vid användning av pekare, man kan därför ej av misstag vid användningen av en viss pekare ändra värden på variabler av andra datatyper. Att man måste deklarera att en variabel kan vara mål ökar dessutom både säkerheten och kompileringseffektiviteten.

11.1 Enkla pekare

Det gäller att tänka sig för när man använder pekare. I det följande enkla exemplet tittar vi på vanliga skalära flyttal.

```
REAL, TARGET    :: A
REAL, POINTER   :: P, Q
A = 3.1416
P => A
Q => P
A = 2.718
WRITE(6,*) Q
```

Här blir värdet av `Q` lika med 2,718 eftersom både `P` och `Q` pekar på samma verkliga variabel `A`, och den har just ändrat värde från 3,1416 till 2,718. Vi gör nu en enkel variation.

```
REAL, TARGET    :: A, B
REAL, POINTER   :: P, Q
A = 3.1416
B = 2.718
P => A
Q => B
```

Här är nu värdet av både `A` och `P` lika med 3,1416 och värdet av både `B` och `Q` är 2,718. Om vi nu ger satsen

```
Q = P
```

så får samtliga fyra variabler värdet 3,1416, det får således samma effekt som

```
B = A
```

Om vi i stället gett satsen

```
Q => P
```

så hade de tre variablerna `A, P` och `Q` fått värdet 3,1416, medan `B` behållet värdet 2,718. I det senare fallet pekar nämligen `Q` bara på samma variabel som `P`, medan i det första fallet `Q` blir samma som `P`.

11.2 Pekare och fält

En enkel användning av pekare är att ge namn till fältsektioner.

```
REAL, TARGET    :: B(10,10)
REAL, POINTER   :: A(:), C(:)
A => B(4,:)   ! Vektorn A blir fjärde raden och
C => B(:,4)   ! vektorn C blir fjärde kolumnen
              ! av matrisen B
```

Det är inte nödvändigt att ta ut hela sektioner, utan man kan ta ut bara en del. I följande exempel tar vi ut en delmatris `FONSTER` av en stor matris `MATRIS`.

```
REAL, TARGET      :: MATRIS(100,100)
REAL, POINTER     :: FONSTER(:,:)
INTEGER           :: N1, N2, M1, M2
FONSTER => MATRIS(N1:M1,N2:M2)
```

Om man senare vill ändra dimensioneringen av delmatrisen `FONSTER` så är det bara att göra en ny pekar-tilldelning. Observera dock att indexen i `FONSTER` inte går från `N1` till `M1` respektive `N2` till `M2` utan från 1 till `M1-N1+1` respektive 1 till `M2-N2+1`.

Det finns inte fält av pekare direkt i Fortran 90, men man kan konstruera sådana genom att skapa en ny datatyp. Ett exempel är att lagra en vänstertriangulär matris som rader med varierande längd. Inför först den nya typen `RAD`.

```
TYPE RAD
        REAL, POINTER :: R(:)
END TYPE
```

och deklarera därefter de två vänstertriangulära matriserna `V` och `L` som vektorer av rader med varierande längd.

```
INTEGER   :: N
TYPE(RAD) :: V(N), L(N)
```

Därefter allokeras matrisen `V` enligt nedan (och motsvarande för `L`).

```
DO I = 1, N
   ALLOCATE (V(I)%R(1:I))  ! Varierande radlängd
END DO
```

Satsen

```
V = L
```

blir då ekvivalent med att göra

```
V(I)%R => L(I)%R
```

för alla komponenterna, dvs alla värdena på `I`.

Man kan även använda pekare för att deklarera en vektor på ett sådant sätt att den tilldelas sin storlek (sitt omfång) i en subrutin men kan användas i huvudprogrammet. Detta ger då en dynamisk minnesallokering. Detta har diskuterats utförligt i sektion 3.3.2, sid 37.

Notera att vid denna användning av pekare behövs inget deklarerat som `TARGET`, begreppet `POINTER` är därför snarare att betrakta som ett alternativ till `ALLOCATABLE`. Viktiga tillämpningar av pekare är listor och träd, liksom dynamiska fält.

Övningar

(11.1) Använd pekare för att på ett listigt sätt tilldela alla jämna element av en vektor värdet 13 och alla udda element värdet 17.

(11.2) Deklarera två pekare och låt den ena peka på en hel vektor och den andra på det sjunde elementet i samma vektor.

Kapitel 12

Diverse begrepp från Fortran 77

Inledning

Fortran 77 innehåller en del begrepp som inte så naturligt passar in i beskrivningen av Fortran 90. De ersättes i huvudsak av nyare begrepp i Fortran 90, och bör därför undvikas i så stor omfattning som möjligt.

12.1 COMMON

Med hjälp av `COMMON` kan man på ett smidigt, effektivt och riskfyllt sätt överföra data mellan olika programenheter (huvudprogram, subrutin, funktion). Härvid utnyttjas samma lagringsutrymme i minnet för de data som hör till de olika programenheterna.

```
COMMON A, B, C, D   ! I huvudprogrammet
COMMON A, B, X, C   ! I subrutinen
```

Om ovanstående båda satser finns med den första i huvudprogrammet och den andra i subrutinen kommer de båda variablerna `A` och `B` att vara gemensamma, medan `C` i huvudprogrammet heter `X` i subrutinen och `D` i huvudprogrammet heter `C` i subrutinen (vilket är förvirrande). Om de motsvarande variablerna har samma datatyp (och slag) kommer de även att ha samma värde.

Formella parametrar (argument) får ej ingå i `COMMON`!

Det finns två varianter av `COMMON`, nämligen dels ett som ser ut som ovan (utan namn, kallat blank `COMMON`), dels namnade `COMMON`. Det senare är av formen

```
COMMON / BLOCK1 / VAR1, VAR2
```

dvs namnet på COMMON-blocket ges inom snedstreck. Storheter i namnat `COMMON` får ej ingå i `SAVE`, men väl ett helt namnat `COMMON`, se vidare avsnittet 12.3. Även då skall namnet på `COMMON`-blocket ges inom snedstreck.

Varning: Om variablerna i ett `COMMON`-block ej överensstämmer, mellan de olika förekomsterna i de olika programenheterna, till ordning, typ och slag, samt

till storlek (antal element vid fält) blir det fel. Dessutom gäller att textsträngsvariabler (CHARACTER) ej får finnas i ett COMMON-block som även innehåller variabler av andra slag.

Blankt COMMON kan även skapas som ett namnat COMMON, men med ingenting (blankt) mellan snedstrecken. Därav namnet!

Om variabeln C i huvudprogrammet ovan är av typ DOUBLE PRECISION, och variablerna X och C i subrutinen är av typ REAL, så kommer C i huvudprogrammet att i normalfallet omfatta samma lagringsutrymme som X och C i subrutinen, varför D i huvudprogrammet och C i subrutinen inte kommer att ha något gemensamt. Normalt upptar ju en dubbelprecisionsvariabel dubbelt så stort lagrringsutrymme som en enkelprecisionsvariabel.

Initiering av variabler i ett COMMON-block sker geom att utnyttja BLOCK DATA.

I Fortran 90 kan man utnyttja moduler för att på ett säkrare sätt överföra bland annat data mellan olika programenheter. Se även sektion 13.4.3, sid 115, där ett exempel som något liknar COMMON ges.

Jag rekommenderar naturligtvis att göra om COMMON till moduler, och ger därför ett enkelt exempel nedan. Ett allvarligt problem uppstår dock i de fall som man utnyttjat någon av de ovan nämnda "konstigheterna", speciellt då möjligheten att en variabel har olika namn i olika programenheter. Detta kan ej realiseras med moduler, utan antingen måste man byta till ett konsekvent namn eller behålla COMMON.

I exemplet common.f90 nedan finns ett program som innehåller ett huvudprogram, en subrutin och en funktion samt utnyttjar ett blankt COMMON och ett namnat COMMON /BLOCK1/, vars variabler getts värden i BLOCK DATA BD1. Detta exempel har reviderats till modul.f90 där COMMON-variablerna flyttats till en modul MD1. På flera system krävs att modulen ligger först i filen, eller är i den första filen som kompileras/länkas!

```
PROGRAM TEST_AV_COMMON
        IMPLICIT NONE
        REAL :: A, B, C, D, E, F
        REAL, EXTERNAL :: FUN
        COMMON A, B
        COMMON /BLOCK1/ C, D
        WRITE(*,*) ' Början av programmet TEST_AV_COMMON'
        CALL SUB1(A, B)
        E = A + B
        F = FUN(A+C)
        WRITE(*,*) ' E = ', E,  ' F = ', F
        WRITE(*,*) ' Slutet av programmet TEST_AV_COMMON'
END PROGRAM TEST_AV_COMMON

BLOCK DATA BD1
        IMPLICIT NONE
        REAL :: C, D
        COMMON /BLOCK1/ C, D
        DATA C, D / 1.0, 19.0 /
END BLOCK DATA BD1
```

```
SUBROUTINE SUB1(X, Y)
        IMPLICIT NONE
        REAL :: A, B, C, D, X, Y
        COMMON A, B
        COMMON /BLOCK1/ C, D
        X = C
        Y = D
        WRITE(*,'(A)', ADVANCE='NO') '  Ge A = '
        READ(*,*) A
        WRITE(*,'(A)', ADVANCE='NO') '  Ge B = '
        READ(*,*) B
END SUBROUTINE SUB1

REAL FUNCTION FUN(X)
        IMPLICIT NONE
        REAL :: A, B, C, D, X
        COMMON A, B
        COMMON /BLOCK1/ C, D
        FUN = X + D
END FUNCTION FUN
```

Här följer det reviderade programmet som utnyttjar en modul i stället för `COMMON` och `BLOCK DATA`.

```
MODULE MD1
        IMPLICIT NONE
        REAL :: A, B, C, D
        DATA C, D / 1.0, 19.0 /
END MODULE MD1

PROGRAM TEST_AV_MODUL
        USE MD1
        IMPLICIT NONE
        REAL :: E, F
        REAL, EXTERNAL :: FUN
        WRITE(*,*) ' Början av programmet TEST_AV_MODUL'
        CALL SUB1(A, B)
        E = A + B
        F = FUN(A+C)
        WRITE(*,*) ' E = ', E,  ' F = ', F
        WRITE(*,*) ' Slutet av programmet TEST_AV_MODUL'
END PROGRAM TEST_AV_MODUL

SUBROUTINE SUB1(X, Y)
        USE MD1
        IMPLICIT NONE
        REAL :: X, Y
        X = C
        Y = D
        WRITE(*,'(A)', ADVANCE='NO') '  Ge A = '
        READ(*,*) A
```

```
        WRITE(*,'(A)', ADVANCE='NO') '  Ge B = '
        READ(*,*) B
END SUBROUTINE SUB1

REAL FUNCTION FUN(X)
        USE MD1
        IMPLICIT NONE
        REAL :: X
        FUN = X + D
END FUNCTION FUN
```

12.2 EQUIVALENCE

Med `COMMON` delar variabler mellan olika programenheter samma lagringsutrymme, men med `EQUIVALENCE` delar i stället olika variabler i samma programenhet på utrymme.

```
        REAL :: A, B, C, D, E
        REAL, DIMENSION(10) :: G
        EQUIVALENCE (A, B), (C, D, E)
        EQUIVALENCE (C, G(7))
```

Med ovanstående delar flyttalsvariablerna `A` och `B` på samma lagringsutrymme (och värde), medan flyttalsvariablerna `C, D` och `E` samt flyttalselementet `G(7)` delar på ett annat utrymme.

```
        COMPLEX CC
        REAL A
        EQUIVALENCE(CC, A)
```

Med ovanstående får flyttalsvariabeln `A` värdet av realdelen av det komplexa flyttalet `CC`.

```
        DOUBLE PRECISION D
        REAL A
        EQUIVALENCE (A,D)
```

Med ovanstående får flyttalsvariabeln `A` enkelprecisionsvärdet av dubbelprecisionsflyttalet `D` på en del datorsystem, men helt fel på de flesta datorsystem.

```
        REAL :: A
        INTEGER :: I
        EQUIVALENCE (A,I)
        I = 1
```

På de flesta datorsystem innebär ovanstående att flyttalet `A` blir ett litet tal, oftast noll.

Förutom för att spara lagringsutrymme kan `EQUIVALENCE` användas för att ta bort verkan av konsekventa stavfel. Om man har ett program med flyttalsvariabeln `DEFENCE` kan man ibland ha stavat den på amerikanska, dvs `DEFENSE`. Man kan göra dessa ekvivalenta med satsen

```
        EQUIVALENCE (DEFENCE, DEFENSE)
```

En bättre metod att åtgärda ett sådant fel är att bestämma sig för vilken stavning (motsvarande) som skall gälla och använda editorn till att ändra till rätt variabelnamn.

En annan användning av `EQUIVALENCE` var då man ville nollställa ett helt flerdimensionellt fält, men detta sker numera enklare med de kraftfulla fältkommandona, som i nedanstående exempel

```
REAL, DIMENSION(10,10,10) :: A
A = 0.0 ! Nollställning av hela fältet
```

Kommentaren på raden är bra för att påminna läsaren om att det inte är en enkel (skalär) variabel som skall nollställas, utan ett helt fält.

I Fortran 90 bör man helst undvika `EQUIVALENCE`, det kan dock ibland behöva utnyttjas när man vill kontrollera (simulera) vad som sker vid användning av fel datatyp vid exempelvis funktionsanrop.

12.3 SAVE

Detta är ett begrepp som infördes i Fortran 77 och inte fanns i tidigare standard, och knappast heller i tidigare implementationer.

Anledningen är det välkända fenomenet att vid uthopp från en funktion eller subrutin så glöms automatiskt värdena på alla lokala variabler (utom de som antingen givits begynnelsevärden med `DATA`-sats eller motsvarande och ej modifierats, eller de som getts som permanenta konstanter med satsen eller attributet `PARAMETER`). Anledningen till denna "glömska" är att man vid behov av minnesutrymme skall kunna växla ut (eng. swap) rutinen från primärminnet till skivminnet. Om inga värden behöver sparas från exekveringen behöver man då bara på nytt kopiera in rutinen till primärminnet från skivminnet, men aldrig kopiera tillbaks den från primärminnet till skivminnet. Man undviker således halva arbetet!

För att kunna spara värden har därför satsen (och attributet) `SAVE` införts. Om det står ensamt på en rad bland deklarationerna innebär det att alla variabler skall sparas. Alternativt ger man det som en vanlig deklaration, dvs man räknar efter kommandot `SAVE` upp de variabler och/eller `COMMON` block som skall sparas. Observera att enskilda variabler i `COMMON` block ej kan sparas, utan bara hela namnade `COMMON` block.

I ett eventuellt onamnat (blank) `COMMON` sparas automatiskt (obligatoriskt) alla variabler. För övriga `COMMON` block gäller att värdena sparas automatiskt om `COMMON` blocket finns längre upp i anropssekvensen av funktioner och subrutiner. Klarare uttryckt, om en subrutin anropar två subrutiner som innehåller samma namnade `COMMON` block bör detta `COMMON` block även finnas i den anropande subrutinen.

Kapitel 13

Program-uppbyggnad

Inledning

Förutom de fyra gamla programenheterna `PROGRAM` (dvs huvudprogrammet), subrutin, funktion och `BLOCK DATA` har tillkommet moduler (`MODULE`) samt en hel del nyheter i de gamla enheterna. Jag upprepar att subprogram är ett sammanfattande namn på subrutin och funktion. Om man är varsam med det svenska språket bör man dock skriva underprogram och underrutin.

Jag vill även upprepa att under Fortran 77 är alla programenheter i stort sett på samma nivå, även om huvudprogrammet logiskt sett verkar vara överordnat de subrutiner och funktioner som anropas, och att man till och med kan rita upp en anropsgraf som ser ut som ett träd. I verkligheten ligger eventuella `BLOCK DATA` på en högre nivå, och alla andra programenheter ligger på samma nivå, ur Fortransystemets synpunkt, dock med huvudprogrammet ett litet snäpp högre. Ett undantag är de så kallade satsfunktionerna, vars definitioner kan ligga först i en programenhet (direkt efter deklarationerna), och som är interna för den enheten, och således befinner sig på en logiskt sett lägre nivå. Den normale Fortranprogrammeraren använder dock tyvärr aldrig sådana funktioner.

Ovanstående innebär att alla rutinnamn är på samma logiska nivå, vilket innebär att två olika rutiner i helt olika delar av ett stort program inte får ha samma namn. Ofta innehåller numeriska och grafiska bibliotek tusentals rutiner (med högst sex tecken i namnen under gamla Fortran), varför det är en påtaglig risk för namnkonflikt. Detta var en sak som de gamla satsfunktionerna kunde avhjälpa, eftersom de är interna för respektive enhet, och därför kan finnas under samma namn men med olika uppgift i olika programenheter. Nackdelen är att de bara kan hantera vad som ryms på en programrad (men de kan anropa varandra på så sätt att en senare kan anropa en tidigare, men ej tvärtom).

Nu har interna funktioner och subrutiner tillkommet. Dessa definieras sist i respektive programenhet (dock ej i `BLOCK DATA`) efter det helt nya kommandot `CONTAINS` och före `END`. Ett internt subprogram har tillgång till samma variabler som enheten den tillhör, inklusive möjligheten att anropa dess andra interna subprogram. Det skrives i övrigt som ett vanligt subprogram, men får ej kapslas, dvs ej ha egna interna funktioner eller subrutiner.

Vanliga subrutiner och funktioner kallas liksom tidigare externa subrutiner och externa funktioner, men nu finns det en större anledning till den benämning-

en än tidigare. Numera finns ju även interna subprogram, tidigare fanns bara inbyggda (intrinsic) som alternativ. För övrigt har antalet inbyggda funktioner ökat mycket kraftigt.

I variabeldeklarationer till subprogram har för argumenten tillkommet möjligheten att ange om en variabel är invariabel, utvariabel, eller båda samtidigt. Detta anges med `INTENT` som kan vara `IN, OUT` eller `INOUT`. Om `IN` gäller så kan det verkliga argumentet vid anropet vara ett uttryck som `X+Y` eller `SIN(X)` eller en konstant som `37`, eftersom ett värde skall överföras till subprogrammet men ej åter till anropande enhet. Variabeln får i detta fall ej tilldelas något nytt värde i subprogrammet. Om `OUT` gäller så måste däremot det verkliga argumentet vara en variabel. Vid inträde i subprogrammet anses då variabeln som odefinierad. Det tredje fallet täcker båda möjligheterna, ett värde in, ett annat (eller eventuellt samma) ut. Även då måste naturligtvis det verkliga argumentet vara en variabel. Om argumentet har ett pekar-attribut så får `INTENT` ej sättas. Implementeringen av `INTENT` är dock ännu ej fullständig.

Den ena användningen för den nya programenheten `MODULE` är att ta hand om globala data, och den ersätter då `BLOCK DATA`, den andra är att paketera nya datatyper.

13.1 Huvudprogram

Huvudprogrammet är den enda obligatoriska programenheten i ett Fortranprogram, så ger exempelvis Cray:s Fortran 90 kompilator en kraftig varning om huvudprogram saknas vid kompilering. Ett huvudprogram kan inledas med den helt frivilliga satsen

```
PROGRAM programnamn
```

och måste avslutas med

```
END
```

eller

```
END PROGRAM
```

eller

```
END PROGRAM programnamn
```

Observera att ordet `PROGRAM` måste vara med i `END` satsen om programnamnet är det!

Det kan vara lämpligt att låta huvudprogrammet vara ett mycket kort program, som bara består av viss grundläggande inläsning samt anrop av en del av de olika subrutiner och funktioner som tillsammans med huvudprogrammet bildar programmet.

13.2 Subrutiner

En subrutin (eng. subroutine) är en programenhet som ej returnerar något funktionsvärde genom sitt namn och som har ett antal parametrar (eng. arguments).

Det är tillåtet att subrutinen ändrar värdet på ett eller flera av argumenten. En subrutin kan dock ha en annan uppgift, till exempel att läsa in eller skriva ut data. En subrutin har, till skillnad från en funktion, ingen typ.

Antalet subrutiner i ett program kan vara noll. Det kan finnas externa och interna subrutiner, dessutom kan de sex i Fortran inbyggda (eng. intrinsic) subrutinerna användas.

13.3 Funktioner

En funktion (eng. function) är en programenhet som returnerar ett funktionsvärde (eventuellt flera, dvs ett fält) i sitt namn och som har ett antal parametrar. Det är faktiskt tillåtet med funktioner utan parametrar (i detta fall måste dock parameterlistan ges i form av vänsterparentes och högerparentes, utan något emellan). Det är likaså tillåtet men ej tillrådligt att funktionen ändrar värdet på ett eller flera av argumenten.

Antalet funktioner i ett program kan vara noll. Det kan finnas externa och interna funktioner, samt satsfunktioner, dessutom kan de många i Fortran inbyggda (eng. intrinsic) funktionerna användas.

13.4 Moduler

En modul innehåller deklarationer (specifikationer) och definitioner som skall användas av de andra program-enheterna. Ersätter `BLOCK DATA`, som bara kunde utnyttjas för att tilldela initialvärden till storheter i `COMMON` block, se vidare sektion 13.5.

Moduler illustreras här med tre exempel, nämligen en för byte av två variabler (av samma men varierande typ), en för intervallaritmetik och slutligen en som bestämmer vissa standardvärden, bland annat vilken flyttalsprecision (snarare maskinprecision) som skall användas.

13.4.1 Enkelt exempel på modul

I sektion 4.2.1, sid 51, gav jag ett fullständigt exempel på en rutin `SWAP(A,B)` som byter plats på `A` och `B`, utnyttjande olika underliggande rutiner beroende på om de båda variablerna är av typ `REAL`, `INTEGER` eller `CHARACTER`. Användningen av detta var inte så trevlig, eftersom man var tvungen att ha en "massa" `INTERFACE` som definierade de tre olika fallen. Allt detta kan nu döljas i en modul.

Man flyttar över allt som rör `SWAP` till en modul, vilken sedan kan användas i huvudprogrammet med satsen `USE modulnamnet`. Notera att i modulens `INTERFACE` skall den speciella satsen `MODULE PROCEDURE` användas för att undvika att rutinerna specificeras både i `INTERFACE` och i `CONTAINS`. Vid användning måste naturligtvis både modulen och huvudprogrammet länkas ihop, till exempel med satsen `f90 del1.f90 del2.f90`. Här följer närmast modulen:

```
MODULE SWAP_MODUL
        INTERFACE SWAP
                MODULE PROCEDURE SWAP_R, SWAP_I, SWAP_C
        END INTERFACE
CONTAINS

        SUBROUTINE SWAP_R(A,B)
        IMPLICIT NONE
        REAL, INTENT (INOUT) :: A, B
        REAL                 :: TEMP
                TEMP = A ; A = B ; B = TEMP
        END SUBROUTINE SWAP_R

        SUBROUTINE SWAP_I(A,B)
        IMPLICIT NONE
        INTEGER, INTENT (INOUT) :: A, B
        INTEGER                 :: TEMP
                TEMP = A ; A = B ; B = TEMP
        END SUBROUTINE SWAP_I

        SUBROUTINE SWAP_C(A,B)
        IMPLICIT NONE
        CHARACTER, INTENT (INOUT) :: A, B
        CHARACTER                 :: TEMP
                TEMP = A ; A = B ; B = TEMP
        END SUBROUTINE SWAP_C
END MODULE SWAP_MODUL
```

Här följer så huvudprogrammet, vilket nu är rensat på all ointressant information om `SWAP`.

```
PROGRAM SWAP_HUVUD
USE SWAP_MODUL
        IMPLICIT NONE
        INTEGER   :: I, J, K, L
        REAL      :: A, B, X, Y
        CHARACTER :: C, D, E, F

        I = 1   ; J = 2     ;   K = 100 ; L = 200
        A = 7.1 ; B = 10.9  ;   X = 11.1; Y = 17.0
        C = 'a' ; D = 'b'   ;   E = '1' ; F = '"'

        WRITE (*,*) I, J, K, L, A, B, X, Y, C, D, E, F
        CALL SWAP(I,J) ; CALL SWAP(K,L)
        CALL SWAP(A,B) ; CALL SWAP(X,Y)
        CALL SWAP(C,D) ; CALL SWAP(E,F)
        WRITE (*,*) I, J, K, L, A, B, X, Y, C, D, E, F
END
```

13.4.2 Intervallaritmetik

Som ett ganska stort exempel på en modul så försöker jag mig på att införa ett paket för intervallaritmetik. Till varje närmevärde `X` hör då ett intervall `[X_lägre ; X_övre]`. Vid användning av paketet vill man ha det så enkelt att man bara ger variabeln `X` då man menar intervallet. Variabeln `X` skall då ha en ny datatyp, intervall. Följande ligger på filen `intervall_aritmetik.f90` eller `intv_ari.f90`, som bland annat innehåller ett gränssnitt, `INTERFACE`.

```
MODULE INTERVALL_ARITMETIK
        TYPE INTERVALL
                REAL LAEGRE, OEVRE
        END TYPE INTERVALL

        INTERFACE OPERATOR (+)
           MODULE PROCEDURE ADDERA_INTERVALL
        END INTERFACE
        INTERFACE OPERATOR (-)
           MODULE PROCEDURE SUBTRAHERA_INTERVALL
        END INTERFACE

        INTERFACE OPERATOR (*)
           MODULE PROCEDURE MULTIPLICERA_INTERVALL
        END INTERFACE
        INTERFACE OPERATOR (/)
           MODULE PROCEDURE DIVIDERA_INTERVALL
        END INTERFACE
CONTAINS
        FUNCTION ADDERA_INTERVALL(A,B)
           TYPE(INTERVALL), INTENT(IN) :: A, B
           TYPE(INTERVALL) :: ADDERA_INTERVALL
           ADDERA_INTERVALL%LAEGRE = A%LAEGRE + B%LAEGRE
           ADDERA_INTERVALL%OEVRE  = A%OEVRE  + B%OEVRE
        END FUNCTION ADDERA_INTERVALL

        FUNCTION SUBTRAHERA_INTERVALL(A,B)
           TYPE(INTERVALL), INTENT(IN) :: A, B
           TYPE(INTERVALL) :: SUBTRAHERA_INTERVALL
           SUBTRAHERA_INTERVALL%LAEGRE = A%LAEGRE - B%OEVRE
           SUBTRAHERA_INTERVALL%OEVRE  = A%OEVRE  - B%LAEGRE
        END FUNCTION SUBTRAHERA_INTERVALL

        FUNCTION MULTIPLICERA_INTERVALL(A,B)
!                POSITIVA TAL FÖRUTSÄTTES
           TYPE(INTERVALL), INTENT(IN) :: A, B
           TYPE(INTERVALL) :: MULTIPLICERA_INTERVALL
           MULTIPLICERA_INTERVALL%LAEGRE = &
                          A%LAEGRE * B%LAEGRE
           MULTIPLICERA_INTERVALL%OEVRE  = &
                          A%OEVRE  * B%OEVRE
        END FUNCTION MULTIPLICERA_INTERVALL
```

```
        FUNCTION DIVIDERA_INTERVALL(A,B)
!                POSITIVA TAL FÖRUTSÄTTES
           TYPE(INTERVALL), INTENT(IN) :: A, B
           TYPE(INTERVALL) :: DIVIDERA_INTERVALL
           DIVIDERA_INTERVALL%LAEGRE = A%LAEGRE / B%OEVRE
           DIVIDERA_INTERVALL%OEVRE  = A%OEVRE  / B%LAEGRE
        END FUNCTION DIVIDERA_INTERVALL
END MODULE INTERVALL_ARITMETIK
```

Vid kompilering av ovanstående skapas en fil `intervall_aritmetik.mod` eller `intervall_aritmetik.M` som innehåller en intressant modifierad version av koden ovan. En del system genererar en begriplig sådan, en del en för människan obegriplig. Ett program som vill utnyttja detta paket inkluderar satsen `USE INTERVALL_ARITMETIK` först bland specifikationssatserna, då finns direkt både datatypen `INTERVALL` och de fyra räknesätten på denna typ tillgängliga. I en del fall är det önskvärt att bara inkludera en del av faciliteterna i en modul, detta sker med `ONLY` enligt nedan.

```
USE modul_namn, ONLY : lista_över_utvalda_rutiner
```

Följande är ett exempel på ett mycket enkelt huvudprogram för test av intervallaritmetiken. Det ligger på filen intervall.f90 eller intv.f90.

```
USE INTERVALL_ARITMETIK
IMPLICIT NONE
TYPE (INTERVALL) :: A, B, C, D, E, F
A%LAEGRE = 6.9
A%OEVRE  = 7.1
B%LAEGRE = 10.9
B%OEVRE  = 11.1
WRITE (*,*)  A,  B
C = A + B
D = A - B
E = A * B
F = A / B
WRITE (*,*)  C, D
WRITE (*,*)  E, F
END
```

Körning av detta program på Sun-dator med NAG:s Fortran 90 kompilator följer.

```
f90 intervall.f90 intervall_aritmetik.f90
intervall.f90:
intervall_aritmetik.f90:
a.out
   6.9000001   7.0999999  10.8999996  11.1000004
  17.7999992  18.2000008  -4.2000003  -3.7999997
  75.2099991  78.8100052   0.6216216   0.6513762
```

men det riktiga kommandot är

```
f90 intervall_aritmetik.f90 intervall.f90
```

dvs modulen skall komma före användningen!

I ovanstående exempel har betydelsen av bland andra tecknet + generaliserats till att gälla även intervall. Fortran 90 innehåller en spärr mot att definiera om betydelsen av + på vanliga variabler.

Som framgår av Laboration F.12 har Sun i sin Fortran-kompilator lagt in en verklig utvidgning till att korrekt hantera intervallaritmetik.

Övningar

(13.1) Komplettera modulen så att paketet klarar godtyckliga tecken på talen även vid multiplikation och division.

(13.2) Komplettera modulen så att paketet utför lämplig felhantering vid division med ett intervall som innehåller noll.

(13.3) Komplettera även så att hänsyn tas till det lokala avrundningsfelet vid operationen.

13.4.3 Modul för standardparametrar

En trevlig användning av modul är att där stoppa in dels parametrar som ofta användes i beräkningen, dels specifikation av önskad noggrannhet.

Det följande programmet är ett enkelt exempel på användning av en sådan modul.

```
MODULE START
IMPLICIT NONE
INTEGER, PARAMETER :: AP = SELECTED_REAL_KIND(14, 300)
REAL (KIND=AP), PARAMETER :: ONE_TENTH = 0.1_AP
END MODULE START

PROGRAM TEST_START
USE START
REAL (KIND=AP) :: PI, X, Y, Z
PI = 4.0_AP*ATAN(1.0_AP)
X = 0.1
Y = 10*(X - ONE_TENTH)
Z = 10*ONE_TENTH - 1.0_AP
WRITE(*,*) X, ONE_TENTH
WRITE(*,*) Y, Z
END PROGRAM TEST_START
```

Utmatningen visar dels att det hela fungerar, dels vikten av att explicit ange dubbel precision eller motsvarande vid konstanter som inte är exakta i "kort" precision. Tyvärr har jag inte hittat något bra sätt att ge π ett värde i arbetsprecisionen redan i modulen.

```
0.1000000014901161      0.1000000000000000
1.4901161138336505E-08  0.0000000000000000E+000
```

13.4.4 Diverse om moduler

I en modul kan vissa begrepp definieras som `PRIVATE`, vilket innebär att programenheter utanför modulen ej kan nå dessa. Ibland utnyttjas även en explicit `PUBLIC` deklaration, normalt är dock `PUBLIC` underförstått. Genom att ge följande satser

```
PRIVATE
PUBLIC            :: VAR1
```

blir samtliga variabler utom `VAR1` lokala, medan `VAR1` blir globalt tillgänglig. Notera att båda dessa begrepp antingen kan ges som kommandon, till exempel

```
INTEGER           :: IVAR
PRIVATE           :: IVAR
```

eller som attribut

```
INTEGER, PRIVATE  :: IVAR
```

och motsvarande för `PUBLIC`.

13.5 `BLOCK DATA` programenheter

Variabler i `COMMON` får ej tilldelas initialvärden med `DATA`-sats, utom i en speciell programenhet kallad `BLOCK DATA`. Anledningen till detta är att annars blir det svårt för systemet att avgöra i vilken programenhet som initieringen skall ske. Denna nya programenhet består av följande element

```
BLOCK DATA namn
Deklarationer
Datasatser
END
```

Ett enkelt exempel följer, där det första blocket även sparas. Det är inte nödvändigt att alla variabler i ett sådant block har initierats, i exemplet nedan har således inte heltalsvariabeln `ANTAL` getts något startvärde.

```
BLOCK DATA START
INTEGER :: VEK(10), ANTAL
CHARACTER*29 :: ALFA

COMMON /BLOCK1/ VEK, ANTAL
COMMON /BLOCK2/ ALFA

SAVE /BLOCK1/

DATA VEK / 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 /
DATA ALFA /'ABCDEFGHIJKLMNOPQRSTUVWXYZÅÄÖ'/

END
```

13.6 INCLUDE satsen

INCLUDE kan användas för att inkludera källkod från en extern fil. Konstruktionen är att på en rad finns INCLUDE samt en textsträng samt eventuellt avslutande kommentar, men satsnummer är ej tillåtet. Tolkningen är implementationsberoende, normalt tolkas textsträngen som namnet på den fil som skall inkluderas på det ställe i källkoden där INCLUDE satsen finns. Kapsling är tillåten (antalet nivåer är implementationsberoende), men ej rekursion. Ett par exempel från UNIX följer, där subrutinen i filen cls.f inkluderas, antingen direkt från aktuell filkatalog, eller med en relativ respektive absolut filadress. Däremot har jag inte lyckats att använda symboliska namn från UNIX.

```
INCLUDE 'cls.f'
INCLUDE '../fortran90/cls.f'
INCLUDE '/mailocal/lab/numt/TANA70/cls.f'
```

Denna sats fanns i flera utvidgningar till Fortran 77 och användes främst för att lägga in identiska kopior av COMMON-block i flera subrutiner och funktioner. Som tidigare påpekats är det ju väsentligt att speciellt namnade COMMON-block är identiska varje gång de uppträder.

Under Fortran 90 kan man med fördel i stället använda moduler. En fördel med att inkludera programfiler med INCLUDE är att det kan röra sig om ofullständiga programavsnitt som uppträder på flera ställen, medan moduler måste följa vissa syntax-regler. Observera dock att sedan avsnittet inkluderats måste resultatet följa Fortrans syntaxregler. I exemplet ovan, som inkluderar en hel subrutin, måste därför INCLUDE satsen antingen ligga först eller efter ett END (svarande mot en programenhet, ej bara END DO eller END IF).

13.7 Ordningen mellan satser

Ordningen mellan de olika programenheterna är i nästan alla sammanhang godtycklig. I enstaka fall har jag noterat att ett huvudprogram måste vara först, men jag har inte funnit detta krav konsistent, det är bara ibland som kravet funnits på vissa maskiner.

Däremot så är inte ordningen mellan satserna i en programenhet godtycklig. Den första satsen skall vara PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA eller MODULE, den sista satsen skall vara END. Det gäller dock, för att uppnå kompatibilitet bakåt mot gamla Fortranversioner, att satsen PROGRAM i ett huvudprogram är frivillig. En programenhet skall vara uppbyggd på följande sätt

```
Specifikation av programenheten
        USE modul
        IMPLICIT NONE
        Övriga IMPLICIT satser
        INTERFACE
        Deklarationer
        Satsfunktioner
        Exekverbar del
CONTAINS
        Interna subrutiner och funktioner
END
```

Det enda som är obligatoriskt är `END`-satsen, ett helt korrekt huvudprogram (och därmed även program) kan bestå av enbart satsen `END`. Det programmet utför naturligtvis ingenting, det kan sägas vara det enklast möjliga Fortran-programmet, och kan naturligtvis användas för test. Jag prövade det under NAG:s kompilator på Sun och fick inga felutskrifter och ett körbart program på 73 728 bytes, medan det under Sun:s Fortran 77 kompilator gav hela 188 416 bytes.

`FORMAT`-satser kan finnas var som helst mellan `USE` och `CONTAINS`, men det är lämpligt att samla dom i början eller slutet eller, vilket är vanligast, i direkt samband med den första läs- eller skrivsats som använder den.

Likaså kan `DATA`-satser och `PARAMETER`-satser placeras nästan var som helst, men jag rekommenderar varmt att de placeras bland deklarationerna (eftersom de ej är exekverbara).

Även eventuella `ENTRY`-satser kan placeras ganska fritt, men här rekommenderar jag inplacering bland de exekverbara satserna. `ENTRY`-satsen ger en möjlighet till en alternativ ingång i en subrutin eller funktion. Jag avråder dock bestämt från dess användning.

I de specifika namnen på de inbyggda funktionerna inledes namnen på komplexa funktioner med ett C och funktioner i dubbel precision med ett D. Detta system användes även av en del programbibliotek, som LAPACK, och då utvidgat med ett S som första tecken i vanlig enkel precision och Z vid komplexa tal i dubbel precision (men komplex dubbel precision finns ej i standard Fortran 77). Vid fyrdubbel precision tillämpas ibland Q som första bokstav. Hur man kan skapa de båda senare precisionerna utnyttjande det nya precisionsbegreppet framgick av kapitel 9.3, sid 89.

Kapitel 14

Inkompatibilitetsproblem

Inledning

- Det finns nu två former av källkod, dels den gamla hålkortsorienterade (fix form), dels den nya fria formen. Dessa två former får ej blandas utan måste särskiljas vid kompileringen. Ett gammalt program (en eller flera programenheter i form av huvudprogram, subrutin eller funktion) som kompileras som fri form kan eventuellt ge annat resultat än tidigare, kompileringsfel är sannolika.
- Man kan i ett och samma program blanda programenheter skrivna i fix form och fri form, men varje enhet måste vara i endast en form, och vid kompileringen får normalt ej båda formerna finnas i samma källkodsfil. Dock tillåter vissa system ett direktiv mellan programenheterna i samma fil som talar om för kompilatorn vilken form som gäller.
- Det går att fritt blanda gamla och nya kommandon, men man bör trots detta söka vara konsekvent.
- Notera att när man byter kompilator kan "nya" fel uppträda på grund av att den gamla kompilatorn inte var lika strikt som den nya. Normalt är det så att en ny kompilator upptäcker en del gamla men tidigare dolda fel. I specifikationen för Fortran 90 ingår kravet på att fel skall hittas i möjligaste mån redan vid kompileringen.
- Den större kraftfullheten i Fortran 90 medför även att en del kommandon hör ihop på olika sätt i olika sammanhang, varför det är omöjligt att beskriva språket enbart kommando för kommando.

14.1 Inkompatibilitet mellan olika Fortran 90 implementationer

Den nya standarden har en sak som tyvärr lämnar fältet fritt för variationer mellan olika implementationer. Det är slagen `KIND`, vilka kan ges olika heltalsvärden svarande mot olika precisioner. Härvid har NAG valt att använda 1 för enkel precision och 2 för dubbel precision, medan Cray har baserat sig på IBM:s gamla

utvidgning av Fortran 66 genom att ange antalet bytes i ordet. NAG använder exempelvis `KIND=2` för dubbel precision medan Cray i stället skriver `KIND=16`. Det kan här nämnas att å andra sidan har NAG valt att lägga in IBM:s variant `REAL*4` som synonym för `REAL` och `REAL*8` som synonym för `DOUBLE PRECISION` i sin Fortran 90.

14.2 Skillnad i behandling av logiska variabler

Den nya standarden är mer ordrik och explicit, varför en del regler är tydligare formulerade och sannolikheten därför är större att de måste följas. Ett exempel är jämförelse av logiska variabler, som under Fortran 90 måste ske med `.EQV.` eller `.NEQV.`, medan detta i praktiken gick i Fortran 77 även med `.EQ.` och `.NE.`

14.3 Små saker av stor betydelse

Man använde ofta tidigare variabelnamnet `SUM` för att exempelvis lagra det temporära värdet vid en summation i en `DO`-slinga. Det namnet är numera mindre lämpligt, eftersom `SUM` är namnet på en automatisk summation, se avsnittet om fältfunktioner i Bilaga D.14, sid 196. Andra farliga variabelnamn kan vara `ALL, HUGE, INDEX, INT, KIND, MASK, PRESENT, REPEAT, SCALE, SIZE, TINY` och `TRIM`. Om man använder ett "upptaget" namn blir den normala effekten att den inbyggda funktionen blir otillgänglig.

I en del gamla Fortran-dialekter användes `TYPE` för utskrift på skrivmaskinsterminalen och `PRINT` för utskrift på radskrivaren. Begreppet `TYPE` har nu fått en helt ny betydelse, nämligen för att deklarera användardefinierade datatyper.

14.4 Undertryckning av radframmatning

Ett problem som kan uppstå vid flyttning från Fortran 77 till Fortran 90 beror på en vanlig avvikelse från standard. Den avvikelse jag tänker på är användningen i `FORMAT` av $, för att undertrycka radframmatningen innan användaren skall ge ett värde, vilket är en vanlig utvidgning av Fortran 77. Eftersom denna avvikelse normalt ej finns i Fortran 90 erhålles där kompileringsfel, varför en annan lösning måste sökas.

Under många Fortran 77 implementationer skrev man således

```
      PROGRAM TEST
      REAL X
      WRITE(*,10)
10    FORMAT(' GE X = ',$)
      READ(*,*) X
      WRITE(*,*) X
      END
```

I Fortran 90 använder man i stället icke-avancerande in/utmatning eller "non-advancing I/O", man skriver därför skrivsatsen på följande sätt

```
      WRITE(*,'(A)',ADVANCE='NO') ' GE X = '
```

Dessa båda program ger samma resultat, man blir uppmanad att ge värdet på variabeln X på samma rad som texten

```
GE X =
```

Icke avancerande in/utmatning kan ej användas på liststyrd in/utmatning eller på `NAMELIST`.

14.5 Varierande system för matriser

Eftersom Fortran 77 saknar dynamisk minnesallokering måste man där "ta till" en tillräcklig dimensionering i anropande programenhet, och hålla ordning på ledande dimension"i den anropade programenheten. När man utnyttjar Fortran 90 rutiner vill man dock i allmänhet ha ett fält som överensstämmer med matrisens logiska storlek. En tilldelning som åstadkommer denna transformation är enkel att göra. Vi antar att en kvadratisk matris från den anropande programenheten finns i aktuell programenhet som fältet `A` med dimensioneringen `A(IA,*)`, dvs med `IA` som den ledande dimensionen, och att vi vill föra över värdena till ett fält `B`, med dimensioneringen `B(N,N)`, där `N` samtidigt är matrisens matematiska dimension. Följande sats utför önskad operation om `IA >= N`.

```
B = A(1:N,1:N)
```

14.6 Deklarationer

Deklaration av variabler kan nu samlas i en sats per variabel. Under Fortran 77 deklarerade man till exempel

```
REAL A, B, C
PARAMETER (A = 3.141592654)
DIMENSION B(3)
DATA B / 1.0, 2.0, 3.0 /
DIMENSION C(100)
DATA C /100*0.0/
```

dvs en variabel kunde förekomma på flera rader. Under Fortran 90 kan man naturligtvis skriva som tidigare, men man bör i stället skriva

```
REAL, PARAMETER         :: A = 3.141592654
REAL, DIMENSION(1:3)    :: B = (/ 1.0, 2.0, 3.0 /)
REAL, DIMENSION(1:100) :: C
DATA C /100*0.0/
```

14.7 Bakåt- och framåtkompatibilitet

Mycket viktigt vid införandet av en ny programspråksstandard är att gamla program (åtminstone sådana som följer den utgående standarden) kan användas även under den nya.

När man gick från Fortran 66 till Fortran 77 tog man bort den utvidgade `DO`-slingan (möjligheten att, om man inte ändrar några av `DO`-slingans styrparametrar, hoppa ut ur slingan och sedan hoppa in igen, dvs något av motsatsen till strukturerad programmering) och Hollerith-konstanter (utom i `FORMAT`). Detta innebär att det finns program som uppfyller Fortran 66 men ej uppfyller Fortran 77. De flesta leverantörer har dock valt att låta dessa två begrepp ligga som utvidgningar i sin version av Fortran. För Fortran 90 gäller att inget tagits bort från Fortran 77. En viktig praktisk fråga blir dock huruvida leverantörerna kommer att fortsätta att behålla de båda saker som egentligen skulle ha försvunnit då Fortran 77 kom. Detta är tillåtet.

Innan antagen dimension (asterisken * som sista dimension) infördes i samband med Fortran 77 fuskademan i stället med att ge en etta (1) för den sista dimensioneringen. Denna metod fungerar inte alls under de flesta Fortran 90 system! Tre mycket små skillnader mellan Fortran 77 och Fortran 90 har noterats av Cray, och gäller allmänt.

- Genom att Fortran 90 har så många nya inbyggda funktioner och dessutom några inbyggda subrutiner kan det hända att ett användarprogram har en egen funktion eller subrutin med samma namn, och att då felprogramenhet kommer att utnyttjas. Detta kan undvikas genom att konsekvent deklarera de egna programenheterna som `EXTERNAL`.

- Vi har redan i avsnitt 7.1.1, sid 73, diskuterat att det föreligger en skillnad mellan Fortran 77 och Fortran 90 om inmatningslistan inte innehåller lika många element som svarar mot formateringsinformationen. Under Fortran 77 krävdes att det var tillräckligt många element, under Fortran 90 läggs i stället vid behov blanka till på slutet. Under Fortran 90 får man vad som gällde under Fortran 77 genom att lägga till attributet `PAD = 'NO'`.

- Vid editering med `G`-format skriver Fortran 77 ut en flyttals-nolla i `E`-format, medan Fortran 90 skriver ut den i `F`-format. Anm. Sun Fortran 77 skrev bara 0.

En grupp High Performance Fortran Forum har utarbetat ett förslag för att hantera parallell databehandling i Fortran, i första hand i form av ett tillägg till Fortran 90. Syftet med detta projekt HPF är att erbjuda ett flyttbart (portabelt) språk som ger ett effektivt utnyttjande av olika parallella system. Projektet framlade ett slutligt förslag den 3 maj 1993, och syftar mot en de facto standard. Se vidare Appendix 9 i internetversionen och den utmärkta boken av Koelbel [23]. Något förenklat kan man säga att Fortran 90 effektivt klarar vektorprocessorer, medan HPF även klarar parallella processorer.

14.7.1 Borttaget ur Fortran 95

Följande egenskaper fanns i Fortran 90 men inte i standard Fortran 95. De flesta kompilatorer har kvar dem, men de bör trots detta undvikas.

- Styrvariabler i `DO`-slingan som är reella eller tal i dubbel precision

- Hopp till `END IF` från ett yttre block

- `PAUSE`

- `ASSIGN` med "assigned `GOTO`" och "assigned" `FORMAT`, dvs hela begreppet "satsnummervariabel" inklusive tilldelad hoppsats
- Hollerith-konstanter i `FORMAT`, dvs editering med `nHtext`

14.7.2 Borttaget ur Fortran 2003

Endast en egenskap i Fortran 95 togs bort i Fortran 2003, nämligen styrtecken för pappersutmatning på skrivare, se avsnitt 6.2, sid 63. I praktiken har detta ej fungerat sedan millenieskiftet. Det fungerade så att första tecknet i varje post betraktades som styrtecken, se vidare sektion 7.2 i internetversionen.

14.7.3 Föråldrade begrepp

I Fortran användes begreppet utdöende (eng. obsolescent), vilket innebär att vissa ålderdomliga konstruktioner kan komma att tas bort när Fortran ändras nästa gång. Dessa konstruktioner är

- Fix form av källkoden
- Styrd hoppsats
- Alternativen `CHARACTER*(length)` och `CHARACTER*length` till deklaration av textsträng `CHARACTER(len=length)`
- Datasatser blandade med exekverbara satser
- Satsfunktioner
- Antagen längd hos funktionsresultat
- Aritmetisk `IF`-sats
- Avsluta flera `DO`-slingor på samma sats
- Avsluta en `DO`-slinga på annat sätt än med `CONTINUE` eller `END DO`
- Alternativa återhopp

14.8 Skillnader mellan olika Fortran-standarder

I tabell 14.1 innebär till exempel -2 i position Hela Fortran 66 / Fortran 90 att två egenskaper försvunnit vid övergången från Fortran 66 till Fortran 90. Detta gäller både vid fix form och fri form hos källkoden.

I de fyra första raderna anger ett likhetstecken = att det gäller oförändrat, ett minus - att flera egenskaper saknas.

I de följande sju raderna anger ett plus + att egenskapen finns, ett minus - att den inte finns.

"Forts kol 6" innebär att fortsättningsrad markerars med något i kolumn 6 på nästa rad, medan "Forts med &" inebär att fortsättningsrad markerars med & sist på den aktuella raden. Blanka rader var inte tillåtna under Fortran 66, men är nu betraktade som blanka kommenarerer. Blanka är inte signifikanta

Egenskap	Fortran 66 Fix form	Fortran 77 Fix form	Fortran 90 Fix form	Fortran 90 Fri form	Fortran 95 Fri form
Hela Fortran 66	=	-2	-2	-2	-5
Hela Fortran 77	-	=	=	=	-5
Hela Fortran 90	-	-	=	=	-5
Hela Fortran 95	-	-	-14	-14	=
Forts kol 6	+	+	+	-	-
Forts med &	-	-	-	+	+
Blank kommentar	-	+	+	+	+
Blank signikant	-	-	-	+	+
Generiska funk.	-	+	+	+	+
Anv. generiska	-	-	+	+	+
REAL*8	-	-	-	-	-
Kommentarsymbol	C	C *	C * !	!	!
Filtyp i Unix	.f	.f	.f	.f90	.f90
Filtyp i DOS	.FOR	.FOR	.FOR	.F90	.F90

Tabell 14.1: **Skillnader mellan olika Fortran-standarder.**

under fix form (utom i textsträngar). "Anv. generiska" markerar att användaren kan skriva egna generiska funktioner från Fortran 90.

Fix form av källkod bör undvikas under Fortran 95!

`REAL*8` är en alternativ benämning på `DOUBLE PRECISION` som införts av IBM och användes även av andra leverantörer. Fler varianter i samma stil finns.

Som kommentarsymbol rekommenderas utropstecknet !, som accepteras även i flera implementationer av Fortran 77.

Under UNIX finns egentligen inget som heter filtyp, och dessa är inte specificerade i Fortran-standarderna utan är informella fabrikantstandarder.

Övningar

(14.1) Kompilera och kör något av dina mindre program med Fortran 90 utnyttjande fix form.

(14.2) Modifiera programmet genom att byta ut eventuella kommentarer inledda med `C` eller * mot ! och försök att utnyttja Fortran 90 fri form.

(14.3) Vad händer om följande lilla program körs under fix form respektive fri form?

```
LOGICAL L
L = .FALSE.
IF (L) THEN Z = 1.0
ELSE Y = Z ENDIF
END
```

14.9 Systemparametrar

Systemparametrar på flera datorer som uppfyller IEEE 754 [20] kan utnyttja två vanliga varianter för `KIND`-numret. De flesta utnyttjar antalet bytes som `KIND`-parameter, till exempel tillåtes värdena 1, 4, 8 och 16 för logiska variabler (med samma resultat som skönsvärdet 4), värdena 2, 4 och 8 för heltal, värdena 4, 8 och 16 för flyttal, samt värdena 4 och 8 för komplexa tal. Alternativet är att använda de naturliga talen 1, 2, 3, ... för stigande noggrannhet.

Man kan få fram dessa systemparametrar med de inbyggda funktionerna i Bilaga D.8.

14.9.1 Vanliga IEEE 754 system

Systemparametrarna för flera vanliga system ges nedan. Benämningen `quad` för fallet `KIND=16` är min egen! Likaså är benämningarna `int15, int31`, och `int63` mina egna.

```
LOGICAL              Default        byte        word
  KIND number =            4           1           4
LOGICAL              Default      double        quad
  KIND number =            4           8          16

INTEGER                int15          int31               int63
 KIND number =             2              4                   8
      digits =            15             31                  63
       radix =             2              2                   2
       range =             4              9                  18
        huge =         32767     2147483647 9223372036854775807
    bit_size =            16             32                  64

REAL                    single          double            quad
  KIND number =              4               8              16
       digits =             24              53             113
  maxexponent =            128            1024           16384
  minexponent =           -125           -1021          -16381
    precision =              6              15              33
        radix =              2               2               2
        range =             37             307            4931
      epsilon =  0.11920929E-06  0.22204460E-15  0.19259299E-33
         tiny =  0.11754944E-37  0.22250739-307 0.33621031-4931
         huge =  0.34028235E+39  0.17976931+309 0.11897315+4933

COMPLEX                 single         double
  KIND number =             4              8
    precision =             6             15
        range =            37            307
```

Tabell 14.2: **Systemparametrar på vanliga IEEE system.**

14.9.2 SGI 3000

Systemparametrarna för SGI 3000 under SGI Fortran 90 kompilator ges nedan. Beteckningarna `halfword, quad, int15, int31` och `int63` är mina egna.

```
INTEGER                    default
 KIND number =                   4
      digits =                  31
       radix =                   2
       range =                   9
        huge =          2147483647
    bit_size =                  32

INTEGER                  int15          int31                int63
 KIND number =               2              4                    8
      digits =              15             31                   63
       radix =               2              2                    2
       range =               4              9                   18
        huge =           32767     2147483647  9223372036854775807
    bit_size =              16             32                   64

LOGICAL                   byte    halfword        word     double
  KIND number =              1           2           4          8

REAL                     single          double            quad
  KIND number =               4               8              16
       digits =              24              53             107
  maxexponent =             128            1024            1023
  minexponent =            -125           -1021            -915
    precision =               6              15              31
        radix =               2               2               2
        range =              37             307             275
      epsilon =   0.11920929E-06  0.22204460E-15  0.12325952E-31
         tiny =   0.11754944E-37  0.22250739-307  0.18051944-275
         huge =   0.34028235E+39  0.17976931+309  0.89884657+308

COMPLEX                  single          double            quad
  KIND number =               4               8              16
    precision =               6              15              31
        range =              37             307             275
```

Tabell 14.3: **Systemparametrar på SGI 3000.**

OBS! Enligt SGI:s `man models` skall värdet av `minexponent` i quad-precision vara -967 i stället för det erhållna -915.

Det lilla omfånget för exponenten i `quad`-precision är förvånande! När jag använder metoderna från den elementära kursen i Numeriska Metoder, utnyttjar Fortran 90 men undviker de inbyggda funktionerna, får jag att parametrarna `maxexponent, minexponent` och `range` förefaller bli samma i quad-precision

som i dubbel precision. Motsvarande variabler får dock ej full quad-precision utanför de ovan givna parametrarna, varför tabellen ovan är korrekt.

SGI har en helt annan quad-precision än vad de flesta andra har, vilka senare har ett väldigt stort omfång. Orsaken är att hos SGI representeras quad-variabler som summan eller skillnaden av två tal i dubbel precision, normaliserade så att det mindre är $<= 0.5$ enheter i sista postionen av det större. De flesta andra leverantörer ansluter sig däremot till den föreslagna utvidgningen av IEEE 754 [20], som i förslaget från juli 2007 även innefattar fyrdubbel precision.

Kapitel 15

Felsökning

Inledning

Felsökning eller avlusning av program görs numera oftast i interaktiv miljö. De flesta språk innehåller kraftfulla hjälpmedel för sådan verksamhet. Interaktiva avlusningsmetoder anses, förmodligen med rätta, vara överlägsna andra avlusningsmetoder.

Vidare bör nämnas att benämningen DDT nedan inte har med något välkänt och förbjudet bekämpningsmedel att göra, utan står för Dynamic Debugging Technique. Begreppet avlusning (debugging) kommer från en av de första matematikmaskinerna MARK i USA, där vid ett tillfälle en skalbagge fastnat på ett av de elektromekaniska reläerna, varför maskinen inte längre fungerade. I den verkliga loggen står det dock "moth", vilket på svenska (från brittisk engelska) blir mal, mott eller nattfjäril. En bild på denna mal finns på nätet.

På amerikansk engelska heter skalbagge "bug", men översatt från brittisk engelska till svenska blir "bug" i stället vägglus. På brittisk engelska heter skalbagge i stället "beetle".

Tekniken DDT kan användas för "on-line" testning av bl a Fortran-program. Sedan användarprogrammet blivit assemblerat eller kompilerat kan det laddas med DDT. Genom att ge kommandon till DDT kan användaren sätta brytpunkter där exekveringen tillfälligtvis avbryts.

Detta ger möjlighet att testa ut program genom att enkelt inspektera godtyckliga programvariabler i brytpunkterna. Variabler och källkod kan även modifieras utan att omkompilering behöver göras.

Jag har lett arbetet på en bok om korrekt och tillförlitlig teknisk programvara som utkom augusti 2005 [11]. Denna kan vara till hjälp både vid felsökning och vid arbetet på att göra programmet mer tillförlitligt.

15.1 Dataflödesanalys

Genom att studera dataflödet kan man kontrollera programmen, dvs man kan finna en del möjliga fel. Titta först på följande avsnitt ur ett Fortran-program

```
X = A
X = B
```

Här gäller att variabeln `X` först tilldelas värdet av `A`, för att sedan omedelbart tilldelas värdet av `B`. Man får således aldrig någon nytta av att `X` först fick värdet på `A`. Det är därför sannolikt att det rätta programavsnittet i stället skall vara någonting som

```
X = A
Y = B
```

En annan anomali är

```
SUBROUTINE SUB(X,Y,Z)
Z = Y + W
```

Här är `W` odefinierat när det skall användas för att bilda `Z`. Menade författaren X i stället för `W`, eller `W` i stället för `X` i anropet, eller skall `W` vara i ett `COMMON`-block?

Som dessa exempel antyder kan vanliga programmeringsfel orsaka anomalier i dataflödet. Sådana fel är stavfel, namnförväxling, felaktig parameteröverföring i proceduranrop, saknade satser och så vidare.

Om man finner en anomali i dataflödet är detta dock inget säkert tecken på ett fel, bland annat kan det vara så att just den betraktade flödesvägen ej kan exekveras. Man kan i viss mån undvika sådana felgenom att utnyttja symbolisk exekvering, men sådan exekvering är mycket dyrbar.

Dataflödesanalysen, se till exempel Fosdick och Osterweil [16], ger förutom upptäckten av anomalier i flödet även värdefull information för dokumentationen av programmet. Den ger information om vilka variabler som får (returnerar) värden vid ett proceduranrop, eller som ger (tilldelar) värden till den anropade proceduren. Den identifierar kopplingen med `COMMON` och `EQUIVALENCE`. Den identifierar avsnitt av koden där vissa variabler ej används, liksom i vilken ordning procedurerna anropas.

När en sats i ett program exekveras påverkas förekommande variabler på flera sätt, av vilka vi vill särskilja använda (REFERENCE), tilldela (DEFINE) och glömma (UNDEFINE). När vid exekveringen av en sats värdet på en viss variabel erfordras säger vi att den användes i satsen. Om den i stället ges ett värde säger vi att den tilldelas. Slutligen förekommer det att en variabel tappar sitt värde vid (efter) exekveringen av en viss sats, då säger vi att den glömmes. Som förkortningar för dessa begrepp användes “r”, “d” och “u”.

När glömmer en variabel sitt värde? I ett blockstrukturerat språk (Algol) så inträffar detta varje gång man går ut från ett inre block, då försvinner alla lokala variabler. På motsvarande sätt gäller i Fortran att uthopp från ett underprogram gör att värdena på alla lokala variabler försvinner, med undantag av de fall då variabeln sparats med `SAVE` eller är tilldelad med en `DATA`-sats och ej förändrats.

I Fortran-satsen

```
A = B + C
```

användes `B` och `C`, medan `A` tilldelas, medan i

```
I = I + 1
```

variabeln `I` både användes och tilldelas. I satsen

```
A(I) = B + 1.0
```

användes `I` och `B`, medan `A(I)` tilldelas.

I Fortran 66 gäller att `DO`-slingans index glöms (blir odefinierat) när `DO`-slingan är tillfredsställd. Detta har dock ändrats till det bättre i och med senare versioner (Fortran 77 och Fortran 90). `DO`-slingans index har nämligen numera sitt nästa värde som slutvärde. Ett annat fall av att en variabel blir odefinierad är när den tilldelas ett odefinierat värde, till exempel i Fortran 66

```
X = 1.0
X = SIGN(X,0.0)
```

eller i Fortran 77

```
X = 1.0
X = AMOD(2.0,0.0)
```

Det är illustrativt att betrakta följande avsnitt ur ett Fortran 66 program.

```
      DO 10 K = 1, N
         X = X + A(K)
         IF (X .LE. 0.0 ) GOTO 20
         Y = Y + A(K)**2
10       CONTINUE
20    WRITE (*,*) K
```

Det intressanta att notera här är att `DO`-variabeln `K` ej är odefinierad när satsen `X = X + A(K)` exekveras efter `CONTINUE`, men att `K` är odefinierad då skrivsatsen exekveras efter `CONTINUE` men definierad då den exekveras efter `GOTO`-satsen. Till och med Fortran 66 gällde nämligen att `DO`-indexet vid normal utgång, dvs från en fullständigt genomlöpt `DO`-slinga, var odefinierat, medan från och med Fortran 77 det i stället har nästa värde. I ovanstående exempel har `K` således värdet `N+1` i fallet fullständigt genomlöpt slinga, dvs det fall som var odefinierat i Fortran 66.

Ett annat problem orsakas av vektorer och matriser. För ett enkelt fall som

```
B = A(1) + 1.0
```

är det inget problem att se att det är det första elementet i `A` som används, men i

```
B = A(K) + 1.0
```

är det mycket svårare, till och med omöjligt i det fall att `K` just har lästs in. För att klara av detta problem kan man helt enkelt göra en förenkling på så sätt att man jämställer alla element i en vektor eller matris, ändras ett så ändras alla vad gäller dataflödesanalysen.

Vi återgår nu till de tidigare införda förkortningarna “r”, “d”, och “u”, och använder följder av dessa bokstäver att betrakta hur en viss variabel genomlöper ett program. För variabeln A i

```
A = A + B
```

gäller "rd", medan för B gäller enbart "r". I det mer komplicerade fallet

```
A = B + C
B = A + D
A = A + 1.0
B = A + 2.0
GO TO 10
```

gäller för `A` "drrdr" och för `B` "rdd". Vi kallar dessa väguttryck (eng. path expressions). Det för `A` är logiskt, medan det för `B` är ologiskt, det erfordras naturligtvis en tilldelning före avsnittet, men allvarligare är att `B` beräknas (olika) två gånger, men det först beräknade används inte.

Om ett väguttryck innehåller någon av delföljderna "ur", "dd" eller "du" bör man studera programmet närmare. Det första innebär att en variabel används direkt efter att den blivit odefinierad, det andra innebär upprepade definitioner utan mellanliggande användning, och det tredje att efter att värdet har definierats så blir det odefinierat utan att någonsin ha använts. – Man kan införa konventionen att från början är alla variabler "u" om de ej har getts initialvärden genom en `DATA` eller `PARAMETER` sats.

Vi bör notera att den nämnda förenklingen av matrishanteringen är allvarlig i den meningen att den ofta förhindrar upptäckt av ett felaktigt dataflöde.

```
      DIMENSION R(100,2)
      READ(5,*) (R(I,1), I = 1, 100)
      CALL SQUARE(R)
      WRITE(6,20) (R(I,2), I = 1, 100)
20    FORMAT(1X,8F10.2)
      STOP
      END

      SUBROUTINE SQUARE(R)
      DIMENSION R(100,2)
      DO 10 I = 1, 100
         R(I,1) = R(I,2)**2
10    CONTINUE
      RETURN
      END
```

Här borde tilldelningssatsen i subrutinens `DO`-slinga i stället varit

```
         R(I,2) = R(I,1)**2
```

eftersom man uppenbarligen blandat ihop de båda kolumnerna, men förenklad dataflödesanalys ger ingen ledning.

15.2 Avlusare

Mycket avancerade felsökningssystem finns under UNIX utnyttjande modern fönsterteknik, ge kommandot `man dbx` eller `man xdbx`. Man använder det med kommandot `dbx a.out` (eller annat programnamn) och när det skrivet (dbx)

ger man sedan kommandot run för att starta exekveringen. När man är klar kommer man ur debuggern med kommandot quit. För ytterligare information ge kommandot man dbx.

Själv tycker jag inte om interaktiva avlusningshjälpmedel. En anledning är att jag använt Fortran på flera olika datorsystem (DEC ULTRIX, Sun, Silicon Graphics, Cray, IBM PC, IBM 7090, IBM 360, CD 6600, DEC-20, DEC VAX/VMS och Hewlett Packard). De flesta av dessa hade helt olika system för felsökning, en del av dem har i och för sig bra system för interaktiv avlusning, men det blir helt enkelt alltför jobbigt att lära sig alla avlusningssystem.

Jag föredrar därför att göra felsökning i satsmiljö, dels utnyttjande systemens möjligheter, dels genom att lägga in egna Fortran satser i koden på de ställen där det verkar gå fel. Ett viktigt exempel på inlagda hjälpmedel är de väljare som kan utnyttjas vid kompileringen.

Väljaren `-C` vid kompilering på UNIX med flera olika kompilatorer slår på indexkontroll. I vissa fall (konstanta men felaktiga index) fås då felutskriften redan vid kompileringen, annars vid exekveringen.

En del av de modernare kompilatorerna innehåller utmärkta hjälpmedel för att hitta fel. Jag har således upptäckt att NAG:s Fortran 95 kompilator kan upptäcka om två rutiner i ett användarprogram använder olika datatyp i en viss position i argumentlistan, vid anropet av en tredje rutin som befinner sig i ett programbibliotek. Eftersom den tredje rutinen är i ett programbibliotek kan inte systemet normalt avgöra vilken datatyp som är den rätta, utan den rapporterar bara att olika datatyp har utnyttjats vid de båda anropen.

15.3 Felsökningstips

Det finns fyra olika typer av fel, vilka jag vill nu i tur och ordning kommenterar.

15.3.1 Fel som visar sig vid kompileringen

Det är lämpligt att kompilera sitt program på ett relativt tidigt stadium, kanske rutin för rutin, för att få bort de rent formella felen. Dessa kan nämligen bli väldigt många vid stora helt nyskrivna program. Ett problem är att om en sats i början av en programenhet underkänns kan hela fortsättningen bli rappakaljaff-ör kompilatorn. Man bör därför alltid börja med att eliminera felen uppifrån. En användbar teknik är att göra en stomme för det fullständiga programmet men att till en början låta ingående subrutiner och funktioner vara tomma", dvs bara innehålla argumentlista, deklarationer av variablerna i denna, samt återhopp (med eventuellt funktionsvärde till exempel noll).

Ett sätt att få fler fel att visa sig redan vid kompileringen är att använda `IMPLICIT NONE` i varje programenhet eller att ge väljaren -u vid kompileringen under många Fortran 77 system. Då signaleras alla variabler som ej deklarerats. Som tidigare nämnts är det då ofta fråga om stavfel.

Ett annat sätt att få fler fel att visa sig redan vid kompileringen är att vid Fortran 90 använda det nya begreppet avsikt eller `INTENT` vid deklaration av alla formella variabler i funktioner och subrutiner. Då signaleras alla fall då variabler använts på fel sätt (ur denna synpunkt). De tre alternativen är som bekant `IN`, `INOUT` och `OUT`. Man måste dock vara noggrann vid specifikationen, och tänka sig för noga om variabeln är in eller ut eller både och i den aktuella rutinen. Valet

`INOUT` bör ej användas i onödan, eftersom det innebär att test-möjligheterna minskar (pga helgarderingen).

Om man glömmer `INTERFACE` eller har ett felaktigt sådant erhålles ofta felet "Segmentation error". Detta fel kan dock även erhållas vid ett i princip korrekt `INTERFACE`, men då man glömt att allokera något av de ingående fälten i den anropande programenheten.

15.3.2 Fel som visar sig vid exekveringen

Ett vanligt fel är att index på dimensionerade fält är fel. Man bör därför vid uttestningen kontrollera att alla använda index på dimensionerade variabler ligger inom sina tillåtna gränser. Det är ett mycket allvarligt men vanligt fel att de inte gör det.

Vid in- och utmatning på filer kan ett flertal fel uppstå. Det är viktigt att skilja på formaterad och oformaterad in- och utmatning, blandning av dessa två begrepp får ej ske på samma fil. Notera att list-styrd in- och utmatning är ett specialfall av formaterad, men att den följer speciella regler vad avser antalet tecken som skall matas in, vilket gör den praktiskt oanvändbar vid inmatning av text.

Vid användning av dynamisk minnesallokering erhålles ofta fel på grund av att nödvändigt minnesutrymme ej har allokerats.

Ett vanligt fel vid användning av programbibliotek är att detta har en annan flyttalsprecision än den man använder i det egna programmet. Exempelvis kan NAG-biblioteket vara i dubbel precision, varvid användning från program i enkel precision ger helt fel resultat och ibland obegripliga felutskrifter (om man inte konverterar vid anropet).

15.3.3 Fel som visar sig i resultatet

Det gäller att ha ett antal väl utarbetade och genomräknade testfall att kontrollera programmet med. Dessutom måste man skriva programmet på ett systematiskt sätt, så att det är lätt även för en människa att förstå vad som skall ske.

- Tänk på att lägga in robusthet i programmet, till exempel genom en felutskrift om något ej tillåtet värde av en viss variabel erhålles.
- Tänk på att lägga in utmatning av alla indata, så att du kan kolla att de kommit in rätt. Detta ger då även en värdefull dokumentation av beräkningen.

15.3.4 Fel som inte visar sig

- Komplettera testfallen ovan med extrema fall, av typ dimension negativ eller noll.
- Komplettera testfallen! Vad händer till exempel vid slumpvisa indata?
- Komplettera användarhandledningen (så att programmet förhoppningsvis åtminstone användes på det sätt du avsett).

- Kom ihåg att Fortran kompilatorn arbetar programenhet för programenhet, och därför normalt inte ser eventuella inkonsekvenser mellan huvudprogram, subrutin och funktion.

15.4 Underligheter i språket Fortran 95

En del av nedanstående underligheter är faktiskt till hjälp vid felsökningen.

- Under fri källkodsform tillåtes ej att kommentar markeras med `C` eller `*` i kolumn 1.

  Kommentarer inledda med `C` eller `*` skulle innebära ett brott mot just den fria formen. Det nya tecknet ! måste därför användas. Notera att detta tecken lyckligtvis även är tillåtet under den gamla (fixa) formen, samt i många implementationer av Fortran 77.

- Till skillnad från tidigare Fortran-standarder kräves numera i standarden att kompilatorn kan signalera om användaren avviker från det som standarden tillåter. Det kräves av en Fortran 95 kompilator att den kan signalera

  - Användning av syntax ej definierad i standarden;
  - Brott mot gällande syntaxregler;
  - Användning av ej tillgängliga slag (`KIND`);
  - Användning av föråldrade (utdöende) konstruktioner;
  - Användning av icke-Fortran tecken (till exempel svenska bokstäver) utanför textsträngskonstanter och kommentarer;
  - Brott mot giltighetsområdet för variabelnamn, namn på slingor och motsvarande, samt operatorer;
  - Anledningen till att ett program underkänts.

  Ovanstående innebär dock inget hinder för utvidgningar till Fortran 95.

- Kompilatorn utför ofta även en del dataflödesanalys, till exempel

  - Användning av värdet av en variabel som ej tilldelats något värde.

Kapitel 16

Optimering

Inledning

Optimering av Fortranprogram sker numera oftast helt automatiskt av kompilatorn. I detta kapitel skall vi gå igenom ett par speciella optimeringsegenskaper, som det kan vara bra att känna till.

En bra bok om Fortran-optimering i allmänhet är den av Metcalf [26], medan optimering främst på superdatorer behandlas av Levesque och Williamson [24]. En nyare bok på samma tema är den av Goedecker och Hoisie [17].

I sin enklaste form består optimering av att finna gemensamma deluttryck, beräkna dessa samt undvika att upprepa beräkningen, till exempel genom att ta ut beräkningen ur `DO`-slingor och spara resultatet för användning inuti.

Ett krav på optimering är att det matematiska resultatet ej ändras, däremot kan det numeriska resultatet komma att ändras på grund av att avrundningsfelet hos flyttalsberäkningen blir något annorlunda. I vissa fall skulle till och med allvarliga konsekvenser kunna uppträda genom att spill ("overflow") eller bottning ("underflow") uppkommer som en följd av ändrad beräkningsordning. Ett teoretiskt exempel är de matematiskt ekvivalenta uttrycken (a*b)/c och a*(b/c), men eftersom parenteser skall respekteras, så är dessa båda uttryck inte ekvivalenta i Fortran-mening. Det är dessutom föreskrivet att all beräkning (av samma prioritet) skall ske från vänster, utom i exponenter (efter **), jämför prioritetsreglerna i sektion 2.16, sid 26.

En annan teknik för optimering är att, eftersom funktions- och subrutinanrop är ganska kostsamma, flytta in respekive kod direkt i den anropande rutinen, så kallad "inlining". Detta rekommenderas endast om respektive subrutin är "liten" i lämplig mening. Man kan således skriva ett strukturerat program med många små funktioner och subrutiner och överlåta åt kompilatorn att framställa en effektiv version genom "inlining".

Mycket arbete läggs ner på att optimera slingor, speciellt då `DO`-slingor. Speciella problem (och möjligheter) uppstår vid vektormaskiner och/eller parallella maskiner. Att optimera slingor (även multipel-slingor) för vektormaskiner har nu blivit en etablerad teknik.

16.1 Slingor

Det vanligaste tricket för att snabba upp exekveringen av en multipel DO-slinga är att se till att indexen varieras i optimal ordning, normalt så att den inre slingan är den där även fältets index varierar snabbast.

Jag har testat ett enkelt exempel `slinga3.f90` på DEC Alfa utnyttjande Digitals Fortran med de olika optimeringsgraderna `O0` till `O5`.

```
PROGRAM SLINGA3
!       Undersökning av hastigheten vid access av stora fält
!       på olika sätt.
!       Rättad 1999-09-10
IMPLICIT NONE
INTEGER, PARAMETER                 :: DIM = 50
INTEGER                            :: Cykler_per_sekund
INTEGER                            :: Startcykel, Slutcykel
INTEGER                            :: I, J, K
REAL, DIMENSION(1:DIM,1:DIM,1:DIM) :: FAELT
REAL                               :: TID
!       Initiering av fältet
CALL RANDOM_SEED
CALL RANDOM_NUMBER(FAELT)
!       Initiering av tidmätning
CALL SYSTEM_CLOCK(COUNT_RATE=Cykler_per_sekund)

!  Metod 1 med explicit slinga, sista index varierar snabbast
OPEN(UNIT=1,FILE='slask1',STATUS='REPLACE',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)
DO I = 1, DIM
   DO J = 1, DIM
      DO K = 1, DIM
         FAELT(I,J,K) = FAELT(I,J,K)**2
      END DO
   END DO
END DO
CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 1 = ', TID, ' sekunder'
WRITE(1) SUM(FAELT)
CLOSE(1)

!  Metod 2 med explicit slinga, första index varierar snabbast
OPEN(UNIT=1,FILE='slask1',STATUS='OLD',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)
DO K = 1, DIM
   DO J = 1, DIM
      DO I = 1, DIM
         FAELT(I,J,K) = FAELT(I,J,K)**2
      END DO
   END DO
```

```
END DO
CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 2 = ', TID, ' sekunder'
WRITE(1) SUM(FAELT)
CLOSE(1)

!  Metod 3 med fält-operation
OPEN(UNIT=1,FILE='slask1',STATUS='OLD',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)
         FAELT = FAELT**2
CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 3 = ', TID, ' sekunder'
WRITE(1) SUM(FAELT)
CLOSE(1)

END PROGRAM SLINGA3
```

Resultat blev följande, efter några enkla redigeringar. Här är metod 1 en explict slinga där sista index varierar snabbast, metod 2 en explict slinga där första index varierar snabbast, samt metod 3 en fält-operation.

```
Script started on Fri Sep 10 13.14.33 1999
[Setting up eno.mai.liu.se (OSF1-V4.0-alpha) done.]

eno\[~/tana70\]> f90 -O0 slinga3.f90
 Tid i metod 1 =   0.0566 sekunder
 Tid i metod 2 =   0.0400 sekunder
 Tid i metod 3 =   0.0176 sekunder
eno[~/tana70]> f90 -O1 slinga3.f90
 Tid i metod 1 =   0.0410 sekunder
 Tid i metod 2 =   0.0137 sekunder
 Tid i metod 3 =   0.0107 sekunder
eno[~/tana70]> f90 -O2 slinga3.f90
 Tid i metod 1 =   0.0380 sekunder
 Tid i metod 2 =   0.0117 sekunder
 Tid i metod 3 =   0.0117 sekunder
eno[~/tana70]> f90 -O3 slinga3.f90
 Tid i metod 1 =   0.0341 sekunder
 Tid i metod 2 =   0.0078 sekunder
 Tid i metod 3 =   0.0088 sekunder
eno[~/tana70]> f90 -O4 slinga3.f90 (default)
 Tid i metod 1 =   0.0341 sekunder
 Tid i metod 2 =   0.0078 sekunder
 Tid i metod 3 =   0.0088 sekunder
eno[~/tana70]> f90 -O5 slinga3.f90
 Tid i metod 1 =   0.0078 sekunder
 Tid i metod 2 =   0.0078 sekunder
 Tid i metod 3 =   0.0079 sekunder
```

Vi ser således att i detta fall förefaller det ha stor betydelse om första (metod 2, bäst) eller sista (metod 1, sämst) index varierar snabbast, medan man vid en fältoperation får en betydande ytterligare uppsnabbning vid låg optimering. Utmatningen av summan av alla elementen sker i syfte att lura optimeringen, i annat fall kan en kraftfull optimering undvika att räkna ut något överhuvudtaget.

Utnyttjande NAG:s nya Fortran 95 kompilator på en Sun SPARC station 10 erhölls motsvarande resultat.

Jag har upprepat körningen på en Cray C90 och då använt `DIM = 100` i stället för `DIM = 50`.

```
Script started on Fri Sep 10 13:33:47 1999
sn4004 1 % f90 -V
Cray CF90 Version 2.0.1.0 07/19/99 15:29:15
sn4004 3 % f90 -O0 slinga3.f90
sn4004 4 % a.out
  Tid i metod 1 =  1.240 sekunder
  Tid i metod 2 =  1.293  sekunder
  Tid i metod 3 =  0.019  sekunder
sn4004 6 % f90 -O1 slinga3.f90
sn4004 7 % a.out
  Tid i metod 1 =  0.016  sekunder
  Tid i metod 2 =  0.004  sekunder
  Tid i metod 3 =  0.004  sekunder
sn4004 8 % f90 -O2 slinga3.f90     (default)
sn4004 9 % a.out
  Tid i metod 1 =  0.002  sekunder
  Tid i metod 2 =  0.002  sekunder
  Tid i metod 3 =  0.002  sekunder
sn4004 10 % f90 -O3 slinga3.f90    (inkl. autotasking)
sn4004 11 % a.out
  Tid i metod 1 =  0.050  sekunder
  Tid i metod 2 =  0.002  sekunder
  Tid i metod 3 =  0.002  sekunder
```

Vid låg optimering är metod 3 den klart bästa, vid måttlig optimering blir även metod 2 bra. Vid autotasking blir metod 1 extra ineffektiv jämfört med de andra metoderna.

Jag har nu valt ett annat exempel på Cray C90, nämligen där kompilatorn har problem med att vektorisera den slinga som ligger innerst. I exemplet `chanel1.f90` sker i den första trippelslingan en initiering och i den andra en beräkning, vilken även tidmätes med Crays funktion `SECOND()`. I den innersta beräkningsslingan finns ett beroende (elementet i position k beror av det i position k-1), vilket medför att denna ej kan vektoriseras utan vidare. Jag har valt att göra körningen med optimeringsgrad 1.

```
Program CHANEL1
parameter (n = 100)
dimension fz(n,n,n), cz(n)
```

```
      do 130 i = 1, n
        cz(i) = i**2
        do 120 j = 1, n
          do 110 k = 1, n
            fz(i,j,k) = i + j**2 + k**3
110       continue
120     continue
130   continue

      t1 = second()
      do 220 i = 1, n
        do 220 j = 1, n
          do 220 k = 2, n
220         fz(i,j,k) = fz(i,j,k) - cz(k)*fz(i,j,k-1)

      t2 = second()
      t  = t2 - t1
      write(6,400) t
400   format(' Normal beraekningsordning tar ',f9.6,' sek.')
      write(*,*) sum(fz)
      end program CHANEL1
```

När jag i stället kör `chanel2.f90` med samma optimeringsgrad sker exekveringen mycket snabbare. Detta beror på att den innersta slingan nu är över index j och det finns inte längre något beroende som förhindrar vektorisering. Med högre optimeringsgrad klarar kompilatorn av problemet på egen hand.

```
  program CHANEL2
      parameter (n = 100)
      dimension fz(n,n,n), cz(n)

      do 130 i = 1, n
        cz(i) = i**2
        do 120 j = 1, n
          do 110 k = 1, n
            fz(i,j,k) = i + j**2 + k**3
110       continue
120     continue
130   continue

      t1 = second()
      do 220 i = 1, n
          do 220 k = 2, n
        do 220 j = 1, n
220         fz(i,j,k) = fz(i,j,k) - cz(k)*fz(i,j,k-1)

      t2 = second()
      t  = t2 - t1
      write(6,400) t
400   format(' Optimerad beraekningsordning tar ',f9.6,' sek.')
      write(*,*) sum(fz)
```

```
       end program CHANEL2

Script started on Tue Jul 20 08:44:52 1999
sn4004 3 % f90 -V -O1 chanel1.f90
Cray CF90 Version 2.0.1.0 07/20/99 08:46:02
cf90: COMPILE TIME 1.588000 SECONDS
cf90: MAXIMUM FIELD LENGTH 238659 DECIMAL WORDS
cf90: 26 SOURCE LINES
cf90: 0 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
cf90: CODE: 97 WORDS, DATA: 180 WORDS
sn4004 4 % a.out
 Normal beraekningsordning tar  0.147152 sek.
 -2.320756189087945E+323
sn4004 5 % f90 -V -O1 chanel2.f90
Cray CF90 Version 2.0.1.0 07/20/99 08:46:38
sn4004 6 % a.out
 Optimerad beraekningsordning tar  0.008127 sek.
 -2.320756189087945E+323
sn4004 7 %
```

Exekveringen av en DO-slinga kan ibland snabbas upp genom att dela upp den ("unrolling") så att systemets parallellism kan utnyttjas maximalt. Detta gäller framför allt på superdatorer. En utmärkt genomgång av "unrolling" ges på sid 138-142 i boken av Levesque och Williamson [24].

16.2 Utmatning av hela fält

Vid utmatning av fält gäller att denna sker mycket snabbare om "hela fältet" kan matas ut direkt, utan att gå via en DO-slinga, jämför nedanstående exempel.

```
      REAL, DIMENSION (1:10000) :: A
! Metod 1, explicit DO-slinga
      DO I = 1, 10000
         WRITE(1) A(I)
      END DO
! Metod 2, implicit DO-slinga
      WRITE(1) (A(I), I = 1, 10000)
! Metod 3, implicit
      WRITE(1) A
```

Det visar sig här att metod 3 är mycket snabbare, även i ren CPU-tid. Jag har här ett fullständigt körbart exempel slinga1.f90 på detta.

```
PROGRAM SLINGA1
!       Undersökning av hastigheten vid skrivning av ett
!       stort fält  på olika sätt till skivminne.
```

```
IMPLICIT NONE
INTEGER, PARAMETER     :: DIM = 10000
INTEGER                :: Cykler_per_sekund
INTEGER                :: Startcykel, Slutcykel
INTEGER                :: I
REAL, DIMENSION(1:DIM) :: FAELT
REAL                   :: TID
!       Initiering av fältet
CALL RANDOM_SEED
CALL RANDOM_NUMBER(FAELT)
!       Initiering av tidmätning
CALL SYSTEM_CLOCK(COUNT_RATE=Cykler_per_sekund)

!       Metod 1 med explicit slinga
OPEN(UNIT=1,FILE='slask1',STATUS='REPLACE',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

DO I = 1, DIM
         WRITE(1) FAELT(I)
END DO

CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 1 = ', TID, ' sekunder'
CLOSE(1)

!       Metod 2 med implicit slinga
OPEN(UNIT=2,FILE='slask2',STATUS='REPLACE',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

     WRITE(2) (FAELT(I), I = 1, DIM)

CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 2 = ', TID, ' sekunder'
CLOSE(2)

!       Metod 3 helt implicit
OPEN(UNIT=3,FILE='slask3',STATUS='REPLACE',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

     WRITE(3) FAELT

CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 3 = ', TID, ' sekunder'
CLOSE(3)

END PROGRAM SLINGA1
```

Här följer utmatningen från programmet samt storleken på de erhållna resultatfilerna.

```
Script started on Fri Jul  9 11:25:57 1999
On a Sun SPARCstation 10 with the NAGWare f90 compiler
Version 2.2 (261)
%./a.out
  Tid i metod 1 =    0.240  sekunder
  Tid i metod 2 =    0.040  sekunder
  Tid i metod 3 =    0.020  sekunder
%/bin/ls -slFag
 128 -rw-r--r--  1 boein    nsc  120000 Jul  9 11:26 slask1
  40 -rw-r--r--  1 boein    nsc   40008 Jul  9 11:26 slask2
  40 -rw-r--r--  1 boein    nsc   40008 Jul  9 11:26 slask3
```

Vi ser här att metod 1 (explicit `DO`-slinga) både är långsammare och genererar en större fil.

Nu följer resultatet från DEC Alpha där jag dock ökat antalet element `DIM` från 10 000 till 100 000. På Sun blev det problem med en buffer vid för många element.

```
Script started on fre jul  9 12.24.50 1999
[Setting up eno.mai.liu.se (OSF1-V4.0-alpha) done.]
On a DEC Alpha station with the DIGITAL Fortran 90 compiler
Version 5.0-492

eno[~/tana70]> a.out
  Tid i metod 1 =   3.125   sekunder
  Tid i metod 2 =   0.694   sekunder
  Tid i metod 3 =   0.695   sekunder
eno[~/tana70]> /bin/ls -slFag
1264 -rw-r--r--   1 boein    num  1200000 Jul  9 12:26 slask1
 448 -rw-r--r--   1 boein    num   400008 Jul  9 12:26 slask2
 448 -rw-r--r--   1 boein    num   400008 Jul  9 12:26 slask3
```

Nu följer resultatet från CRAY C90 där jag även där har ökat antalet element `DIM` från 10 000 till 100 000. Filerna blir dessutom större eftersom precisionen på den maskinen är högre.

```
Script started on Fri Jul  9 12:59:06 1999
On a Cray C90 with Cray CF90 Version 2.0.1.0

sn4004 1 % a.out
  Tid i metod 1 =  2.457419  sekunder
  Tid i metod 2 =  0.002330  sekunder
  Tid i metod 3 =  0.002134  sekunder
sn4004 2 % ls -slFag
 3136 -rw-------   1 g97900   1605632 Jul  9 12:59 slask1
 1568 -rw-------   1 g97900    802816 Jul  9 12:59 slask2
 1568 -rw-------   1 g97900    802816 Jul  9 12:59 slask3
```

16.3 Formaterat eller oformaterat?

Det visar sig inte helt förvånande att oformaterad utmatning både är snabbare och normalt ger mindre filer än formaterad utmatning. Notera även att den oformaterade utmatningen inte ger någon noggrannhetsförlust. Jag har här ett fullständigt körbart exempel `slinga2.f90` på detta.

```
PROGRAM SLINGA2
!       Undersökning av hastigheten vid skrivning av ett
!       stort fält med olika format till skivminne.
IMPLICIT NONE
INTEGER, PARAMETER     :: DIM = 10000
INTEGER                :: Cykler_per_sekund
INTEGER                :: Startcykel, Slutcykel
INTEGER                :: I
REAL, DIMENSION(1:DIM) :: FAELT
REAL                   :: TID
!       Initiering av fältet
CALL RANDOM_SEED
CALL RANDOM_NUMBER(FAELT)
!       Initiering av tidmätning
CALL SYSTEM_CLOCK(COUNT_RATE=Cykler_per_sekund)

!       Metod 1 med ett tal per rad
OPEN(UNIT=1,FILE='slask1',STATUS='REPLACE',FORM='FORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

     WRITE(1,'(G13.6)') (FAELT(I), I = 1, DIM)

CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 1 = ', TID, ' sekunder'
CLOSE(1)

!       Metod 2 med lämpligt antal tal per rad
OPEN(UNIT=2,FILE='slask2',STATUS='REPLACE',FORM='FORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

     WRITE(2,'(10G13.6)') (FAELT(I), I = 1, DIM)

CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 2 = ', TID, ' sekunder'
CLOSE(2)

!       Metod 3 helt implicit
OPEN(UNIT=3,FILE='slask3',STATUS='REPLACE',FORM='UNFORMATTED')
CALL SYSTEM_CLOCK(COUNT=Startcykel)

     WRITE(3) (FAELT(I), I = 1, DIM)
```

```
CALL SYSTEM_CLOCK(COUNT=Slutcykel)
TID = REAL(Slutcykel - Startcykel)/REAL(Cykler_per_sekund)
WRITE(*,*) ' Tid i metod 3 = ', TID, ' sekunder'
CLOSE(3)

END PROGRAM SLINGA2
```

Här följer utmatningen från programmet samt storleken på de erhållna resultatfilerna.

```
Script started on Fri Jul  9 14:40:24 1999
On a Sun SPARCstation 10 with the NAGWare f90 compiler
Version 2.2 (261)
hugo 3 % a.out
  Tid i metod 1 =    2.240  sekunder (ett tal per rad)
  Tid i metod 2 =    1.980  sekunder (tio tal per rad)
  Tid i metod 3 =    0.020  sekunder (oformaterat)
hugo 4 % /bin/ls -slFag
 152 -rw-r--r--  1 boein    nsc  140000 Jul  9 14:40 slask1
 136 -rw-r--r--  1 boein    nsc  131000 Jul  9 14:40 slask2
  40 -rw-r--r--  1 boein    nsc   40008 Jul  9 14:40 slask3
```

Kapitel 17

Fortran 2003

Inledning

Fortran 2003 publicerades den 18 november 2004 som ISO/IEC 1539-1:2004 [21]. Jag rekommenderar böckerna av Metcalf, Reid och Cohen [27] och Chapman [5] samt den utmärkta artikeln av Reid om de nya egenskaperna [29]. Den fullständiga "The Fortran 2003 Handbook" [1] fungerar utmärkt som uppslagsbok.

Inte alla kompilatorer inkluderar alla nyheterna i Fortran 2003, se vidare [6].

17.1 Nya egenskaper

Det nya språket Fortran 2003 är nu Fortran 95 plus:

1. Utökade möjligheter för egna datatyper, inklusive med parametrar.
2. Utökat stöd för objekt-orienterad programmering.
3. Utökade möjligheter att hantera data.
4. Utökad in/utmaning, inklusive tillgång till felmeddelanden.
5. Pekare till procedurer.
6. Stöd för IEEE 754 [20] (riktad avrundning, division med noll, osv).
7. Samverkan med språket C.
8. Internationalisering, stöd för ISO 10646 (4 byte tecken) och val av decimalpunkt eller decimalkomma.
9. Samverkan med operativsystemet (kommandoradsargument, tillgång till systemparametrar).

Detta kapitel innehåller ett fåtal exempel som jag testat ut under Sun Fortran 95 8.2 2005/10/13, som innehåller en del Fortran 2003. Till skillnad från i övriga kapitel använder jag här normalt små bokstäver, vilket håller på att bli modernt även i Fortran. Det finns ett litet fel i Sun:s implementation, standardens

```
subroutine sub(f, b)  bind(c)
```

måste hos Sun skrivas med ett komma för att undvika kompileringsfel:

```
subroutine sub(f, b) , bind(c)
```

Motsvarande gäller naturligtvis även funktioner.

17.2 Val av decimalpunkt eller decimalkomma

För att byta från punkt till komma kan man i `OPEN` satsen ge `DECIMAL='COMMA'`, vilket då kommer att gälla alltid för den öppnade filen, eller man kan i `READ` eller `WRITE` satsen ge `DECIMAL='COMMA'`, vilket då gäller den satsen. Slutligen kan man inuti ett `FORMAT` ge `DC` för tillfälligt byte till komma.

Man kan naturligtvis byta tillbaks med `DECIMAL='POINT'` respektive `DP`. Om man använder liststyrd in- eller ut-matning tillsammans med decimalkomma ersättes komma som separator mellan värdena av semikolon.

Jag har nedan gjort ett mycket enkelt exempel, där man skriver ut talet π med ett valfritt antal decimaler (ett slags dynamiskt format) och först med decimalpunkt och sedan med decimalkomma.

```
PROGRAM DPDC
! Program för att visa decimalkomma och dynamiskt format

INTEGER           :: ANTAL
DOUBLE PRECISION  :: PI

CHARACTER(LEN=1)  :: FORMA
CHARACTER(LEN=30) :: FORM
CHARACTER(LEN=*), PARAMETER :: FORMP = "(DP,F15.", FORMC = "(DC,F15."

PI = 4.0D0*ATAN(1.0D0)
WRITE(*,'(A)',ADVANCE='NO') 'Ge antal decimaler (0-9): '
READ(*,*) ANTAL

FORMA = ACHAR((ANTAL+48)) ! 48 är ASCII-positionen för siffran noll

FORM = TRIM(FORMP) // FORMA // ')'

WRITE(*,FORM) PI ! skrives ut med decimalpunkt

FORM = TRIM(FORMC) // FORMA // ')'

WRITE(*,FORM) PI ! skrives ut med decimalkomma

END
```

17.3 Samverkan med språket C

Jag har vid en konferens i Kyoto 1995 hållit ett föredrag [10] om användning av rutiner i C från Fortran, och tvärtom. Det visade sig att man var tvungen

att göra detta något annorlunda under olika leverantörers system. Jag ger här dessa exempel under den nya portabla formalismen.

Vid denna moderna metod skall man i Fortran-rutinen deklarera samtliga aktuella variabler att vara av den typ som användes i C, se [27, tabell 14.1]. För att slippa göra detta i alla rutiner kan man kolla att C-typen och Fortran-typen stämmer överens, dvs. har samma `KIND` eller slag. För detta kan Du modifiera nedanstående exempel till att innefatta alla aktuella datatyper. Som framgår av detta program fick jag på Sun bara problem med logiska (boolska) variabler, vilket medför att de logiska variablern i Fortran (om de skall användas även i C) måste deklareras att inte ha standard-slaget (eller konverteras i en kommunikations-rutin).

```
Program CochF
! Program för att testa samverkan mellan C och Fortran
use, intrinsic :: iso_c_binding
IF (c_int == KIND(1)) THEN
   WRITE (*,*) 'Slaget av c_int =    ', c_int, &
 ' stämmer med INTEGER = ', KIND(1)
ELSE
   WRITE (*,*) 'Slaget av c_int =    ', c_int, &
 ' stämmer inte med INTEGER = ', KIND(1)
END IF

IF (c_float == KIND(1.0)) THEN
   WRITE (*,*) 'Slaget av c_float =  ', c_float, &
 ' stämmer med REAL = ', KIND(1.0)
ELSE
   WRITE (*,*) 'Slaget av c_float =  ', c_float, &
 ' stämmer inte med REAL = ', KIND(1.0)
END IF

IF (c_double == KIND(1.0D0)) THEN
   WRITE (*,*) 'Slaget av c_double = ', c_double, &
 ' stämmer med DOUBLE PRECISION = ', KIND(1.0D0)
ELSE
   WRITE (*,*) 'Slaget av c_double = ', c_double, &
 ' stämmer inte med DOUBLE PRECISION = ', KIND(1.0D0)
END IF

IF (c_bool == KIND(.TRUE.)) THEN
   WRITE (*,*) 'Slaget av c_bool =   ', c_bool, &
 ' stämmer med LOGICAL = ', KIND(.TRUE.)
ELSE
   WRITE (*,*) 'Slaget av c_bool =   ', c_bool, &
 ' stämmer inte med LOGICAL = ', KIND(.TRUE.)
END IF

IF (c_char == KIND('A')) THEN
   WRITE (*,*) 'Slaget av c_char =   ', c_char, &
 ' stämmer med CHARACTER = ', KIND('A')
```

```
ELSE
   WRITE (*,*) 'Slaget av c_char =   ', c_char, &
 ' stämmer inte med CHARACTER = ', KIND('A')
END IF

IF (c_float_complex == KIND((1.0, 1.0))) THEN
   WRITE (*,*) 'Slaget av c_float_complex =  ', c_float_complex, &
 ' stämmer med COMPLEX = ', KIND((1.0, 1.0))
ELSE
   WRITE (*,*) 'Slaget av c_float_complex =  ', c_float_complex, &
 ' stämmer inte med COMPLEX = ', KIND((1.0, 1.0))
END IF

END Program CochF
```

Körning av programmet på Sun gav

```
 Slaget av c_int =     4  stämmer med INTEGER =  4
 Slaget av c_float =   4  stämmer med REAL =  4
 Slaget av c_double =  8  stämmer med DOUBLE PRECISION =  8
 Slaget av c_bool =    1  stämmer inte med LOGICAL =  4
 Slaget av c_char =    1  stämmer med CHARACTER =  1
 Slaget av c_float_complex =   4  stämmer med COMPLEX =  4
```

17.3.1 Användning av en subrutin och en funktion skriven i Fortran från ett program i C

Vi beskriver här ett fall där först ett program skrivet i Fortran och sedan en rutin i C båda anropar en subrutin och en funktion i Fortran. Både huvudprogrammet och subrutinen anropar funktionen.

Huvudprogrammet i Fortran är i filen `f2sam.f` medan både subrutinen och funktionen är i filen `sam.f`.

```
% cat f2sam.f
        program f2sam
        external f
        integer f
        character*7 s
        integer b(3)
        call sam(f, b(2), s)
        write(6,10) b(2), f(real(b(2))), s
   10   format(i5,i5,10x,a7)
        stop
        end program f2sam
```

```
% cat sam.f90
        subroutine sam(f, b, s)
        external f
        character*7 s
        integer b, f
        x = 1.3
        s = 'Bo G E '
        b = f(x)
        end subroutine sam
        integer function f(x)
        f=3*x**3
        return
        end function f
```

Körning av dessa filer ger resultatet `6 648 Bo G E`.

Vi vill nu från C kunna anropa filen `sam.f` med subrutinen `sam` och funktionen `f` *utan* att ändra dessa två! Vi skriver därför först en kommunikations-fil `c_sam.f90` i Fortran

```
        subroutine c_sam(c_f, b, s) , bind(c)
        use, intrinsic :: iso_c_binding
        implicit none
        external c_f
        character(len=7)  :: s
        integer (c_int)   :: b, c_f
        real (c_float)    :: x
        integer, external :: f
        call sam(f,b,s)
        end subroutine c_sam

        integer (c_int) function c_f(x) , bind(c)
        use, intrinsic :: iso_c_binding
        implicit none
        real(c_float)     :: x
        integer, external :: f
        c_f=f(x)
        return
        end function c_f
```

och sedan ett huvudprogram i C som anropar dessa kommunikations-rutiner `c_sam` och `c_f`, vilka i sin tur anropar de "rena" Fortran-rutinerna `sam` och `f`.

```
% cat c2sam.c
#include <stdio.h>
#include <math.h>

/*     C anropar Fortran-rutinerna via kommunikations-rutiner */

       int c_f(float *);
       int c_sam(int (*c_f)(), int *, char *);
int main()
```

```
{
       char s[7];
       int b[3];
       float x;

       c_sam(c_f, &b[1], s);
       x = b[1];
       printf(" %d %d %s \n ", b[1], c_f(&x), s);
       return 0;
}
```

Körning med

```
f95 -c c_sam.f90 sam.f90
cc -c c2sam.c
cc c2sam.o c_sam.o sam.o
a.out
```

ger resultatet `6 648 Bo G E` som tidigare.

17.3.2 Användning från Fortran av en matris tilldelad värden i C

I detta fall har vi en matris där ett element tilldelas sitt värde i en C-rutin och som vi vill använda i ett Fortran-program. Vi måste acceptera att indexen i C och Fortran är i omvänd ordning, och att de i C måste minskas med ett jämfört med de i Fortran. C-rutinen `mlp4.c` ser ut så här

```
#include <stdio.h>
#include <math.h>

/*      Matris-hantering */

void
p(a,i,j)
int *i, *j, a[3][2];    /* Ordning 3 2 */
{
a[*j-1][*i-1] = 99;     /* Indexen minskade med 1 */
}
```

medan huvudprogrammet i Fortran `mlp3.f90` är

```
      program mlp3
      use, intrinsic :: iso_c_binding
      interface
        subroutine p(a,i,j) , bind(c)
        use iso_c_binding
          integer (c_int) :: a(2,3)
          integer (c_int) :: i, j
        end subroutine p
      end interface
```

```
      integer (c_int) ::  a(2,3) ! Ordning 2 3

      call p(a,1,3)
      write (6,10) a(1,3)
 10   format (1x,i9)
      stop
      end program mlp3
```

vilka körs med

```
cc  -c mlp4.c
f95 -c mlp3.f90
f95 mlp3.o mlp4.o
./a.out
```

med resultatet `99`.

17.3.3 Användning av Fortran `COMMON` i ett C-program

Liksom i det första examplet 17.3.1 vill vi här kunna använda en Fortran-rutin både från Fortran och C, varför vi åter behöver en kommunikationsrutin. `COMMON`-blocket tilldelas sina värden i rutinen `sam` i filen `mlp2.f`. I Fortran har vi således ett huvudprogram i `mlp0.f90`

```
program mlp0
implicit none
integer i
real r
common /namn/ i, r
call sam
write(*,*) i, r
end  program mlp0
```

som tillsammans med rutinen i filen `mlp2.f`

```
      subroutine sam()
      common /namn/ i, r
      i = 786
      r = 3.2
      return
      end subroutine sam
```

ger resultatet `786 3.2`.

Om vi nöjer oss med att kunna anropa rutinen `sam` **enbart** från C är det hela ganska enkelt, vi modifierar `sam` till filen `mlp2.f90`

```
      subroutine sam(), bind(c)
      use, intrinsic    :: iso_c_binding

         integer(c_int) :: i
         real(c_float)  :: r
         common /com/ i, r
```

```
         bind(c) :: /com/

      i = 786
      r = 3.2
      return
      end subroutine sam
```

och använder C-programmet i filen `mlp1.c`

```
#include <stdio.h>
#include <math.h>

/*      Använder COMMON Block från Fortran */

struct  {
   int i; float r;
} com;

main()
{
       sam();
       printf(" %d %f \n ",com.i,com.r);
}
```

och kör med

```
cc  -c mlp1.c
f95 -c mlp2.f90
cc     mlp1.o mlp2.o
a.out
```

och får rätt resultat.

Eftersom vi helst inte bör ändra rutinen `sam` i filen `mlp2.f` till den i `mlp2.f90` måste vi skapa en kommunikations-rutin `mlp2_c.f90`. Det visar sig vid mina tester att det inte fungerar att låta `COMMON`-blocket både användas av C och Fortran (förutom den Fortran-rutin i vilken den tilldelas sina värden). Jag använder därför i det följande de båda namnen `namn` och `com` på egenligen samma `COMMON`-block. Körningen helt i Fortran kan nu ske som tidigare.

Vid användning från C användes först och främst kommunikations-rutinen i filen `mlp2_c.f90`

```
      subroutine sam_c(), bind(c)
      use, intrinsic    :: iso_c_binding
          implicit none
          integer(c_int) :: i
          real(c_float)  :: r
          common  /com/  i, r
          bind(c)        :: /com/
          integer        :: i1
          real           :: r1
          common  /namn/ i1, r1
```

```
      call sam()
      i = i1
      r = r1
      return
      end subroutine sam_c
```

där jag alltså har två `COMMON`-block, ett med namnet `namn` riktat mot Fortran och ett med namnet `com` riktat mot C. Eftersom blocket tilldelas sina värden i Fortran måste erhållna värden kopieras över till variablerna i `COMMON`-blocket riktat mot C. Om C-programmet `mlp1.c` nu ändras till `mlp5.c` att anropa Fortran-rutinen `sam_c` (i stället för som tidigare Fortran-rutinen `sam`) sker körning med

```
f95 -c mlp2.f
f95 -c mlp2_c.f
cc  -c mlp5.c
f95 mlp5.o mlp2.o mlp2_c.o  (eller cc mlp5.o mlp2.o mlp2_c.o)
a.out
```

ger rätt resultat. Om man vill tilldela nya värden i C på `COMMON`-variablerna måste ytterligare en kommunikationsrutin skapas.

17.3.4 Andra metoder för att blanda Fortran och C

Tillsammans med två kamrater har jag skrivet en artikel [12] i detta ämne.

Det finns andra artiklar för att blanda Fortran och C, en sådan är [7] om att överföra fält.

17.4 Stöd för IEEE 754

Den nya Fortran-standarden innehåller kommandon för att kunna utnyttja egenskaperna i flyttalsaritmetiken IEEE 754 [20], bland annat när spill (overflow) eller bottning (underflow) uppträder.

Ett så enkelt problem som Pytagoras sats $x = \sqrt{a^2 + b^2}$ kan ge problem vid beräkningen. Om $a = 10^{30}$ erhålles spill vid normal enkel precision, och om $a = 10^{-30}$ erhålles bottning. I båda fallen kan det dock vara så att x har ett värde som ryms inom talområdet.

I följande exempel, modifierat från [27, sektion 11.10.4], sker beräkningen först på enklast möjliga sätt (vilket fungerar bra för "normala" tal), men om problem i form av spill eller bottning uppträder bytes metod. Det första specialfallet är att minst ett av argumenten är noll, det andra att storleksordningen skiljer sig så mycket att det ena kan försummas. I det tredje fallet är fortfarande storleksordningen stor, man skalar då om det första argumentet till att bli nära ett. Det andra skalas sedan om med samma skalning som det första, varefter resultatet skalas tillbaks (dvs. i motsatt riktning).

```
  real function pytagoras(x, y)
! Fortran 2003
  use, intrinsic :: ieee_arithmetic
  use, intrinsic :: ieee_features, only: &
                    ieee_underflow_flag
! Notera att ieee_overflow är alltid tillgänglig
```

```
! vid ieee_arithmetic!
  implicit none
  real :: x, y
  real :: scaled_x, scaled_y, scaled_result
  logical, dimension(2) :: flags
  type(ieee_flag_type), parameter, dimension(2) :: &
    out_of_range =(/ ieee_overflow, ieee_underflow /)
  intrinsic :: sqrt, abs, exponent, max, digits, scale
! Slå av avbrott vid spill
  call ieee_set_halting_mode(out_of_range, .false. )
! Försök med en snabb algoritm
  pytagoras = sqrt( x**2 + y**2 )
  call ieee_get_flag(out_of_range, flags)
! Om det blev problem, använd en långsam algoritm
! Kolla flaggorna
  if ( any(flags) ) then
     call ieee_set_flag(out_of_range, .false. )
!    Slå av flaggorna
     if ( x == 0.0 .or. y == 0.0 ) then
!    Specialfall 1
        pytagoras = abs(x) + abs(y)
        ! Minst en av x och y är noll i fallet ovan
     else if ( 2*abs(exponent(x) - exponent(y) ) > &
               digits(x) + 1 ) then
!    Specialfall 2
        pytagoras = max( abs(x), abs(y) )
        ! I detta fall kan vi ignorera en av x och y
     else ! Skala så att  abs(x) blir nära ett
!    Specialfall 3
        scaled_x = scale( x, -exponent(x))
        scaled_y = scale( y, -exponent(x))
        scaled_result = sqrt( scaled_x**2 + scaled_y**2 )
!       Skala nu tillbaks resultatet
        pytagoras = scale(scaled_result, exponent(x))
!       Vi kan fortfarande få ett ‘‘verkligt’’ spill
        call ieee_get_flag(out_of_range, flags)
!       Kolla flaggorna igen
        if ( (flags(1)) ) then
           write(*,*) 'Spill'
        end if
        if ( (flags(2)) ) then
           write(*,*) 'Bottning'
        end if
!       Slå av flaggorna
        call ieee_set_flag(out_of_range, .false. )
     end if
  end if
end function pytagoras
```

Kapitel 18

Fortran 2008

Inledning

Fortran 2008 publicerades som ISO/IEC 1539-1:2010 [21].

Inte alla kompilatorer inkluderar alla nyheterna i Fortran 2008, se vidare [6].

18.1 Nya egenskaper i Fortran 2008

1. Adderar "Co-arrays" för parallella datorer, vilket med till exempel deklarationen `X(1:16,1:16)[1:4,1:4]` fördelar data på olika processorer. Se vidare hemsidan för Co-Array Fortran `http://www.co-array.org/` och boken [28] (en bok som enbart behandlar Co-array). Dimensionen inom raka parenteser avser den så kallade co-dimensionen. Därför införs de inbyggda funktionerna `CO_LBOUND` och `CO_UBOUND` och kommandon `SYNC ALL, SYNC IMAGES, NOTIFY, QUERY, READY_TEAM`, och `SYNC MEMORY`. En beskrivning ges i [19] och [30]. Se även Donev [8].
2. Totala antalet dimensioner inklusive co-dimension är begränsad till 15.
3. Fler inbyggda funktioner, som Besselfunktioner, felfunktionen, gammafunktionen, ...
4. Tillåta realdelen och imaginärdelen i ett komplext tal att uppdateras oberoende.
5. Interna funktioner eller subrutiner som argument.
6. Start av externa program från Fortran-programmet.
7. Garantera förekomsten av en heltalstyp av högt talområde svarande mot `SELECTED_INT_KIND(18)`.
8. Ta bort `ENTRY`.

Kapitel 19

Fortran 2018

Inledning

Fortran 2018 blev officiellt i december 2018, kallades tidigare Fortran 2015.

Inte alla kompilatorer inkluderar alla nyheterna i Fortran 2018, se vidare [6].

19.1 Egenskaper i Fortran 2018

En beskrivning ges i i [31].

1. Adderar datatypen `BITS`. Dess `KIND`-nummer är specificerat som antalet bitar, och konkatenering med // är tillgängligt. Viktigt är att ett formellt argument av typen `BITS` kan anropas med argument av typ `REAL, INTEGER, LOGICAL` eller `COMPLEX` av samma storlek uttryckt i bitar, vilket kan utnyttjas vid arkivering då man inte känner den verkliga typen vid varje tillfälle. Några bitmanipuleringsfunktioner är hämtade från HPF, men många fler har tillkommet.
2. Intelligenta makron. Detta ger bland annat möjlighet att specificera en allmän typ av lista, för att senare med kommandot `EXPAND` föreskriva vilken datatyp listan skall ha.
3. Utvidgad `STOP`-sats.
4. Inbyggd funktion för att finna ett ledigt enhetsnummer.
5. Möjlighet att finna alla slag av logiska och text-typer.
6. Lokala namn på konstruktioner.
7. Tillåta en inbyggd funktion att ingå i en `TYPE`-sats.
8. Tillåta ASCII som standard sorteringsordning.
9. En inbyggd heltalsfunktion för 2-logaritmer.
10. En C *sizeof* procedur.
11. En inbyggd funktion för att ge avståndet i "ulps" mellan två flyttal.

12. Utökat stöd för HPF (för parallella datorer).
13. Möjlighet att dokumentera kompilatorns version.
14. Tydliggöra vad som menas med "approximation" av flyttal.
15. Utvidgad samverkan med språket C. Detta projekt har hög prioritet.
16. Ge skönsvärden för icke-närvarande valfria argument.
17. Skapa en parallell iteration.
18. Tillåta specifikation av fysikaliska enheter, d.v.s skapa en möjlighet att kontrollera att måttenheter stämmer överens, t. ex. cm och tum.
19. Skapa en överladdad version av `ATAN` med två argument att på sikt ersätta `ATAN2`.

Bilaga A

Sammanställning över Fortran 77 satser

Inledning

De satser som jag ej rekommenderar har markerats med #, i allvarliga fall till och med ##.

A.1 Deklarationer av programenheter

```
    PROGRAM     Huvudprogram
    FUNCTION    Funktion, FUNCTION kan föregås av
                någon av nedanstående deklarationer
                av variabler, utom IMPLICIT.
    SUBROUTINE  Subrutin
##  ENTRY       Extra ingång i underprogram.
#   BLOCK DATA  Gemensamma data, vilka delvis har
                initialvärden.
```

A.2 Deklarationer av variabler

```
#   IMPLICIT  IMPLICIT REAL (A-H,O-Z), INTEGER (I-N)
              är skönsvärde.
```

Satsen kan användas för att låta variabler med viss begynnelsebokstav få viss typ, jämför Bilaga C.5, sid 175.

```
    IMPLICIT NONE   Ej standard men bra, finns i Fortran 90,
                    ger "Pascal-konvention", dvs
                    att alla variabler måste deklareras.
                    På Sun och DEC kan motsvarande funktion
                    även erhållas med väljaren -u
                    i kompileringskommandot.
    INTEGER
    REAL
```

```
    DOUBLE PRECISION
    COMPLEX
    LOGICAL
    CHARACTER
```

A.3 Ytterligare specifikationer

```
    DIMENSION    Kan även ges direkt i typdeklarationen,
                 liksom i COMMON.
##  COMMON       Gemensam lagringsarea för variabler som
                 finns i flera programenheter.
##  EQUIVALENCE  Gemensam lagringsarea för flera
                 variabler i samma programenhet.
    PARAMETER    Gör om variabeln till en konstant med
                 ett visst värde.
    EXTERNAL     Talar om att ett variabelnamn svarar mot
                 en extern funktion eller underrutin.

    INTRINSIC    Talar om att ett variabelnamn svarar
                 mot en inbyggd funktion.
    SAVE         Sparar värden mellan återhopp från och
                 nytt inhopp i underprogram.
#   DATA         Sätter begynnelsevärden på variabler.
```

A.4 Exekverbara hoppsatser

```
    GOTO snr1
                 Hopp till satsnummer snr1 (Vanlig hoppsats).

#   GOTO (snr1, snr2, snr3), heltalsuttryck
                 Styrd hoppsats, om heltalsuttrycket
                 är 1, 2, eller 3 sker hopp till
                 satsnummer snr1, snr2, respektive snr3
                 (godtyckligt antal snr tillåtna)

##  GOTO satsnummervariabel, (snr1, snr2, snr3)
                 Tilldelad hoppsats, hopp sker till det
                 satsnummer som svarar mot
                 satsnummervariabeln.
                 (godtyckligt antal snr tillåtna).

##  GOTO satsnummervariabel
                 Tilldelad vanlig hoppsats (hybrid).

##  ASSIGN snr TO satsnummervariabel
                 En satsnummervariabel kan ej tilldelas
                 med en vanlig tilldelningssats
                 (heltalsvariabel = heltalsuttryck)
```

```
                utan endast med ASSIGN.
                Den kan sedan användas både i tilldelad
                hoppsats och i vanlig hoppsats, samt
                även i samband med FORMAT.

#   IF (numeriskt_uttryck) snr1, snr2, snr3
                Aritmetiskt hoppvillkor, hopp till satsnummer
                   snr1 om uttrycket är negativt
                   snr2 om uttrycket är noll
                   snr3 om uttrycket är positivt
                Kallas aritmetisk IF-sats.
```

A.5 Exekverbara andra satser

```
    IF (logiskt_uttryck) sats
                Villkorssats, om det logiska uttrycket
                är sant så utföres satsen, annars hoppar
                man direkt till nästa sats. Satsen får
                vara en vanlig tilldelningssats eller en
                vanlig hoppsats eller ett anrop av
                en underrutin. Kallas "logisk IF-sats".

    IF (logiskt_uttryck) THEN    ! Fullständig alternativsats.

                satsföljd1 ! Varianter utan ELSE-del,
                                 ! liksom kapslade med ELSE
        ELSE                     ! ersatt med
                                 ! ELSE IF (log_uttr) THEN
                satsföljd2 ! finns även.

        END IF

    CONTINUE    Fortsättningssats, gör inget.
                Rekommenderas för snygg avslutning av
                DO-slingan.

    STOP        Avslutningssats, avbryter exekveringen.

    END         Avslutningssats, avbryter kompileringen
                av programenheten, samt även
                exekveringen om man befinner sig där i
                huvudprogrammet. Om ett END påträffas
                under exekvering av ett underprogram sker
                i stället återhopp till den anropande
                programdelen.

#   PAUSE       Paussats, avbryter exekveringen
                tillfälligt (implementationsberoende).
```

```
    DO snr numerisk_variabel = var1, var2, var3
                DO-slinga
#               Flyttal tillåtes tyvärr som variabler
                i DO-slingor.
```

A.6 In/utmatningssatser

```
    OPEN        Öppna fil  ! Öppna en fil innan
                           ! programmet kan använda den.

    CLOSE       Stänga fil ! En fil som ej har
                           ! stängts kan vara
                           ! omöjlig att läsa!
    READ        Inläsning
    WRITE       Utmatning
#   PRINT       Tidigare utmatning till radskrivaren,
                numera synonym till WRITE
    INQUIRE     Förfrågan om fil-status
    REWIND      Återspola en fil till början
    BACKSPACE   Återspola en fil en post
    ENDFILE     Markera filslut
    FORMAT      Fortrans specialitet (se nedan)
```

A.7 Anropssatser

```
    CALL        Anropa en underrutin
    fnktn       En funktion anropas genom att bara ge
                funktionsnamnet
    RETURN      Återhopp från underprogram
                (underrutin eller funktion)
```

A.8 FORMAT-bokstäverna

```
                    Exempel

    Heltal      I   I5     5 positioner reserveras.
    Flyttal     F   F8.3   8 positioner, varav
                           3 för decimalerna.
                E   E14.6 14 positioner, varav
                           6 för decimalerna,
                           4 för exponenten,
                           1 för tecken,
                           1 för inledande nolla,
                           1 för decimalpunkt, samt
                           1 för inledande blank.
                D D28.12   Som E fast för dubbel-
                           precisionstal.
```

```
                G G14.6     Som F om talet ryms i
                            fältet, annars som E.

    Komplexa tal            Som par av flyttal.
    Logiska     L

    Textsträng A  A7   Exempel (7 bokstäver ryms i A7)

                       '    '   'Exempel'
#                       nH      7HExempel   Hollerith-konstant

    Positionering   Tn      n positioner från början
                    TLn     n positioner åt vänster
                    TRn     n positioner åt höger
                    nX      n positioner åt höger
#   Ej ny rad       $       Användes då man vill göra
                            inmatning i direkt
                            anslutning till utmatning
                            Ej standard! Ej Fortran 90!
    Avbryt          :       Om listan är slut avbryts
                            formatet här.
    Ny post         /       Normalt ny rad
    Binär           B       Ej standard men Fortran 90
    Oktal           O       Ej standard men Fortran 90
    Hexadecimal     Z       Ej standard men Fortran 90
```

A.8.1 Behandling av tecken och blanka

```
    Utmatning       SP      + skrives ut
                    SS      + skrives ej ut
                    S       Standard (normalt SS)

    Inmatning       BZ      Blanka tolkas som noll
                    BN      Blanka nonchaleras
```

`BN` är standard under ULTRIX, under hålkortseran var `BZ` standard. Jämför `BLANK='ZERO'` respektive `'NULL'` i `OPEN`-satsen, se avsnitt 7.1.1, sid 73.

OBS: `S, SP, SS, BN` och `BZ` gäller formatet ut, eller till dess en ny av samma slag uppträder.

A.8.2 Skalning av variabler

```
    Skalfaktor      kP
```

Inmatning:
Vid exponent, ingen verkan;
Utan exponent multipliceras talet med `10**(-k)` före tilldelning, dvs ändring av värdet.

Utmatning:
Vid exponent multipliceras mantissan med `10**k` och exponenten reduceras med `k`, dvs oförändrat värde;
Utan exponent multipliceras talet med `10**k` före utskrift, dvs ändring av värdet.

OBS: `kP` gäller formatet ut, eller till dess en ny uppträder. Skalning `kP` är bra på `E`-format, för då undvikes att den inledande siffran är noll och därigenom erhålls mer information. Skalning `kP` är katastrof på `F`-format, men den var dock intressant på hålkortstiden.

A.8.3 Tillägg i Fortran 90 beträffande `FORMAT`

Man kan nu ersätta `E` som markering för utmatning i exponentiell form med `ES` och får då en naturvetenskaplig (Scientific) form med utmatning av en siffra skild från noll före decimalpunkten. Om man i stället använder `EN` får man en teknisk (ENgineering) form med en till tre siffror före decimalpunkten och exponent delbar med tre. Om det utmatade värdet är noll erhålles samma utmatning från `ES` och `EN` som från `E`.

En annan utvidgning är att `FORMAT`en `I, B, O` och `Z` kan skrivas `Iw.m, Bw.m, Ow.m` och `Zw.m`, där `w` är den vanliga fältbredden, medan det valfria tillägget `m` indikerar ett minimalt antal siffror, med ledande nollor om så erfordras.

A.8.4 Tillägg i Fortran 95 beträffande minimalt fält

För att erhålla ett maximalt utnyttjande av antalet positioner vid utmatning är det nu möjligt att enbart ge det eventuella antalet decimaler och inte hela fältbredden vid formaten `B, F, I, O` och `Z`. Exempel är `I0` och `F0.6`. Det blir naturligtvis inte ett fält med noll tecken, utan ett med lagom många tecken.

Bilaga B

Sammanställning över de nya satserna i Fortran 95

Inledning

Denna bilaga innehåller en kort sammanfattning av de nytillkomna satserna i den nya standarden, och är *främst avsett för den som redan kan Fortran 77*. En systematisk genomgång av det fullständiga språket Fortran 95 sker i Bilaga C.

B.1 Källkoden

`INCLUDE` kan användas för att inkludera källkod från en extern fil. Konstruktionen är att på en rad finns `INCLUDE` samt en textsträng samt eventuellt avslutande kommentar, men satsnummer är ej tillåtet. Tolkningen är implementationsberoende, normalt tolkas textsträngen som namnet på den fil som skall inkluderas på det ställe i källkoden där satsen `INCLUDE` finns. Kapsling är tillåten (antalet nivåer är implementationsberoende), men däremot ej rekursion. Se vidare avsnitt 13.6, sid 117.

Om man vill använda Algol-principen att deklarera samtliga variabler underlättas detta av att kommandot `IMPLICIT NONE` slår av de implicita typ-reglerna. Notera att denna sats måste finnas först i varje programenhet, om den skall ha verkan i hela programmet.

Liksom i de flesta Algol-språk kan nu `END` kompletteras med namnet på rutinen eller funktionen, som `END PROGRAM ALPHA` och `END FUNCTION GAMMA`.

B.2 Alternativa representationer

Olikhetstecknen $<$ och $>$ kan användas enligt

```
<   .LT.        >   .GT.
<=  .LE.        >=  .GE.
==  .EQ.        /=  .NE.
```

Detta innebär att man inte behöver så många punktsymboler som tidigare. Det är dock inga fler punktsymboler som fått ny representation i och med Fortran

90/95. Notera att vid felaktig användning av exempelvis <= kan kompilatorn komma att klaga på .LE. i stället.

B.3 Specifikationer

Dessa kan nu skrivas på en rad utnyttjande dubelkolon ::

```
REAL, DIMENSION(2), PARAMETER :: a =  (/ 1.0, 7.0 /)

COMPLEX, DIMENSION(10) :: kurre
```

där variablerna a blir en konstant vektor med två element och flyttalsvärdena 1.0 respektive 7.0, medan kurre blir en komplex vektor med 10 (komplexa) element, vilka ej initierats.

Dubbel precision har kompletterats med en mer generell metod att ange önskad precision, nämligen parametern KIND för vilken precision som önskas, användbar på alla variabeltyper.

```
INTEGER, PARAMETER :: LP = SELECTED_REAL_KIND(20)
REAL (KIND = LP)   :: X, Y, Z
```

Ovanstående båda satser deklarerar således variablerna X, Y och Z att vara reella flyttalsvariabler med (minst) 20 decimala siffrors noggrannhet, av en datatyp som här kallas LP (står för lång precision).

B.4 Villkorssatser

Ett nytt kommando är

```
SELECT CASE (uttryck)
  CASE block-väljare
       block
  CASE block-väljare
       block
  CASE DEFAULT
       defaultblock
END SELECT
```

Typisk konstruktion:

```
SELECT CASE(3*i-j)  !  väljarvariabeln är 3*i-j
       CASE(0)      !  för väljarvariabeln noll
        ...         !  exekverbar kod
       CASE(2,4:7)  !  för väljarvariabeln 2,4,5,6,7
        ...         !  exekverbar kod
       CASE DEFAULT !  för andra värden på
                    !  väljarvariabeln
            ...     !  exekverbar kod
END SELECT
```

Om CASE DEFAULT saknas och inget av alternativen gäller så fortsätter exekveringen med nästa sats, utan felutskrift.

Annat exempel:

```
INTEGER FUNCTION SIGNUM(N)
SELECT CASE (N)
  CASE (:-1)
    SIGNUM = -1
  CASE (0)
    SIGNUM =  0
  CASE (1:)
    SIGNUM =  1
END SELECT
END
```

B.5 DO-slinga

Tre nya varianter införes. Den första ger i princip en oändlig slinga, som dock kan avslutas med till exempel en villkorlig hoppsats.

```
namn:   DO
          exekverbara satser
        END DO namn
```

Den gamla vanliga DO-slingan har nu följande utseende,

```
namn:   DO i = heltalsuttr_1, heltalsuttr_2, heltalsuttr_3
          exekverbara satser
        END DO namn
```

där i kallas kontrollvariabel eller styrvariabel, och där heltalsuttr_3 markerar ej nödvändig del. Slutligen finns även den nya DO WHILE-slingan

```
namn:   DO WHILE (logiskt_uttryck)
          exekverbara satser
        END DO namn
```

Namnet behöver ej anges men kan utnyttjas för kapslade slingor för att ange vilken som skall itereras på nytt med CYCLE eller avslutas med EXIT.

```
   S1:   DO
            IF ( X > Y ) THEN
               Z = X
               EXIT S1
            END IF
            CALL NEW(X)
         END DO S1

         N = 0
SLINGA1: DO I = 1, 10
            J = I
SLINGA2:    DO K = 1, 5
               L = K
               N = N + 1
            END DO SLINGA2
         END DO SLINGA1
```

I det senare fallet blir slutvärdena `I = 11, J = 10, K = 6, L = 5` och `N = 50` i enlighet med vad standarden föreskriver.

Namnsättning av slingorna är helt frivillig. Notera att denna typ av namn är begränsad till `DO`-slingor, `CASE` och `IF...THEN...ELSE...ENDIF`. Den gamla möjligheten med satsnummer (utan kolon) att hoppa till finns kvar även i fri form.

B.6 Programenheter

Rutiner kan anropas med nyckelordsargument och kan utnyttja underförstådda argument.

```
SUBROUTINE solve (a,b,n)
REAL, OPTIONAL, INTENT (IN) :: b
```

kan anropas med

```
CALL solve ( n = i, a = x)
```

där två av argumenten ges med nyckelord i stället för med position, och där det tredje är ett standardvärde. Om solve är en extern rutin fordras ett `INTERFACE` block i det anropande programmet. Se vidare avsnittet om **frivilliga argument** sid 49.

Rutiner kan specificeras att vara rekursiva

```
RECURSIVE FUNCTION fakultet(n) RESULT (fak)
```

men måste då ha ett särskilt `RESULT` namn för att föra tillbaks resultatet. Se vidare avsnittet om **rekursiva funktioner** sid 45.

B.7 Textsträngsvariabler

`CHARACTER` har utvidgats till att även innefatta en tom sträng

```
a = ''
```

och tilldelning av överlappande strängar

```
a(:5) = a(3:7)
```

samt ett antal nya inbyggda funktioner som `TRIM`, vilken tar bort avslutande blanka. Vidare så kan vi nu valfritt använda apostrof ' eller citattecken för att innesluta texten. Detta kan bland annat utnyttjas om man vill skriva en apostrof inuti en text, då omsluter man det hela med citattecken, och tvärtom om man vill ha ett citattecken inuti en text. Variabel längd på en textsträng finns bara under vissa restriktioner. Längden ges då antingen via en variabel vars värde bestäms vid anropet av programenheten, eller med en asterisk som kopplar textsträngen till ett verkligt argument av typ textsträng eller till en konstant textsträng. I ett tillägg till standarden [21] behandlas textsträngar med variabel längd.

B.8 Inmatning

Slutligen kommer `NAMELIST` in i standarden! Den satsen måste dock finnas bland specifikationerna. I exemplet nedan anger `list2` listans namn medan `a` och `x` är flyttalsvariabler och `i` en heltalsvariabel.

```
NAMELIST / list2 / a, i, x
...
READ(enhet, NML = list2)
```

önskar indata av formen (ej samtliga variabler behöver vara med, godtycklig ordning)

```
&list2 x = 4.3, a = 1.E20, i = -4 /
```

Jag anser dock att `NAMELIST` kom in för sent i standarden, det är inte längre något användbart begrepp vid nutidens interaktiva program. Det är ett gammalt IBM påfund från 1960-talet.

B.9 Vektor- och matrishantering

Detta är en av de viktigaste aspekterna på den nya standarden. Ett fält eller en "array" definieras att ha ett mönster eller en "shape" given av dess antal dimensioner (kallat rang eller "rank") och omfång eller "extent" för var och en av dessa. Två fält överensstämmer om de har samma mönster. Operationer sker normalt på element för element bas! Observera att rangen för ett fält är antalet dimensioner och inte har något att göra med den matematiska rangen för en matris!

```
REAL, DIMENSION(5,20)    :: x, y
REAL, DIMENSION(-2:2,20) :: z
:
z = 4.0*y*SQRT(x)
```

Vi kanske här vill skydda oss för negativa element i `x`. Detta sker genom

```
WHERE ( x >= 0.0 )
        z = 4.0*y*SQRT(x)
ELSEWHERE
        z = 0.0
END WHERE
```

Notera att `ELSEWHERE` måste vara i ett ord! Jämför även funktionen `SUM` som diskuteras sist i nästa avsnitt.

Med deklarationen

```
REAL, DIMENSION(-4:0,7) :: a
```

väljer `a(-3,:)` ut hela den andra raden, medan `a(0:-4:-2, 1:7:2)` väljer i omvänd ordning ut vartannat element i varannan kolumn.

Liksom variabler kan bilda fält, så kan även konstanter det.

```
REAL, DIMENSION(6)   :: B
REAL, DIMENSION(2,3) :: C

B = (/ 1, 1, 2, 3, 5, 8 /)
C = RESHAPE( B, (/ 2,3 /) )
```

där det första argumentet till den inbyggda funktionen `RESHAPE` ger värdena och det andra argumentet ger det nya mönstret. Ytterligare två argument finns som möjliga till denna funktion.

Några kommandon för verklig parallell databehandling finns ej, kommittén bedömde att ytterligare erfarenhet krävs före standardisering. Se därför Appendix 9 och 15 i internetversionen om HPF respektive Co-array Fortran (eller Fält-Fortran), samt Bilaga 19 här i boken om kommande standarder. Det mest använda systemet för parallell databehandling är MPI (Message Passing Interface), se [18], vilket har stöd för både Fortran 90 och C++.

B.10 Dynamisk minneshantering

Fortran 95 innehåller fyra olika sätt att göra dynamisk åtkomst av minne. Den första är att använda sig av pekare, och diskuteras i avsnitt 3.3.2, sid 37.

Det andra är att använda ett allokerbart fält, "allocatable array", dvs att med de båda satserna `ALLOCATE` och `DEALLOCATE` erhålla respektive återlämna lagringsutrymme för ett fält vars typ, rang och namn (och andra attribut) tidigare har deklarerats

```
REAL, DIMENSION(:), ALLOCATABLE :: x
...
ALLOCATE(x(n:m))  ! n och m är heltalsuttryck
                  ! Ge båda gränserna eller bara
                  ! den övre och då utan kolon!
...
x(j) = q
CALL sub(x)
...
DEALLOCATE (x)
...
```

Deallokering förekommer automatiskt (om ej attributet `SAVE` getts) när man i proceduren når `RETURN` eller `END` i samma procedur.

Den tredje varianten är ett automatiskt fält, "automatic array". Detta fanns nästan men inte riktigt i gamla Fortran, där `X` då måste vara med i argumentlistan.

```
SUBROUTINE sub (i, j, k)
REAL, DIMENSION (i, j, k) :: X
```

Dimensioneringen för `X` hämtas från heltalen i det anropande programmet.

Slutligen finns antaget mönster, "assumed-shape array", vars lagring definieras i den anropande proceduren, och för vilken endast typ, rang och namn ges.

```
SUBROUTINE sub (a)
REAL, DIMENSION(:, :, :) :: a
```

Här krävs ett explicit gränssnitt i den anropande programenheten för att överföra dimensioneringsinformationen. Detta skall se ut som följer:

```
INTERFACE
        SUBROUTINE SUB(A)
        REAL, DIMENSION (:,:,:) :: A
        END SUBROUTINE SUB
END INTERFACE
```

Om man glömmer `INTERFACE` eller har ett felaktigt sådant erhålles ofta felet "Segmentation error". Se vidare avsnitt 10.2, sid 97.

B.11 Inbyggda funktioner

Fortran 95 definierar omkring 100 inbyggda funktioner. Många av dessa används för fält, exempelvis för reducering (`SUM`), konstruktion (`SPREAD`) och manipulering (`TRANSPOSE`). Andra tillåter attribut i programmeringsmiljön att bestämmas, som största positiva flyttalet och heltalet, liksom tillgång till systemets klocka. En slumptalsgenerator ingår. Slutligen tillåter funktionen `TRANSFER` att innehållet i en viss fysisk area överföres till en annan area utan typkonvertering.

Vissa inbyggda funktioner finns för att bestämma de verkliga dimensioneringsgränserna

```
DO (i = LBOUND(a,1), UBOUND(a,1))
        DO (j = LBOUND(a,2), UBOUND(a,2))
                DO (k = LBOUND(a,3), UBOUND(a,3))
```

där `LBOUND` ger den lägre dimensioneringsgränsen för den specificerade dimensioneringen, och `UBOUND` den övre.

Summan av de positiva värdena av ett antal tal i ett fält kan skrivas

```
SUM ( X, MASK = X > 0.0 )
```

Samtliga inbyggda funktioner presenteras utförligt i Bilaga D.

B.12 Egna datatyper

Fortran har tidigare inte tillåtet användardefinierade datatyper. Detta blir nu möjligt med hjälp av satsen `TYPE`. Se avsnitt 9.7, sid 94.

För att kunna använda användardefinierade datatyper i exempelvis `COMMON` eller för att se till att två likadana datatyper betraktas som samma datatyp använder man kommandot `SEQUENCE`, i det senare fallet får dessutom ingen variabel vara deklarerad `PRIVATE`.

B.13 Moduler

Moduler är samlingar av data, typ-definitioner och procedur-definitioner, vilka ger en säkrare och allmängiltigare ersättning för `COMMON`. Se avsnitt 13.4, sid 111.

B.14 Datatypen "bit"

Datatypen "bit" finns ej, men väl bit-operationer på heltal enligt en tidigare standard MIL-STD-1753. Dessutom har binära, oktala och hexadecimala konstanter införts, liksom möjligheter att inkludera dessa storheter i in/utmatning genom införande av erforderliga format-bokstäver. I en `DATA` sats använder man då tilldelning enligt

```
B'01010101010101010101010101010101'
```

för binära,

```
O'01234567'
```

för oktala och

```
Z'ABCDEF'
```

för hexadecimala tal.

B.15 Pekare

Pekare har införts, men inte på det vanliga sättet som en egen datatyp utan som attribut till de andra datatyperna. En variabel med pekarattribut kan användas dels som en vanlig variabel, dels på ett antal nya sätt. Pekare i Fortran 95 är inte minnesadresser som i flera andra programspråk eller i vissa Fortran-varianter, utan är mer ett extra namn (alias). Se kapitel 11, sid 100.

B.16 Egna utvidgningar

Det nya språket innebär en möjlighet för användaren att utvidga det med egna begrepp, till exempel intervallaritmetik, rationell aritmetik eller dynamiska textsträngar. Genom att definiera nya datatyper och operatorer, samt överlagra operatorer och procedurer (så att man till exempel kan använda + som symbol även för addition av intervall och inte bara av vanliga tal) kan man bygga upp kompletterande paket (moduler), utan att gå omvägen via preprocessorer. Vi har redan sett ett antal utvidgningar för olika tillämpningar i form av moduler från programvaruleverantörer.

Bilaga C

Genomgång av hela språket Fortran 95

Inledning

Här följer den utlovade systematiska genomgången av hela språket Fortran 95. Den följer vad gäller ordningen Appendix D (syntaxregler) av den formella ISO-standarden [21].

C.1 Teckenkombinationer

Vissa grundbegrepp i Fortran skrives med mer än ett skrivtecken.

- Upphöjt till markeras med **
- Konkatenering, dvs addition av textsträngar, markeras med //
- Jämförelse av tal sker med nedanstående alternativa former

`<`	`.LT.`	`>`	`.GT.`
`<=`	`.LE.`	`>=`	`.GE.`
`==`	`.EQ.`	`/=`	`.NE.`

- Logisk negering markeras med `.NOT.`
- Logisk skärning markeras med `.AND.`
- Logisk union markeras med `.OR.`
- Logisk ekvivalens markeras med `.EQV.`
- Logisk icke-ekvivalens markeras med `.NEQV.`

C.2 Datatyper

- Icke-decimala numeriska konstanter (kan endast användas i `DATA`-satser).
  - Binär konstant `B` 'binära siffror' eller `B` "binära siffror"
    Binära siffror 01
  - Oktal konstant `O` 'oktala siffror' eller `O` "oktala siffror"
    Oktala siffror 01234567
  - Hexadecimal konstant `Z` 'hexadecimala siffror' eller `Z` "hexadecimala siffror"
    Hexadecimala siffror `0123456789ABCDEF`
- Exponentbokstav är `E` och/eller `D`
- Logiska konstanter är `.TRUE.` och `.FALSE.`
- Access-direktiv är `PUBLIC` och `PRIVATE`
- Privat-direktiv är `PRIVATE` och `SEQUENCE`
- Komponent-definition är typ-deklaration (eventuellt med `POINTER` eller `DIMENSION`).
- Kommandot för en användardefinierad datatyp är

```
TYPE, access-direktiv :: namn
        privat-direktiv
        komponent-definition
END TYPE namn
```

  Härvid gäller att om privat-direktivet `SEQUENCE` gäller så måste alla typer i komponent-definitionen också ha den egenskapen. Ett access-direktiv, eller en vanlig `PRIVATE` sats inne i definitionen, är tillåten endast om typ-definitionen befinner sig i en modul.

C.3 Deklarationer

- Deklarationer som kan ha `KIND`-parameter: `INTEGER, REAL, CHARACTER, COMPLEX` och `LOGICAL`
- Deklaration som dessutom kan ha längd-parameter: `CHARACTER`.
  Längd-parametern kan alternativt ges efter varje element.
- Deklaration som har typ-namn: `TYPE`
- Deklaration som inte har något av ovanstående tillägg: `DOUBLE PRECISION`.
- Attribut till ovanstående deklarationer: `PARAMETER PUBLIC` eller `PRIVATE, ALLOCATABLE, DIMENSION, EXTERNAL, EXTENT, INTENT, INTRINSIC, OPTIONAL, POINTER, SAVE` och `TARGET`.

  `DIMENSION`, måste kompletteras med rang och eventuellt även med omfång.

  `KIND`-parametern har formen (`KIND=heltalskonstant`)

`ALLOCATABLE` får ej ges för formella argument eller funktionsresultat.

Ett fält specificerat med `ALLOCATABLE` eller `POINTER` skall ha getts rang men ej omfång. Ett fält får inte ha båda dessa attribut.

Om `POINTER` har använts får inte något av `TARGET, INTENT, EXTERNAL` eller `INTRINSIC` användas.

Om `TARGET` har använts får inte något av `POINTER, EXTERNAL, INTRINSIC` eller `PARAMETER` användas.

`INTENT` och `OPTIONAL` får endast ges vid formella argument.

En variabel kan inte vara `PUBLIC` om motsvarande typ är `PRIVATE`

Attributet `SAVE` får inte ges för element i `COMMON`-block, för ett formellt argument, för en funktion eller för en subrutin.

En storhet får inte ges både `EXTERNAL` och `INTRINSIC`

`INTENT` ges som `INTENT(IN), INTENT(OUT)` eller `INTENT(INOUT)` och innebär att variabeln betecknas som in-variabel, ut-variabel respektive både och.

Explicit dimensionering ges med `(undre_gräns : övre_gräns)`, vid flera dimensioner så många gånger som erfordras, med komma mellan de olika paren, men högsta tillåtna antal dimensioner (rang) är 7. När endast rangen behöver ges utelämnas både undre gräns och övre gräns, dvs bara kolon ges för varje dimension. Om undre gräns är 1 kan undre gräns och : utelämnas. Den "sista" dimensionen kan i ett formellt fält ges med en * i stället för normal specifikation.

`INTENT` och `OPTIONAL` kan bara användas i en funktion, subrutin eller `INTERFACE` specifikation.

`SAVE` kan följas av variabelnamn och/eller `COMMON`-namn. Om så ej är fallet sparas allt som kan sparas, och några fler `SAVE` satser får då ej finnas i den aktuella programenheten.

Om attribut finns med måste deklarationen ges med dubbelkolon, annars kan dubbelkolon utelämnas. Jag rekommenderar att det får vara kvar.

C.4 Initieringar

- Initieringar kan ske med tilldelning direkt i deklarationen, med `DATA`-sats, samt permanent med `PARAMETER`-sats eller `PARAMETER`-attribut.

```
IMPLICIT NONE
REAL :: A = 1              ! Fortran 90
REAL B                     ! Fortran 77 och 90
DATA B / 6 /               ! Fortran 77 och 90
REAL C                     ! Fortran 77 och 90
PARAMETER ( C = 12 )       ! Fortran 77 och 90
REAL, PARAMETER :: D = 17  ! Fortran 90
```

- Varianten med `DATA`-sats är överlägsen vid initiering av fält, på grund av att man kan använda en upprepningsfaktor. Denna kan faktiskt vara en tidigare definierad heltalskonstant.

```
REAL, DIMENSION(4) :: A =  (/ 1, 1, 1, 1 /)
REAL, DIMENSION(4) :: B
DATA B = / 4*1 /
REAL, DIMENSION(4), PARAMETER :: D =  (/ 1, 1, 1, 1 /)
```

C.5 Implicita deklarationer

- Den gamla varianten med implicit typdeklaration, dvs alla variabler som börjar på någon av bokstäverna `IJKLMN` är heltal `INTEGER` och alla övriga är flyttal finns fortfarande kvar som skönsvärde. Jag rekommenderar att i stället alltid använda `IMPLICIT NONE`, för att tvinga fram kompileringsfelutskrift vid alla odeklarerade variabler. Dock finns tyvärr även möjligheten att deklarera att variabler som börjar på viss(a) bokstav skall ha viss typ,

  ```
  IMPLICIT LOGICAL (B-C, L)
  ```

  Satsen ovan innebär att alla variabler som börjar på någon av bokstäverna `B, C` eller `L` betraktas automatiskt som logiska variabler, om de inte explicit (dvs med hela namnet) deklarerats att vara av någon annan typ.

  Om `IMPLICIT NONE` utnyttjas måste den satsen vara först i programenheten (men efter `PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA` eller `MODULE`), och där får då inte finnas några andra `IMPLICIT` satser.

C.6 Speciella specifikationer

- Ett nytt kommando är `NAMELIST`, som tidigare var populärt vid inmatning på IBM:s Fortran-variant. Man utnyttjar därvid såväl gruppnamn inom snedstreck som vanliga variabler.

  ```
  NAMELIST / list_1 / a, i, x
  NAMELIST / list_2 / b, j, y, / list_3 / c, k, z
  ```

  Som variabelnamn får här inte användas ett formellt fältargument med icke-konstanta gränser, en textsträngsvariabel med variabel längd, en pekare, eller ett allokerbart fält.

  Trots att `NAMELIST` kom med i standarden i och med Fortran 90 är det snarare att betrakta som ett tillägg till Fortran 66 än en verklig beståndsdel av Fortran 90. Det har ej utvidgats till att klara alla nyheter hos datatyperna i Fortran 90.

  Om gruppnamnet getts attributet `PUBLIC` får inte något av variabelnamnen getts attributet `PRIVATE`.

- Ett gammalt kommando är `EQUIVALENCE`, som utnyttjas för att spara lagringsutrymme. Efter kommandot kommer en uppräkning av de variabler som skall dela lagringsutrymme, varje par ges inom parentes.

```
REAL, DIMENSION(10) :: C
INTEGER, DIMENSION(20) :: D
EQUIVALENCE (A, B), (C(8), D(3))
```

Som variabelnamn får här inte användas ett formellt argument, en pekare, ett allokerbart fält, en funktion, ett `ENTRY`-namn, ett `RESULT`-namn, namnet för en konstant, eller en struktur-komponent. Alla variabler som görs ekvivalenta måste vara i samma grupp av datatyper, nämligen i någon av

- Heltal, flyttal, dubbelprecisions-flyttal, komplexa tal samt logiska, samtliga av normalt slag (skönsvärdet).
- Textsträng av normalt slag (skönsvärdet).
- Användardefinierad typ (samma).
- Heltal, flyttal, dubbelprecisions-flyttal, komplexa tal samt logiska eller teststräng, samtliga av annat slag än skönsvärdet. I detta fall kan ekvivalens ske endast mellan variabler av samma typ och samma slag!

- Ett annat gammalt kommando är `COMMON`, som utnyttjas för att överföra information mellan olika programenheter. Efter kommandot kommer ett eventuellt block-namn inom snedstreck följt av en uppräkning av de variabler som skall delas.

```
REAL, DIMENSION(10) :: C
INTEGER, DIMENSION(20) :: D
COMMON A, B     ! Blankt COMMON
COMMON / / E, F ! Fortsättning av blankt COMMON
COMMON / VEKTORER / C, D
```

I motsats till vid `EQUIVALENCE` får i detta fall ett variabelnamn bara förekomma en gång.

Som variabelnamn får här inte användas ett formellt argument, ett allokerbart fält, en funktion, ett `ENTRY`-namn eller ett `RESULT`-namn. Om variabelnamnet svarar mot en användardefinierad typ, måste denna vara sekventiell, dvs ha attributet `SEQUENTIAL`. Eventuella fältgränser måste vara konstanta.

C.7 Allokeringssatser

- Med kommandot `ALLOCATE+indexALLOCATE` tilldelar man lagringsutrymme till ett fält, och med kommandot `DEALLOCATE` tar man bort det. Båda kommandona kan utnyttja ett status-attribut, `STAT=heltalsvariabel`. Denna variabel blir noll om operationen lyckas och får ett implementationsbeoroende positivt värde vid misslyckande. Endast fältvariabler som deklarerats som allokerbara eller som pekare kan allokeras. Allokering kan endast ske med exakt den rang (antal dimensioner) som tidigare deklarerats.

```
REAL, DIMENSION (:,:,:), ALLOCATABLE :: A
INTEGER :: I_ALLOKERING, I_AV_ALLOKERING
```

```
...
! Här antas värdena på heltalsvariablerna N och M kända
ALLOCATE ( A(1:10, 7:19, N:M), STAT = I_ALLOKERING)
...
DEALLOCATE (A, STAT = I_AV_ALLOKERING)
```

- Med kommandot `NULLIFY` tar man bort pekar-associering. Efter kommandot ges en parentes med aktuella pekare som skall av-associeras, med komma emellan.

C.8 Tilldelningar

- Normal tilldelning är `variabel = uttryck`. Här får som variabel ej användas ett fält med antagen dimension, dvs att den sista dimensioneringsgränsen endast getts som en * i deklarationen.
- Pekar-associering sker med `pekarobjekt => målobjekt`. Här får som målobjekt användas en variabel eller ett uttryck, men pekarobjektet måste ha deklarerats som pekare, dvs ha attributet `POINTER`. Ett målobjekt, som är en variabel, måste ha något av attributen `POINTER` eller `TARGET`, och det måste ha samma typ och mönster som pekarobjektet. Det får inte vara en fältsektion med vektor-index. Om målobjektet däremot är ett uttryck skall det ge en pekare som resultat.
- Villkorlig tilldelning erhålles med satsen `WHERE` eller konstruktionen `WHERE`. Den enkla satsen ser ut som följer,

  ```
  WHERE (mask_uttryck) tilldelning
  ```

  där maskuttrycket är ett logiskt villkor för ett fält och tilldelningssatsen innehåller fält med samma mönster. Tilldelning kommer då att ske enbart för element svarande mot sanna positioner i maskuttrycket. Konstruktionen däremot användes dels då olika tilldelningar skall ske vid sant och falskt, dels vid behov av flera tilldelningssatser.

  ```
  WHERE (mask_uttryck)
          tilldelningar
  ELSEWHERE
          tilldelningar
  END WHERE
  ```

C.9 Exekveringskontroll

Dessa omfattar `IF`-satser av olika slag, `CASE` och `DO`-slingor, olika varianter av `GO TO` samt den nya `FORALL`.

- Den klassiska enkla (logiska) `IF`-satsen ser ut som följer,

  ```
  IF (logiskt_uttryck) exekverbar_sats
  ```

där om det skalära logiska villkoret är sant så utföres den exekverbara satsen, annars fortsätter exekveringen med nästa sats. Den exekverbara satsen får inte vara en ny `IF`-sats, `CASE` eller `DO`-slinga. Den får inte heller vara en `END`-sats, men väl en `STOP`-sats eller normal `GO TO` sats.

- Den moderna `IF`-konstruktionen ser ut som följer,

```
namn:   IF (logiskt_uttryck) THEN
                exekverbara_satser
        ELSE IF (logiskt_uttryck) THEN namn
                exekverbara_satser
        ELSE namn
                exekverbara_satser
        END IF namn
```

  där om det första skalära logiska villkoret är sant så utföres de första exekverbara satserna, annars fortsätter exekveringen med eventuell `ELSE IF` eller `ELSE`-del. Om namnet (som är frivilligt att ange) användes på alla fyra ställena måste det vara lika, om det ej ges på den första platsen, dvs före kolon skall även kolon utelämnas, och då kan namnet ej heller ges vid något av de andra ställena. Normalt ger man bara namnet på det första och det sista stället (dvs. efter `END IF`).

  Det är tillåtet att sätta in "nästan vad som helst" bland de exekverbara satserna, till exempel en ny `IF`-konstruktion.

- Den gamla aritmetiska `IF`-satsen ser ut som följer,

```
IF (aritmetiskt_uttryck) sats_nr_1, sats_nr_2, sats_nr_3
```

  och medför hopp till det första satsnumret om det aritmetiska uttrycket är negativt, till det andra om det är noll, och till det tredje om det är positivt. Det aritmetiska uttrycket får naturligtvis inte vara komplext.

- Den nytillkomna `CASE`-konstruktionen ser ut som följer,

```
namn:   SELECT CASE (skalärt_uttryck)
        CASE (skalärt_värde_1) namn
                exekverbara satser
        CASE (skalärt_värde-2) namn
                exekverbara satser
        ...
        CASE (skalärt_värde-n) namn
                exekverbara satser
        CASE DEFAULT namn
                exekverbara_satser
        END SELECT namn
```

  där det skalära uttrycket kan vara heltal, textsträng eller logiskt. Värdena skall naturligtvis vara av samma typ, men de kan vara intervall (två värden skilda av kolon, i extremfall kan det ena av dessa värden utelämnas) och upprepningar (skilda av komma). De får däremot inte vara givna med

överlappningar, dvs ett värde får ej tillfredsställa mer än ett `CASE`. Om värdet inte tillfredsställer något av dem användes `CASE DEFAULT` om detta finnes, annars fortsätter exekveringen direkt med nästa sats. Om namnet (som är frivilligt att ange) användes på alla ställena måste det vara lika, om det ej ges på den första platsen, dvs före kolon skall även kolon utelämnas, och då kan namnet ej heller ges vid något av de andra ställena.

Värdena skall vara av samma typ och slag, men för textsträngar är olika längd tillåten. Varianten med kolon är inte tillåten vid logiska värden.

- Den välkända `DO`-slingan har kompletterats på ett flertal punkter, och finns nu i fyra varianter. Beträffande namnsättning gäller samma regler som ovan diskuteras för `IF` och `CASE`. Den första varianten är en evig slinga.

```
namn:     DO
                exekverbara satser
          END DO namn
```

Denna kan avbrytas med konventionella hoppsatser eller med den nya satsen `EXIT namn`, vilken ger ett uthopp ur aktuell slinga, eller slingan med angivet namn. En annan ny sats, `CYCLE namn`, ger på motsvarande sätt en ny iteration av slingan. Dessa båda satser kan naturligtvis även utnyttjas i de följande varianterna.

- Den klassiska `DO`-slingan har följande utseende.

```
      DO snr styrvariabel = start_värde, slut_värde, steg
            exekverbara satser
snr   sista_exekverbara_sats
```

Det bästa valet för `sista_exekverbara_sats` är satsen `CONTINUE`, en sats som inte utför något, men kan användes som platsmarkering för hopp hitan och ditan. Markeringen snr står för satsnummer om en till fem siffror. Kapslade slingor är tillåtna, och kan tillåtas sluta på samma sats. Numera rekommenderas dock att låta varje slinga sluta med sin egen `CONTINUE` sats eller ännu bättre med sin egen `END DO` enligt nedan. En modernare variant är nämligen

```
namn:  DO styrvariabel = start_värde, slut_värde, steg
          exekverbara satser
       END DO namn
```

Styrvariabeln kan i båda fallen vara heltal eller flyttal. Vi antar först att steget är utelämnat eller positivt. Om det är utelämnat användes steget ett. Det hela fungerar då så att först sätts styrvariabeln till startvärdet. Om detta värde är större än slutvärdet är det hela klart, och exekveringen fortsätter på nästa sats efter slingan. I annat fall utförs de exekverbara satserna med aktuellt värde på styrvariabeln, varefter denna uppräknas med steget och ny test sker.

- Den sista varianten är `DO WHILE` slingan med följande utseende.

```
namn:     DO WHILE (logiskt_uttryck)
                exekverbara satser
          END DO namn
```

- Följande bör noteras:
  - Efter genomgången slinga har styrvariabeln "nästa värde", dvs det första underkända.
  - Om en styrvariabel av typ flyttal användes kan avrundningsfelen spela in (flyttal bör ej användas i detta sammanhang i Fortran 95 och får ej användas i Fortran 2003).
  - Om steget är negativt blir det "likadant fast tvärtom".
  - Det är faktiskt tillåtet med ett "onödigt" komma efter `DO` om detta placeras efter eventuellt satsnummer och före styrvariabeln eller mellan `DO` och `WHILE`.
- En ny sats `FORALL` infördes i Fortran 95 för att till skillnad från en `DO`-slinga kunna utföras i godtycklig ordning (och därmed parallellt om den fysiska möjligheten finns). Ett par enkla exempel på `FORALL`-satser följer

```
FORALL (I = 1:N, J = 1:N) H(I,J) = 1.0/REAL(I+J-1)
FORALL (I = 1:N, J = 1:N, Y(I,J) .NE. 0.0) &
                X(I,J) = 1.0/Y(I,J)
FORALL (I = 1:N) A(I,I+1:N) = 4.0*ATAN(1.0)
```

  Den första av dessa definierar Hilbertmatrisen av ordning `N`, den andra inverterar elementen i en matris med undvikande av eventuella nollor. I den tredje tilldelas alla element över huvuddiagonalen i matrisen `A` ett approximativt värde på π.

  I dessa satser kan `FORALL` sägas vara en dubbelslinga som kan utföras i godtycklig ordning. Det allmänna utseendet är

```
FORALL ( v1 = l1:u1:s1, ... , vn = ln:un:sn, mask ) &
        a(e1, ... , em) = right_hand_side
```

  och beräknas enligt väl definierade regler, i princip beräknas alla index först.

  Dessutom finns en `FORALL`-konstruktion. Denna skall tolkas som om de ingående satserna i stället vore skrivna som `FORALL`-satser i samma ordning. Denna restriktion är väsentlig för att få ett entydigt beräkningsresultat.

- Förutom den klassiska hoppsatsen `GO TO snr` finns ett flertal varianter. Notera först att satsnummer i den gamla fixa formen skrives med siffror i de fem första positionerna av raden för en Fortransats. I den nya formen skriver man satsnumret till vänster och följt av minst en blank före den riktigasatsen.
- Den styrda hoppsatsen ser ut på följande sätt.

```
GOTO (snr1, snr2, ... , snrn) , skalärt_heltals_uttryck
```

Här kan ett godtyckligt antal satsnummer finnas inom parentesen. Kommat efter parentesen är faktiskt onödigt. Om det skalära uttrycket blir 1 sker hopp till det första satsnumret snr1, och så vidare.

- Den tilldelade hoppsatsen ser ut på följande sätt.

```
GOTO satsnummervariabel, (snr1, snr2, ... , snrn)
```

Här kan ett godtyckligt antal satsnummer finnas inom parentesen. Kommat före parentesen är faktiskt onödigt även här. Satsnummervariabel är däremot ett helt nytt begrepp. Det liknar heltal men är i själva verket programadresser. Om en satsnummervariabel skall tilldelas ett visst satsnummer kan detta tyvärr inte ske med en vanlig tilldelningssats utan måste ske med den speciella `ASSIGN`-satsen. Notera att parentesen med de olika möjliga satsnumren helt kan utelämnas, då måste naturligtvis även eventuellt komma före utelämnas. Man får då följande förenklade variant av den tilldelade hoppsatsen.

```
GOTO satsnummervariabel
```

Satsnummervariablerna försvann från standarden i och med Fortran 95.

- Den tidigare nämnda `ASSIGN`-satsen har följande utseende

```
ASSIGN satsnummer TO satsnummervariabel
```

Satsnummervariabler kan användas dels för dessa tilldelade hoppsatser, dels vid `FORMAT` i samband med in- och utmatning. Vid användning i aktuell programenhet av deklarationen `IMPLICIT NONE` skall satsnummervariablerna deklareras som `INTEGER`, trots att dom egentligen inte är det. Satsnummervariablerna försvann från standarden i och med Fortran 95.

- Fortsättningssatsen är mycket enkel och utför inget. Den kan användas för att ge lämpliga punkter i programmet att hoppa till, och för att avsluta `DO`-slingor. I det senare fallet rekommenderas att varje `DO`-slinga får sin egen fortsättningssats.

```
snr   CONTINUE
```

- Stoppsatsen är mycket enkel. Den stannar exekveringen med ett standardmeddelande, kompletterad med eventuell stopp-kod.

```
STOP stopp_kod
```

där stopp-koden kan vara en textsträng eller högst fem decimala siffror.

- Paussatsen är likaså mycket enkel. Den stannar exekveringen med ett standardmeddelande, kompletterad med eventuell paus-kod. På ett implementationsberoende sätt kan exekveringen återupptas.

```
PAUSE paus_kod
```

där paus-koden likaså kan vara en textsträng eller högst fem decimala siffror. Paussatsen fanns till och med Fortran 90, men är nu borttagen ur standarden.

C.10 Programenheter

De olika programenheterna är huvudprogram, moduler, `BLOCK DATA`, subrutiner och funktioner.

- Ett huvudprogram har följande uppbyggnad.

```
PROGRAM program_namn
        Specifikationer
        Satsfunktioner
        Exekverbar del
        Interna funktioner och subrutiner
END PROGRAM program_namn
```

  Den första satsen, `PROGRAM program_namn`, är helt frivillig, om den användes måste program-namnet anges. Den sista satsen kan vara `END, END PROGRAM` eller den fullständiga `END PROGRAM program_namn`. Naturligtvis måste Program-namnet i `END`-satsen vara samma som i `PROGRAM`-satsen.

  De exekverbara satserna i ett huvudprogram får inte innehålla någon av satserna `RETURN` eller `ENTRY`.

- En modul har följande uppbyggnad.

```
MODULE modul_namn
        Specifikationer
        Modul-subprogram
END MODULE modul_namn
```

  Motsvarande regler gäller för modul-namnet som för program-namnet. Modulen får inte innehålla satsfunktioner, `ENTRY` eller `FORMAT`.

  En modul användes med något av följande alternativ.

```
USE modul_namn
USE modul_namn, byt_namn_lista
USE modul_namn, ONLY: enbart_lista
```

  I det första fallet blir alla offentliga (`PUBLIC`) storheter i modulen tillgängliga. Här gäller att alternativet `byt_namn_lista` består av komponenter av följande utseende.

```
lokalt_namn => verkligt_namn_i_modulen
```

  Motsvarande regler gäller för `ONLY`, även där kan namnbyte ske. Samtliga storheter som anropas måste naturligtvis vara offentliga.

- En `BLOCK DATA` programenhet ser ut på följande sätt.

```
BLOCK DATA block_data_namn
        Specifikationer
END BLOCK DATA block_data_namn
```

Motsvarande regler gäller för block-data-namnet som för program-namnet, men något block data namn är ej nödvändigt att ange.

Specifikationerna får innefatta användning av moduler via `USE` satser, deklarationer av olika typer inklusive användning av `IMPLICIT, PARAMETER` satser, användardefinierade datatyper, samt följande olika specifikationer: `COMMON, DATA, DIMENSION, EQUIVALENCE, INTRINSIC, POINTER, SAVE` och `TARGET`.

Specifikationerna får däremot inte innefatta `ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE` eller `PUBLIC`. Inte heller får något gränssnitt `INTERFACE` ingå.

- Ett gränssnitt `INTERFACE` ser ut på följande sätt.

```
INTERFACE
        Specifikations_del
        Modul_procedur
END INTERFACE
```

I den första satsen, `INTERFACE`, kan tillägg ske med högst ett av

  - namnet på aktuell generisk funktion
  - `OPERATOR ( )`
  - `ASSIGNMENT (=)`

I fallet `OPERATOR` ovan skall inom parentesen finnas aktuell operator, till exempel +, -, * eller /.

`Specifikations_del` får inte innehålla någon av `ENTRY, DATA, FORMAT` och inte heller någon satsfunktion. Den får inte heller referera till en procedur som definieras i den programenhet i vilken gränssnittet ingår. I övrigt ser den ut som

```
FUNCTION funktions_namn(argument_lista)
         specifikationer
END FUNCTION funktions_namn
SUBROUTINE subrutin_namn(argument_lista)
           specifikationer
END FUNCTION subrutin_namn
```

Man kan förenklat säga att ovanstående erhålles då man tar bort ällt väsentligt", dvs den egentliga procedur-kroppen, ur funktionen eller subrutinen.

Modul-proceduren får endast finnas då `INTERFACE` satsen har något av tilläggen ovan, och den har då utseendet `MODULE PROCEDURE namn_lista`, där listan innehåller namn på procedurer som är tillgängliga.

- En deklaration av externa funktioner och/eller subrutiner ser ut på följande sätt.

```
EXTERNAL lista_över_externa_funktioner_och_subrutiner
```

Notera att om denna lista upptar ett namn på en i Fortran 90 inbyggd funktion eller subrutin blir motsvarande inbyggda ej längre tillgänglig. Detta kan användas om man av någon anledning vill skriva till exempel sin egen funktion för beräkning av sinus.

- En deklaration av inbyggda funktioner och/eller subrutiner ser ut på följande sätt.

```
INTRINSIC lista_över_inbyggda_funktioner_och_subrutiner
```

Notera att denna senare lista endast får upptaga namn på i Fortran 95 inbyggda funktioner och subrutiner, dvs de i Bilaga D.

- Ett funktionsanrop består helt enkelt av funktionsnamnet med de verkliga (aktuella) argumenten inom parentesen. Notera att vid funktionsanrop kan alternativa återhopp ej användas. Om funktionen saknar argument måste en tom parentes användas.

- Ett anrop av en subrutin består däremot av `CALL` följt av rutinnamnet med de verkliga (aktuella) argumenten inom parentesen. Notera att vid subrutinanrop kan alternativa återhopp användas, vilket sker med en asterisk * följt av satsnummer. Om subrutinen saknar argument utelämnas parentesen.

  I de båda fallen (funktion och subrutiner) kan det verkliga argumentet normalt ges antingen enbart på sin korrekta plats jämfört med det formella, eller utnyttjande det formella argumentet som nyckelord, dvs

```
formellt_argument = verkligt_argument
```

  Notera att det formella argument som skall ges som nyckelord måste vara det i gränssnittet, och inte nödvändigtvis det i den verkliga funktionen eller subrutinen. Detta innebär även att ett explicit `INTERFACE` är en förutsättning för att nyckelord skall kunna användas. Så fort som ett argument getts med nyckelord måste alla följande också ges med nyckelord. De som ges med nyckelord kan ges i godtycklig ordning.

  Som argument kan användas även namn på funktioner och subrutiner, men här gäller följande undantag: Man kan som argument inte använda satsfunktioner, interna funktioner, interna subrutiner eller det generiska namnet för en funktion eller subrutin. Observera här särskilt att de specifika och ej de generiska namnen på de inbyggda funktionerna måste användas i detta sammanhang.

- En funktion har nedanstående utseende.

```
FUNCTION prefix funktions_namn(argument_lista) &
                & RESULT (resultat)
                specifikationer
                satsfunktioner
                exekverbara_satser
                interna_funktioner_och_subrutiner
END FUNCTION funktions_namn
```

De båda delarna prefix och `RESULT (resultat)` är ej nödvändiga. För prefix gäller att det kan inledas med ordet `RECURSIVE` och följas av typdeklaration, eller tvärtom.

För `RESULT (resultat)` gäller att om det ges så skall ej någon typdeklaration ges i prefix och funktionsnamnet får ej typdeklareras på annat sätt i funktionen. Namnet `resultat` får ej vara samma som `funktions_namn`.

Ordet `FUNCTION` måste vara med vid `END` vad avser en intern funktion eller en modul-funktion. En intern funktion får inte innehålla en `ENTRY` sats eller i sin tur en intern funktion eller intern subrutin.

- En subrutin har nedanstående utseende.

```
SUBROUTINE subrutin_namn(argument_lista)
           specifikationer
           satsfunktioner
           exekverbara_satser
         interna_funktioner_och_subrutiner
END SUBROUTINE subrutin_namn
```

Första raden kan kompletteras med ordet `RECURSIVE` före rutin-namnet. Argumentlistan kan innehålla en asterisk * i stället för verkligt argument, nämligen när den ålderdomliga varianten med alternativa återhopp utnyttjas. Argumentlistan kan också helt utgå, nämligen om inga argument finns.

Ordet `SUBROUTINE` måste vara med vid `END` vad avser en intern subrutin eller en modul-subrutin. En intern subrutin får inte innehålla en `ENTRY` sats eller i sin tur en intern funktion eller intern subrutin.

- En `ENTRY` sats har nedanstående utseende.

```
ENTRY alternativ_namn(argument_lista) RESULT (resultat)
```

För `RESULT (resultat)` gäller att om det ges så skall funktionsnamnet ej typdeklareras på annat sätt i funktionen. Namnet `resultat` får ej vara samma som det på den alternativa ingången `alternativ_namn`. Inom programenheten får inte `alternativ_namn` förekomma som formellt argument eller i en annan `ENTRY` sats eller i en `EXTERNAL` eller `INTRINSIC` sats.

- En `RETURN` sats ger återhopp till anropande rutin. Vid alternativa återhopp användes i stället `RETURN heltalsuttryck`, och ger då återhopp till den alternativa återhoppspunkten placerad i motsvarande position i anropslistan. `RETURN` är ej tillåtet i huvudprogrammet. Om det saknas ger `END` normal återhoppsoperation i funktion och subrutin.

- En satsfunktion placeras mellan specifikationer och vanliga exekverbara satser i en programenhet och har följande mycket enkla utseende. Satsfunktionerna är enbart tillgängliga i denna programenhet, och ej från andra programenheter.

```
satsfunktion(argument_lista) = skalärt_uttryck
```

Det skalära uttrycket får använda sig av tidigare (ovanför) definierade satsfunktioner. Rekursiv användning (att den anropar sig själv) är ej tillåten

- En `CONTAINS` sats inleder den eventuella del av en programenhet som innehåller interna funktioner och subrutiner, dvs sådana som enbart är tillgängliga i denna, och ej från andra programenheter. Dessa interna funktioner och subrutiner skrives som vanligt, enda undantaget är att variabler från programenheten (delen över `CONTAINS`) är tillgängliga utan att deklareras. Ett eventuellt `IMPLICIT NONE` skall ej upprepas.

C.11 In- och utmatning

En fullständig diskussion gavs i kapitel 6 och upprepas därför ej här. Se även diskussionen av `FORMAT`-bokstäverna i Bilaga A.8, sid 161.

Bilaga D

Inbyggda funktioner och subrutiner i Fortran 95

Inledning

Det finns i Fortran 95 ett mycket stort antal inbyggda funktioner, och sex inbyggda subrutiner.

Detta avsnitt bygger på sektion 13 av standarden ISO [21], vilken innehåller en mer formell framställning. Jag följer standardens gruppering av de olika funktionerna, men ger förklaringen direkt i anslutning till listan. För en mer uttömmande framställning hänvisar jag till Metcalf och Reid [27].

När en parameter nedan är frivillig så har den getts med små bokstäver, gemena. När en argumentlista innehåller flera argument kan funktionen anropas antingen med positionsrelaterade argument eller med nyckelord, eller en kombination. Nyckelord måste användas om något föregående argument utelämnas, samt när man gett ett argument med nyckelord måste även alla följande ges med nyckelord. Nyckelord är de namn som getts med gemena nedan. Jag har däremot försvenskat några av de obligatoriska parametrarna, som ändrat `ARRAY` till `FAELT`. Jag har inte alltid gett alla naturliga inskränkningar i variablers tillåtna värden, som att rangen ej får vara negativ.

D.1 Funktion som undersöker om ett visst argument finns i anropslistan

```
PRESENT(A)
```

Returnerar `.TRUE.` om argumentet `A` är med i anropslistan, `.FALSE.` annars. Användningen illustreras i avsnittet om **Frivilliga argument**, sid 49, och avsnittet 10.7, sid 98.

D.2 Numeriska funktioner

Följande numeriska funktioner kommer från Fortran 77:

```
ABS, AIMAG, AINT, ANINT, CMPLX, CONJG, DBLE, DIM, DPROD,
INT, MAX, MIN, MOD, NINT, REAL, SIGN
```

Vissa av dessa kan nu kompletteras med en slags-parameter som i fallet `AINT(A,kind)`, nämligen `AINT, ANINT, CMPLX, INT, NINT` och `REAL`, samt från Fortran 95 även de nya `CEILING` och `FLOOR`.

Dessutom har `CEILING, FLOOR` och `MODULO` tillkommit. Endast den senare är besvärlig att förklara, vilket bäst sker med ett exempel från ISO [21]. Både `MOD` och `MODULO` ger naturligtvis resten vid heltalsdivision, men de behandlar tecknen i täljare och nämnare olika. Resultatet för funktionerna `MOD` och `MODULO` är odefinierat om det andra argumentet är noll.

```
MOD(8,5)   ger  3 MODULO(8,5)   ger  3
MOD(-8,5)  ger -3 MODULO(-8,5)  ger  2
MOD(8,-5)  ger  3 MODULO(8,-5)  ger -2
MOD(-8,-5) ger -3 MODULO(-8,-5) ger -3
```

En historisk relikt för de numeriska funktionerna är att de i Fortran 66 var tvingade att ha olika namn för olika precisioner, och dessa explicita namn är fortfarande de enda som kan användas då funktionerna användes som argument. En fullständig tabell över samtliga numeriska funktioner ges därför. De markerade med * får ej användas som argument. En del funktioner har två specifika namn, som `INT` och `IFIX`, vilka är helt likvärda. Jag använder nedan beteckningen `C` för komplext flyttal, `D` för flyttal i dubbel precision, `I` för heltal och `R` för flyttal i enkel precision. I samtliga fall förutsättes standard-slaget (`KIND`).

Uppgift	Generiskt namn		Specifikt namn	Data-typ Arg	Res
Konvertering till heltal	INT		-	I	I
		*	INT	R	I
		*	IFIX	R	I
		*	IDINT	D	I
(av realdelen)			-	C	I
Konvertering till flyttal	REAL	*	REAL	I	R
		*	FLOAT	I	R
			-	R	R
		*	SNGL	D	R
(realdelen)			-	C	R
Konvertering till dubbel precision (realdelen)	DBLE		-	I	D
			-	R	D
			-	D	D
			-	C	D
Konvertering till komplex	CMPLX		-	I (2I)	C
			-	R (2R)	C
			-	D (2D)	C
			-	C	C

Trunkering	AINT		AINT	R	R
			DINT	D	D
Avrundning	ANINT		ANINT	R	R
			DNINT	D	D
	NINT		NINT	R	I
			IDNINT	D	I
Absolut värde	ABS		IABS	I	I
			ABS	R	R
			DABS	D	D
			CABS	C	R
Rest	MOD		MOD	2I	I
			AMOD	2R	R
			DMOD	2D	D
	MODULO		-	2I	I
			-	2R	R
			-	2D	D
Golv	FLOOR		-	I	I
			-	R	R
			-	D	D
Tak	CEILING		-	I	I
			-	R	R
			-	D	D
Tecken-överföring	SIGN		ISIGN	2I	I
			SIGN	2R	R
			DSIGN	2D	D
Positiv differens	DIM		IDIM	2I	I
			DIM	2R	R
			DDIM	2D	D
Inre produkt	-		DPROD	R	D
Maximum	MAX	*	MAXO	I	I
		*	AMAX1	R	R
		*	DMAX1	D	D
	-	*	AMAXO	I	R
	-	*	MAX1	R	I
Minimum	MIN	*	MINO	I	I
		*	AMIN1	R	R
		*	DMIN1	D	D
	-	*	AMINO	I	R
	-	*	MIN1	R	I

Imaginärdel	-	AIMAG	C	R
Konjugering	-	CONJG	C	C

De flesta av de ovanstående funktionernas uppgift framgår ur namnet, men förutom ovan diskuterade `MOD` och `MODULO` behöver ytterligare några diskuteras. Trunkering sker mot noll, dvs `INT(-3.7)` blir -3, medan naturligtvis avrundning sker korrekt, dvs `NINT(-3.7)` blir -4. De båda nya funktionerna `FLOOR` och `CEILING` trunkerar mot - oändligheten respektive mot + oändligheten.

Funktionen `CMPLX` kan ha ett eller två argument, vid två argument måste dessa vara av samma typ och ej av typ `COMPLEX`.

Funktionen `MOD(X,Y)` beräknar `X - INT(X/Y)*Y`.

Teckenöverföringsfunktionen `SIGN(X,Y)` tar tecknet hos det andra argumentet och sätter det på det första, dvs `ABS(X)` om `Y >= 0` och `-ABS(X)` om `Y < 0`. I Fortran 95 infördes att teckenfunktionen använder tecknet hos det andra argumentet även om det första argumentet är noll.

IEEE-aritmetiken [20] har för flyttal två olika representationer av noll, en för plus noll och en annan för minus noll. Tidigare tvingade Fortran att noll alltid skulle vara lika, vilket gjorde att man inte kunde utnyttja tecknet hos noll. Nu gäller att för processorer som kan åtskilja dem att de skall behandlas som identiska

- Vid alla jämförelseoperationer
- Som in-argument till alla inbyggda funktioner och subrutiner utom för `SIGN`
- Som det skalära uttrycket i den (föråldrade) aritmetiska IF-satsen

För att särskilja de båda fallen måste därför funktionen `SIGN` användas. Den har därför generaliserats i Fortran 95 så att tecknet hos det andra argumentet beaktas även då värdet är noll (flyttal).

Positiv differens `DIM` är en funktion jag aldrig använt, men `DIM(X,Y)` ger `X-Y` om detta är positivt och annars noll.

Inre produkt `DPROD` är däremot en mycket användbar funktion som ger produkten av två enkelprecisionstal som ett dubbelprecisionstal. Den är både snabb och noggrann.

De båda funktionerna `MAX` och `MIN` är unika i det att de har ett godtyckligt antal argument (av samma typ), dock minst två.

D.3 Matematiska funktioner

De trigonometriska funktionerna och deras inverser arbetar naturligtvis i radianer och ej i grader. Följande matematiska funktioner kommer från Fortran 77:

```
ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP,
LOG, LOG10, SIN, SINH, SQRT, TAN, TANH
```

En historisk relikt för de matematiska funktionerna är att dessa i Fortran 66 var tvingade att ha olika namn för olika precisioner, och dessa explicita namn är fortfarande de enda som kan användas då funktionerna användes som argument. En fullständig tabell över samtliga matematiska funktioner ges därför.

Uppgift	Generiskt namn	Specifikt namn	Data-typ Arg	Res
Kvadratrot	SQRT	SQRT	R	R
		DSQRT	D	D
		CSQRT	C	C
Exponential-funktionen	EXP	EXP	R	R
		DEXP	D	D
		CEXP	C	C
Naturlig-logaritm	LOG	ALOG	R	R
		DLOG	D	D
		CLOG	C	C
10-logaritmen	LOG10	ALOG10	R	R
		DLOG10	D	D
Sinus	SIN	SIN	R	R
		DSIN	D	D
		CSIN	C	C
Cosinus	COS	COS	R	R
		DCOS	D	D
		CCOS	C	C
Tangent	TAN	TAN	R	R
		DTAN	D	D
Arcus-sinus	ASIN	ASIN	R	R
		DASIN	D	D
Arcus-cosinus	ACOS	ACOS	R	R
		DACOS	D	D
Arcus-tangenten	ATAN	ATAN	R	R
		DATAN	D	D
	ATAN2	ATAN2	2R	R
		DATAN2	2D	D
Hyperbolisk Sinus	SINH	SINH	R	R
		DSINH	D	D
Hyperbolisk Cosinus	COSH	COSH	R	R
		DCOSH	D	D
Hyperbolisk Tangent	TANH	TANH	R	R
		DTANH	D	D

De flesta av de ovanstående funktionernas uppgift framgår ur namnet, men

några behöver diskuteras. Notera först att ingen av dessa är definierad för heltalsargument, varför man inte kan beräkna exempelvis kvadratroten ur heltalet 4 med `SQRT(4)`, men väl med `NINT(SQRT(REAL(4)))`. Notera vidare att samtliga komplexa funktioner returnerar principalvärdet.

Kvadratroten ger ett reellt resultat för argument i enkel eller dubbel precision, och ett komplext resultat för ett komplext argument. Så ger `SQRT(-1.0)` en felutskrift (oftast redan vid kompileringen), medan man får den komplexa (rent imaginära) roten med följande satser.

```
COMPLEX, PARAMETER      :: MINUS_ETT = -1.0
COMPLEX                 :: Z
Z = SQRT(MINUS_ETT)
```

Argumentet för de "vanliga" logaritmerna måste vara positivt, medan argument för `CLOG` måste vara skilt från noll. Värdet kommer att ligga i intervallet $(-\pi;\pi]$, det blir π endast om argumentet har negativ realdel och imaginärdel noll.

Argumentet för `ASIN` måste till beloppet vara högst ett. Funktionens värde kommer att ligga i intervallet $[-\pi/2;\pi/2]$.

Argumentet för `ACOS` måste till beloppet vara högst ett. Funktionens värde kommer att ligga i intervallet $[0;\pi]$.

Funktionen `ATAN` kommer att få ett värde i intervallet $[-\pi/2;\pi/2]$.

Funktionen `ATAN2(Y,X)` $= \arctan(y/x)$ kommer att få ett värde i intervallet $(-\pi;\pi]$. Om `Y` är positivt blir resultatet positivt. Om `Y` är noll blir resultatet noll om `X` är positivt och π om `X` är negativt. Om `Y` är negativt blir resultatet negativt. Om `X` är noll blir resultatet $\pm\pi/2$. De båda argumenten `X` och `Y` får ej samtidigt vara noll. Syftet med funktionen är att undvika division med noll vid vinkelberäkningen.

En naturlig begränsning för de matematiska funktionerna beror på den begränsade noggrannheten, vilket innebär att exponentialfunktionen `EXP` ger spill eller bottning redan vid "måttliga" värden på argumentet, och att de trigonometriska funktionerna får mycket låg noggrannhet vid till beloppet stora argument. Dessa begränsningar är implementationsberoende, och bör vara angivna i respektive maskins manual.

D.4 Textsträngsfunktioner

Nedanstående funktioner utför operationer från och till textsträngar. Notera att `ACHAR` arbetar med det standardiserade ASCII-teckensättet, medan `CHAR` arbetar med den aktuella representationen i datorn.

```
ACHAR(I)              Ger det ASCII-tecken som har nr I
ADJUSTL(STRING)       Vänsterjusterar
ADJUSTR(STRING)       Högerjusterar
CHAR(I,kind)          Ger det tecken som har nr I
IACHAR(C)             Ger ASCII-numret för tecknet C
ICHAR(C)              Ger numret för tecknet C

INDEX(STRING,SUBSTRING,back)
                      Ger startpositionen för en
```

```
                    delsträng inom en sträng.
                    Om BACK är sant erhålles den
                    sista förekomsten, annars
                    den första.

LEN_TRIM(STRING)    Ger längden av en sträng utan
                    de eventuellt avslutande blanka.

LGE(STRING_A,STRING_B)
LGT(STRING_A,STRING_B)
LLE(STRING_A,STRING_B)
LLT(STRING_A,STRING_B)
```

Ovanstående fyra rutiner jämför två strängar utgående från sortering enligt ASCII. Om en sträng är kortare än den andra adderas blanka på slutet av den kortare. Om någon sträng innehåller ett tecken utanför ASCII blir resultatet implementationsberoende.

```
REPEAT(STRING,NCOPIES)  Konkatenerar en sträng NCOPIES
                        gånger med sig själv.
SCAN(STRING,SET,back)   Ger positionen för den första
                        förekomsten av något tecken i SET
                        i strängen STRING. Om BACK är sant
                        i stället sista!
TRIM(STRING)            Ger strängen STRING utan avslutande
                        blanka.
VERIFY(STRING,SET,back) Ger positionen i STRING för det
                        första tecken i STRING som ej
                        finns i SET. Om BACK är sant
                        i stället sista!
                        Resultatet är 0 om alla finns!
```

D.5 Textsträngsfunktion för förfrågan

```
LEN(STRING)
```

Denna rutin ger längden av en textsträng. Strängen behöver ej ha tilldelats något värde.

D.6 Slagsfunktioner

```
KIND(X)
SELECTED_INT_KIND(R)
SELECTED_REAL_KIND(p,r)
```

Den första ger slaget av aktuellt argument, som kan vara av typ `INTEGER, REAL, COMPLEX, LOGICAL` eller `CHARACTER`. Argumentet `X` behöver ej ha tilldelats något värde.

Den andra ger ett heltalsslag med önskat antal siffror, och den tredje ett flyttalsslag med numerisk precision minst `P` siffror och en (fiktiv) decimal exponent mellan `-R` och `+R`. Parametrarna `P` och `R` måste vara skalära heltal. Minst en av `P` och `R` måste ges.

Resultatet av `SELECTED_INT_KIND` är heltal från 0 och uppåt, om önskat slag ej finns returneras i stället -1. Om flera implementerade typer satisfierar villkoret användes den med minsta decimala exponent, om flera sådana den med minsta slag (nummer). Resultatet av `SELECTED_REAL_KIND` är heltal från 0 och uppåt, om önskat slag ej finns returneras i stället -1 om precisionen ej finns, -2 om exponenten inte finns, och -3 om ingen finns. Om flera implementerade typer satisfierar villkoren användes den med minsta decimala precision, om flera sådana den med minsta slag (nummer).

Exempel på användning av `KIND` finns i avsnitt 9.3, sid 89, och exempel på olika `KIND`-nummer ges i avsnitt 14.9, sid 125. Exempel på slag i olika implementationer ges även i internetversionens Appendix 6 för NAG och Appendix 7 för Cray.

D.7 Logisk funktion

```
LOGICAL(L,kind)
```

Omvandlar mellan olika slag av logiska variabler.

D.8 Numeriska förfrågningsfunktioner

Dessa funktioner arbetar i en viss modell av heltals- respektive flyttalsaritmetik, se ISO [21, avsnitt 13.7.1]. Funktionerna ger egenskaper hos tal av samma slag som variabeln X, den kan vara `REAL` och i vissa fall `INTEGER`. Funktioner som ger egenskaper hos just argumentet `X` finns under punkt 12, Flyttalsmanipuleringsfunktioner.

```
DIGITS(X)       Antalet signifikanta bitar
EPSILON(X)      Det minsta positiva talet som
                adderat till 1 ger ett tal
                som är större än 1
HUGE(X)         Det största positiva talet
MAXEXPONENT(X)  Den största exponenten
MINEXPONENT(X)  Den minsta exponenten
PRECISION(X)    Decimala precisionen
RADIX(X)        Basen i modellen
RANGE(X)        Decimal exponent
TINY(X)         Det minsta positiva talet
```

D.9 Bitförfrågningsfunktion

```
BIT_SIZE(I)
```

Returnerar antalet bitar enligt modellen för bit-representation i standarden ISO [21, avsnitt 13.5.7]. Normalt erhålles antalet bitar i ett helord.

D.10 Bitmanipuleringsfunktioner

Här användes modellen för bit-representation enligt standarden ISO [21, avsnitt 13.5.7].

```
BTEST(I,POS)          TRUE om position nr POS av I är 1
IAND(I,J)             Logisk addition av bitmönstren i I och J
IBCLR(I,POS)          Nollställer biten i position POS
IBITS(I,POS,LEN)      Använder LEN bitar av ordet I
                      med början i position POS,
                      övriga bitar nollställes;
                      POS + LEN <= BIT_SIZE(I)
IBSET(I,POS)          Sätter biten i position POS till 1
IEOR(I,J)             Utför logiskt exklusivt eller
IOR(I,J)              Utför logisk eller
ISHFT(I,SHIFT)        Utför logiskt skift ett antal steg,
                      åt höger om SHIFT < 0
                      åt vänster om SHIFT > 0
                      Nollor införes på lediga positioner
ISHFTC(I,SHIFT,size) Utför logiskt skift ett antal steg,
                      cirkulärt åt höger om SHIFT < 0
                      cirkulärt åt vänster om SHIFT > 0
                      Om SIZE finns med gäller
                      0 < SIZE <= BIT_SIZE(I)
                      Skift sker bara för de bitar som
                      finns i de SIZE högraste
                      positionerna, men alla om SIZE saknas.
NOT(I)                Utför logiskt komplement
```

D.11 Transferfunktion

```
TRANSFER(SOURCE,MOULD,size)
```

Denna funktion specificerar att den fysiska representationen av det första argumentet `SOURCE` skall behandlas som om det hade typ och parametrar som det andra `MOULD`, men utan att konverteras. Ändamålet är att erbjuda en möjlighet att flytta en storhet av en viss typ via en rutin som saknar just denna typ.

D.12 Flyttals-manipuleringsfunktioner

Dessa funktioner arbetar i en viss modell av heltals- respektive flyttalsaritmetik, se standarden ISO [21, avsnitt 13.7.1]. Funktionerna ger tal relaterade till aktuell variabel `X` av typ `REAL`. Funktioner som ger egenskaper hos tal av samma slag som variabeln `X` finns under punkt 8, Numeriska förfrågningsfunktioner.

```
EXPONENT(X)        Talets exponent
FRACTION(X)        Talets bråkdel
NEAREST(X,S)       Ger nästa representerbara tal
                   i angiven riktning (tecknet hos S)
```

```
RRSPACING(X)        Inverterade värdet av avståndet
                    mellan två tal i omgivningen
SCALE(X,I)          Multiplicerar X med basen upphöjd till I
SET_EXPONENT(X,I)   Ger det tal som har bråkdelen hos X
                    och exponenten I
SPACING(X)          Avståndet mellan två tal i
                    omgivningen
```

D.13 Vektor- och matrismultiplikation

`DOT_PRODUCT(VEKTOR_A,VEKTOR_B)`

Skapar skalärprodukten av två vektorer, vilka måste ha samma längd. Notera att om `VECTOR_A` är av typ `COMPLEX` så blir resultatet den komplexa varianten `SUM(CONJG(VECTOR_A)*VECTOR_B)`.

`MATMUL(MATRIS_A,MATRIS_B)`

Skapar matrisprodukten av två matriser, vilka måste vara konsistenta, dvs ha dimensionering `(M,K)` respektive `(K,N)`. Användes i kapitel 3, avsnitt 3.4.1, sid 40.

D.14 Fältfunktioner

`ALL(MASK,dim)`

Skapar ett logiskt värde som anger om alla relationer i `MASK` är sanna, eventuellt bara längst önskad dimension.

`ANY(MASK,dim)`

Skapar ett logiskt värde som anger om någon relation i `MASK` är sann, eventuellt bara längst önskad dimension.

`COUNT(MASK,dim)`

Returnerar ett numeriskt värde som anger antalet relationer i `MASK` som är sanna, eventuellt bara längst önskad dimension.

`MAXVAL(FAELT,dim,mask)`

Returnerar det största värdet i fältet `FAELT`, eventuellt bara av de som uppfyller relationen i `MASK`, eventuellt bara längst önskad dimension.

`MINVAL(FAELT,dim,mask)`

Returnerar det minsta värdet i fältet `FAELT`, eventuellt bara av de som uppfyller relationen i `MASK`, eventuellt bara längst önskad dimension.

`PRODUCT(FAELT,dim,mask)`

Skapar produkten av alla element i fältet `FAELT`, eventuellt bara längst önskad dimension, eventuellt utnyttjande `MASK` för att bestämma vilka element som skall vara med.

```
SUM(FAELT,dim,mask)
```

Returnerar summan av alla element i fältet `FAELT`, eventuellt bara av de som uppfyller relationen i `MASK`, eventuellt bara längst önskad dimension. Exempel finns i Bilaga B.11, sid 170.

D.15 Fältförfrågningsfunktioner

Se även Bilaga B, avsnitt B.10, sid 169, och avsnitt B.11, sid 170.

```
ALLOCATED(FAELT)
```

Logisk funktion som talar om ifall fältet `FAELT` är allokerat.

```
LBOUND(FAELT,dim)
```

Funktion som ger de lägre dimensioneringsgränserna för `FAELT`, om `DIM` ej är med erhålles en heltalsvektor, om `DIM` är med erhålles ett heltal med just den lägre dimensioneringsgränsen, för just den dimensionen.

```
SHAPE(SOURCE)
```

Funktion som ger mönstret för ett fält `SOURCE` som en heltalsvektor.

```
SIZE(FAELT,dim)
```

Funktion som ger antalet element i ett fält `FAELT` om `DIM` saknas, och antalet element i aktuell dimension om `DIM` finns.

```
UBOUND(FAELT,dim)
```

Funktion som ger de övre dimensioneringsgränserna för `FAELT`, om `DIM` ej är med erhålles en heltalsvektor, om `DIM` är med erhålles ett heltal med just den högre dimensioneringsgränsen, för just den dimensionen.

D.16 Fältkonstruktionsfunktioner

```
MERGE(TSOURCE,FSOURCE,MASK)
```

Funktion som förenar två fält, den ger elementet i `TSOURCE` om villkoret i `MASK` är sant och elementet i `FSOURCE` om det är falskt. De båda fälten `TSOURCE` och `FSOURCE` måste ha samma typ och mönster. Resultatet blir likaså av denna typ och mönster. Även `MASK` måste ha samma mönster.

Jag ger här ett ganska omfattande exempel på användning av `MERGE`, vilket även utnyttjar `RESHAPE` från nästa avsnitt för att bygga upp lämpliga testmatriser.

```
IMPLICIT NONE
INTERFACE
    SUBROUTINE SKRIV (A)
       REAL :: A(:,:)
    END SUBROUTINE SKRIV
    SUBROUTINE LSKRIV (A)
```

```
        LOGICAL :: A(:,:)
    END SUBROUTINE LSKRIV
END INTERFACE

REAL, DIMENSION(2,3)     :: TSOURCE, FSOURCE, RESULT
LOGICAL,  DIMENSION(2,3) :: MASK
TSOURCE = RESHAPE( (/ 11, 21, 12, 22, 13, 23 /), &
                   (/ 2,3 /) )
FSOURCE = RESHAPE( (/ -11, -21, -12, -22, -13, -23 /), &
                   (/ 2,3 /) )
MASK = RESHAPE( (/ .TRUE., .FALSE., .FALSE., .TRUE.,&
                   .FALSE., .FALSE. /), (/ 2,3 /) )

RESULT = MERGE(TSOURCE, FSOURCE, MASK)
CALL SKRIV(TSOURCE)
CALL SKRIV(FSOURCE)
CALL LSKRIV(MASK)
CALL SKRIV(RESULT)
END

SUBROUTINE SKRIV (A)
REAL :: A(:,:)
DO I = LBOUND(A,1), UBOUND(A,1)
   WRITE(*,*) ( A(I,J), J = LBOUND(A,2), UBOUND(A,2) )
END DO
RETURN
END SUBROUTINE SKRIV

SUBROUTINE LSKRIV (A)
LOGICAL :: A(:,:)
DO I = LBOUND(A,1), UBOUND(A,1)
   WRITE(*,"(8L12)") (A(I,J), J = LBOUND(A,2), UBOUND(A,2))
END DO
RETURN
END SUBROUTINE LSKRIV
```

Utmatningen blir följande:

```
 11.0000000  12.0000000  13.0000000
 21.0000000  22.0000000  23.0000000

-11.0000000 -12.0000000 -13.0000000
-21.0000000 -22.0000000 -23.0000000

          T           F           F
          F           T           F

 11.0000000 -12.0000000 -13.0000000
-21.0000000  22.0000000 -23.0000000
```

`PACK(FAELT,MASK,vector)`

Packar ett fält till en vektor under kontroll av `MASK`. Mönstret hos det logiska fältet `MASK` måste överensstämma med det för `FAELT` eller vara skalär. Om `VECTOR` är med måste den vara ett fält av rang 1 (dvs en vektor) med minst lika många element som är sanna i `MASK`, och ha samma typ som `FAELT`. Om `MASK` är en skalär med värdet sant måste `VECTOR` i stället ha minst lika många element som `FAELT`.

Resultatet blir en vektor med lika många element som de som i `FAELT` uppfyller villkoret om `VECTOR` saknas (dvs alla om `MASK` är en skalär med värdet sant), i annat fall lika många som i `VECTOR`. Värdena blir de godkända (dvs som uppfyller villkoret) i `FAELT` tagna i vanlig Fortran-ordning, om `VECTOR` är med och längre fylls det på med de värdena (oförändrad plats).

Det följande exemplet är baserat på en modifiering av det för `MERGE`, men jag ger nu bara resultatet.

```
FAELT
  11.0000000  12.0000000  13.0000000
  21.0000000  22.0000000  23.0000000

VEKTOR
 -11.0000000
 -21.0000000
 -12.0000000
 -22.0000000
 -13.0000000
 -23.0000000

MASK
   T           F           F
   F           T           F

PACK(FAELT, MASK)
  11.0000000
  22.0000000

PACK(FAELT, MASK, VEKTOR)
  11.0000000
  22.0000000
 -12.0000000
 -22.0000000
 -13.0000000
 -23.0000000
```

`SPREAD(SOURCE,DIM,NCOPIES)`

Funktionen `SPREAD (SOURCE, DIM, NCOPIES)` returnerar ett fält av samma typ som argumentet `SOURCE`, men med rangen ökad med ett. Parametrarna `DIM` och `NCOPIES` är heltal. Om `NCOPIES` är negativ så användes i stället värdet noll. Om `SOURCE` är en skalär så blir `SPREAD` helt enkelt en vektor med `NCOPIES` element som alla har samma värde som `SOURCE`. Parametern `DIM` anger vilket

index som skall utökas, det måste vara mellan 1 och `1+(rangen hos SOURCE)`, om `SOURCE` är en skalär måste således `DIM` vara ett. Parametern `NCOPIES` ger antalet element i den nya dimensionen, således ej antalet nya kopior. Ytterligare diskussion i lösningen till övning (3.1), sid 245.

```
UNPACK(VECTOR,MASK,FAELT)
```

Sprider en vektor till ett fält under kontroll av `MASK`. Mönstret hos det logiska fältet `MASK` måste överensstämma med det för `FAELT`. Fältet `VECTOR` måste ha rang 1 (dvs vara en vektor) med minst lika många element som är sanna i `MASK`, och ha samma typ som `FAELT`. Om `FAELT` ges som en skalär betraktas det som ett fält med samma mönster som `MASK` och samma skalära element överallt.

Resultatet blir ett fält med samma mönster som `MASK`, och samma typ som `VECTOR`. Värdena blir de från `VECTOR` för godkända (dvs som uppfyller villkoret i `MASK`) tagna i vanlig Fortran-ordning, medan övriga positioner i `FAELT` behåller sina gamla värden.

D.17 Fältomvandlingsfunktion

```
RESHAPE(SOURCE,SHAPE,pad,order)
```

Konstruerar ett fält med specificerat mönster `SHAPE` utgående från elementen i ett givet fält `SOURCE`. Om `PAD` saknas måste storleken på `SOURCE` vara minst `PRODUCT(SHAPE)`. Om `PAD` finns måste det ha samma typ som `SOURCE`. Om `ORDER` finns måste det vara `INTEGER` och ha samma mönster som `SHAPE`, och värdena måste vara en permutation av (1, 2, 3, ..., n), där n är antalet dimensioner i `SHAPE`, dvs högst 7.

Resultatet har naturligtvis ett mönster `SHAPE`, och elementen är de i som anges i `SOURCE` eventuellt kompletterade med `PAD`. De olika dimensionerna har permuterats vid tilldelningen av element om `ORDER` var med, men detta påverkar inte resultatets mönster. Exempel finns i Bilaga B, avsnitt B.9, sid 168. Ett mer avancerat exempel, som även behandlar de valfria argumenten, följer.

```
PROGRAM TEST_OPTIONAL_ARGUMENTS_RESHAPE
INTERFACE
    SUBROUTINE WRITE_MATRIX(A)
       REAL, DIMENSION(:,:) :: A
    END SUBROUTINE  WRITE_MATRIX
END INTERFACE

REAL, DIMENSION (1:9) :: B = (/ 11, 12, 13, &
                                14, 15, 16, 17, 18, 19 /)
REAL, DIMENSION (1:3, 1:3) :: C, D, E
REAL, DIMENSION (1:4, 1:4) :: F, G, H
INTEGER, DIMENSION (1:2) :: ORDER1 = (/ 1, 2 /)
INTEGER, DIMENSION (1:2) :: ORDER2 = (/ 2, 1 /)
REAL, DIMENSION (1:16)  :: PAD1 = &
    (/ -1, -2, -3, -4, -5, -6, -7, -8, &
       -9, -10, -11, -12, -13, -14, -15, -16 /)
```

```
C = RESHAPE( B, (/ 3, 3 /) )
CALL WRITE_MATRIX(C)

D = RESHAPE( B, (/ 3, 3 /), ORDER = ORDER1)
CALL WRITE_MATRIX(D)

E = RESHAPE( B, (/ 3, 3 /), ORDER = ORDER2)
CALL WRITE_MATRIX(E)

F = RESHAPE( B, (/ 4, 4 /), PAD = PAD1)
CALL WRITE_MATRIX(F)

G = RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER1)
CALL WRITE_MATRIX(G)

H = RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER2)
CALL WRITE_MATRIX(H)
END PROGRAM TEST_OPTIONAL_ARGUMENTS_RESHAPE

SUBROUTINE WRITE_MATRIX(A)
REAL, DIMENSION(:,:) :: A
WRITE(*,*)
DO I = LBOUND(A,1), UBOUND(A,1)
WRITE(*,*) (A(I,J), J = LBOUND(A,2), UBOUND(A,2))
END DO
END SUBROUTINE WRITE_MATRIX
```

Utmatningen av matriserna `C, D` och `E` blir

```
RESHAPE( B, (/ 3, 3 /) )
  11.0000000  14.0000000  17.0000000
  12.0000000  15.0000000  18.0000000
  13.0000000  16.0000000  19.0000000

RESHAPE( B, (/ 3, 3 /), ORDER = ORDER1)
  11.0000000  14.0000000  17.0000000
  12.0000000  15.0000000  18.0000000
  13.0000000  16.0000000  19.0000000

RESHAPE( B, (/ 3, 3 /), ORDER = ORDER2)
  11.0000000  12.0000000  13.0000000
  14.0000000  15.0000000  16.0000000
  17.0000000  18.0000000  19.0000000
```

Utmatningen av matriserna `F, G` och `H` (där man lagt till ytterligare en kolumn) blir

```
RESHAPE( B, (/ 4, 4 /), PAD = PAD1)
  11.0000000  15.0000000  19.0000000  -4.0000000
  12.0000000  16.0000000  -1.0000000  -5.0000000
  13.0000000  17.0000000  -2.0000000  -6.0000000
```

```
  14.0000000  18.0000000  -3.0000000  -7.0000000

RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER1)
  11.0000000  15.0000000  19.0000000  -4.0000000
  12.0000000  16.0000000  -1.0000000  -5.0000000
  13.0000000  17.0000000  -2.0000000  -6.0000000
  14.0000000  18.0000000  -3.0000000  -7.0000000

RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER2)
  11.0000000  12.0000000  13.0000000  14.0000000
  15.0000000  16.0000000  17.0000000  18.0000000
  19.0000000  -1.0000000  -2.0000000  -3.0000000
  -4.0000000  -5.0000000  -6.0000000  -7.0000000
```

D.18 Fältmanipuleringsfunktioner

Skiftfunktionerna lämnar mönstret av fältet oförändrat, men flyttar om elementen. De är så pass krångliga att jag rekommenderar studium även av standarden ISO [21].

```
CSHIFT(FAELT,SHIFT,dim)
```

Utför cirkulärt skift `SHIFT` positioner åt vänster om `SHIFT` är positivt, åt höger om det är negativt. Om `FAELT` är en vektor sker skiftet på ett naturligt sätt, om det är ett fält av högre ordning sker skiftet på alla sektioner längs dimension `DIM`.

Om `DIM` saknas inträffar samma som om det vore 1, annars måste det ha ett skalärt heltalsvärde mellan 1 och n = rangen hos `FAELT`. Argumentet `SHIFT` är ett skalärt heltal om `FAELT` har rang 1, annars kan det vara ett skalärt heltal eller ett heltalsfält av rang n-1 och samma mönster som `FAELT`, utom längs dimensionen `DIM` (som bortfaller på grund av den lägre rangen). Olika sektioner kan således skiftas olika mycket och åt olika håll.

```
EOSHIFT(FAELT,SHIFT,boundary,dim)
```

Utför skift `SHIFT` positioner åt vänster om `SHIFT` är positivt, åt höger om det är negativt, i stället för utskiftade element tas element från `BOUNDARY` in på andra sidan. Om `FAELT` är en vektor sker skiftet på ett naturligt sätt, om det är ett fält av högre ordning sker skiftet på alla sektioner längs dimension `DIM`. Om `DIM` saknas inträffar samma som om det vore 1, annars måste det ha ett skalärt heltalsvärde mellan 1 och n = rangen hos `FAELT`. Argumentet `SHIFT` är ett skalärt heltal om `FAELT` har rang 1, annars kan det vara ett skalärt heltal eller ett heltalsfält av rang n-1 och med samma mönster som fältet `FAELT`, utom längs dimensionen `DIM` (som bortfaller på grund av den lägre rangen).

Motsvarande gäller `BOUNDARY`, som dock naturligtvis har samma typ som `FAELT`. Om parametern `BOUNDARY` saknas användes i stället lämpligt val av noll, `.FALSE.` eller blank beroende på datatyp. Olika sektioner kan således skiftas olika mycket och åt olika håll.

Ett enkelt exempel på ovanstående båda funktioner i vektorfallet följer.

```
      REAL, DIMENSION(1:6) :: A = (/ 11.0, 12.0, 13.0, 14.0, &
                                     15.0, 16.0 /)
      REAL, DIMENSION(1:6) :: X, Y
      WRITE(*,10) A
      X = CSHIFT ( A, SHIFT = 2)
      WRITE(*,10) X
      Y = CSHIFT ( A, SHIFT = -2)
      WRITE(*,10) Y
      X = EOSHIFT ( A, SHIFT = 2)
      WRITE(*,10) X
      Y = EOSHIFT ( A, SHIFT = -2)
      WRITE(*,10) Y
10    FORMAT(1X,6F6.1)
      END
```

Utskriften blev följande.

```
 11.0  12.0  13.0  14.0  15.0  16.0
 13.0  14.0  15.0  16.0  11.0  12.0
 15.0  16.0  11.0  12.0  13.0  14.0
 13.0  14.0  15.0  16.0   0.0   0.0
  0.0   0.0  11.0  12.0  13.0  14.0
```

Ett enkelt exempel på ovanstående båda funktioner i matrisfallet följer. Jag har även här utnyttjat `RESHAPE` för att skapa en lämplig matris att utgå från.

```
 11.0  12.0  13.0   Z = RESHAPE( B, (/3,3/) )
 14.0  15.0  16.0
 17.0  18.0  19.0

 17.0  18.0  19.0   X = CSHIFT ( Z, SHIFT = 2)
 11.0  12.0  13.0
 14.0  15.0  16.0

 13.0  11.0  12.0   X = CSHIFT ( Z, SHIFT = 2, DIM = 2)
 16.0  14.0  15.0
 19.0  17.0  18.0

 14.0  15.0  16.0   X = CSHIFT ( Z, SHIFT = -2)
 17.0  18.0  19.0
 11.0  12.0  13.0

 17.0  18.0  19.0   X = EOSHIFT ( Z, SHIFT = 2)
  0.0   0.0   0.0
  0.0   0.0   0.0

 13.0   0.0   0.0   X = EOSHIFT ( Z, SHIFT = 2, DIM = 2)
 16.0   0.0   0.0
 19.0   0.0   0.0

  0.0   0.0   0.0   X = EOSHIFT ( Z, SHIFT = -2)
```

```
 0.0   0.0   0.0
11.0  12.0  13.0
```

`TRANSPOSE(MATRIS)`

Transponerar en matris, dvs ett fält av rang 2. Den byter således rader och kolumner i en matris.

D.19 Lokaliseringsfunktioner

`MAXLOC(FAELT,mask)`

Returnerar positionen för det största elementet i fältet `FAELT`, eventuellt bara av de som uppfyller relationen i `MASK`. Resultatet kommer som en heltalsvektor! Användes i lösningen till övning (3.1), sid 245.

`MINLOC(FAELT,mask)`

Returnerar positionen för det minsta elementet i fältet `FAELT`, eventuellt bara av de som uppfyller relationen i `MASK`. Resultatet kommer som en heltalsvektor!

I Fortran 95 tillkom att dessa funktionener kunde ha ytterligare ett frivilligt argument, dvs `MAXLOC(FAELT,dim,mask)` resp. `MINLOC(FAELT,dim,mask)`.

D.20 Pekarförfrågansfunktioner

`ASSOCIATED(POINTER,target)`

Logisk funktion som talar om ifall pekaren `POINTER` är associerad med ett mål, om `TARGET` är med om den är associerad med just det målet. Om både `POINTER` och `TARGET` är pekare blir resultatet sant endast om båda är associerade med samma mål. Se i första hand kapitel 11, Pekare.

`NULL()` är en funktion introducerad i Fortran 95 för initial "nollställning" av pekare. Den användes på följande sätt på vektorn `VEKTOR`.

```
REAL, POINTER, DIMENSION(:) :: VEKTOR => NULL()
```

Argumentet är inte nödvändigt, om det är närvarande bestämmer det egenskaperna hos pekaren, vilka annars bestämmes ur sammanhanget.

D.21 Inbyggda subrutiner

Det finns i Fortran 95 sex inbyggda subrutiner (tidsrutiner, bitkopieringsrutin och slumptalsrutiner).

D.21.1 Tidsrutiner

Här finns de två rutinerna `DATE_AND_TIME` och `SYSTEM_CLOCK` samt den nya (från Fortran 95) `CPU_TIME`.

`DATE_AND_TIME(date,time,zone,values)`

Subrutin som ger datum, tid och tidszon. Minst ett argument måste ges.

DATE skall vara en skalär textsträngsvariabel med minst 8 tecken, och den tilldelas värdet CCYYMMDD för århundrade, år, månad och dag. Samtliga ges numeriskt, med blanka om systemet ej innehåller datum.

TIME skall vara en skalär textsträngsvariabel med minst 10 tecken, och den tilldelas värdet hhmmss.sss för timme, minut och sekunder. Samtliga ges numeriskt, med blanka om systemet ej innehåller klocka.

ZONE skall vara en skalär textsträngsvariabel med minst 5 tecken, och den tilldelas värdet +hhmm för tecken, timme och minut för den lokala tidsdifferensen mot UTC (före detta GMT). Samtliga ges numeriskt, med blanka om systemet ej innehåller klocka. Vi får således här i Sverige +0100 under vintern och +0200 under sommaren.

VALUES är i stället en heltalsvektor med minst 8 element, vilket gör det lättare att använda resultaten från DATE_TIME vid beräkningar i det egna programmet. Om systemet saknar datum respektive klocka erhålles -HUGE(0), dvs det minsta heltalet i modellen, som svar. Vektorn kommer att innehålla följande element: år, månad, dag, tidsdifferens i minuter, timme, minut, sekund och millisekunder.

```
SYSTEM_CLOCK(count,count_rate,count_max)
```

Subrutin som ger systemtiden. Minst ett argument måste ges.

COUNT är ett skalärt heltal som ger systemtiden, vilken uppräknas med 1 för varje cykel, upp till COUNT_MAX, då den börjar om. Om systemklocka saknas erhålles -HUGE(0).

COUNT_RATE är ett skalärt heltal som ger antalet cykler per sekund. Om systemklocka saknas erhålles 0.

COUNT_MAX är ett skalärt heltal som ger det maximala värde som COUNT kan nå. Om systemklocka saknas erhålles 0.

```
CPU_TIME(TIME)
```

Denna subrutin introducerades i Fortran 95. I den skalära flytpunktsvariabeln TIME erhålles processortiden i sekunder. För att få tiden för en viss uppgift måste en subtraktion utföras. Den exakta tidsbestämningen, speciellt för parallella processorer, är implementationsberoende.

D.21.2 Bitkopieringsrutin

```
MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)
```

Subrutin som kopierar den följd av bitar i FROM som startar i position FROMPOS och har längden LEN till TO, med början i position TOPOS. Övriga bitar ändras ej. Alla storheter måste vara heltal, alla utom TO med INTENT(IN), medan TO skall ha INTENT(INOUT) och ha samma slag som FROM. Samma variabel kan vara både FROM och TO. Vissa naturliga restriktioner på tillåtna värden på LEN, FROMPOS och TOPOS gäller, även med hänsyn till BIT_SIZE.

D.21.3 Slumptalsrutiner

Här finns de två rutinerna RANDOM_NUMBER och RANDOM_SEED.

En följd av pseudoslumptal genereras från ett startvärde, som är lagrat som en heltalsvektor. Subrutinerna erbjuder ett flyttbart gränssnitt gentemot en implementationsberoende slumptalsföljd.

```
RANDOM_NUMBER(HARVEST)
```

Denna subrutin returnerar i flyttalsvariabeln `HARVEST` ett (eller flera om `HARVEST` är ett fält) slumptal mellan noll och ett.

```
RANDOM_SEED(size,put,get)
```

Denna subrutin omstartar eller ger information om slumptalsgeneratorn. Inget argument behöver vara med. Utvariabeln `SIZE` skall vara ett skalärt heltal och den ger antalet heltal (n) som processorn utnyttjar för startvärdet. Invariabeln `PUT` är en heltalsvektor som sätter startvärdet, medan utvariabeln `GET` avläser aktuellt startvärde. Exempel:

```
CALL RANDOM_SEED                    Initiering
CALL RANDOM_SEED (SIZE=K)           Sätter K = n
CALL RANDOM_SEED (PUT = SEED(1:K)) Använder användarens
                                      startvärden
CALL RANDOM_SEED (GET = OLD(1:K))  Avläser aktuella
                                      startvärden
```

Bilaga E

Fortrans utveckling

Inledning

Följande enkla program, vilket utnyttjar många olika vanliga koncept inom programmering, är baserat på "The Early Development of Programming Languages" av Donald E. Knuth och Luis Trabb Pardo [22, sid 197 - 273]. De gav ett exempel i Algol 60 och översatt till några mycket gamla språk som Zuse's Plankalkül, Goldstine's Flow diagrams, Mauchly's Short Code, Burks' Intermediate PL, Rutishauser's Klammerausdrücke, Böhm's Formules, Hopper's A-2, Laning och Zierler's Algebraic interpreter, Backus' FORTRAN 0, och Brooker's AUTOCODE.

Programmet ges här i Pascal, C, och i fem varianter av Fortran, i första hand för att visa hur Fortran utvecklats från ett kryptiskt närmast maskinberoende språk till ett modernt strukturerat högnivåspråk. Slutligen behandlas även det nya utbildningsspråket F.

E.1 Program TPK i Pascal för UNIX

```
program tpk(input,output);
(*  Pascal program for Unix  *)
var     i : integer;
        y : real;
        a : array [0..10] of real;
        function f ( t : real) : real;
           begin
                f := sqrt(abs(t)) + 5*t*t*t
           end;
begin
        for i := 0 to 10 do read(a[i]);
        for i := 10 downto 0 do
        begin
          y := f(a[i]);
          if y  >  400 then
                writeln(i,' TOO LARGE')
          else
```

```
                writeln(i,y);
        end
end.
```

Detta program innehåller variabler av typ heltal och flyttal samt en vektor av flyttal. Det innehåller vidare inbyggda matematiska funktioner och en funktion f(t) skriven av användaren, upprepningssatser både framlänges och baklänges, villkorssats samt utmatning av både flyttal och text.

Notera att exponentiering ej finns i Pascal, varför t^3 måste skrivas `t*t*t`.

E.2 Program TPK i ANSI C

```
#include <stdio.h>
#include <math.h>
/*  Program TPK in ANSI C       */
double f (double t);
main()
{
        int i;
        double y;
        double a[11];
        for ( i = 0; i <= 10; ++i)
                scanf("%lf", &a[i]);
        for ( i = 10; i >= 0; i = i - 1 )
        {
                y = f(a[i]);
                if ( y > 400 )
                        {printf(" %d",i);
                         printf(" TOO LARGE\n");}
                else
                        {printf(" %d",i);
                         printf(" %lf",y);
                         printf(" \n");}
        }
        return 0;
}
/*  Function */
double f (double t)
{
        double temp;
        temp = sqrt(fabs(t)) + 5*pow(t,3);
        return temp;
}
```

Språket C har i den traditionella varianten den egenheten att funktionsanrop normalt sker i dubbel precision, varför hela programmet här är i dubbel precision. I ANSI C är det dock tillåtet med enkel precision. Exponentiering kan ske med en inbyggd funktion `pow`.

E.3 Fortran

Fortran 90 är faktiskt den första som stavas Fortran, alla de tidigare hette egentligen FORTRAN. På 1960-talet rekommenderade dock IBM stavningen FØRTRAN.

E.3.1 FORTRAN 0

```
        DIMENSION A(11)
        READ A
2       DO 3,8,11 J=1,11
3       I=11-J
        Y=SQRT(ABS(A(I+1)))+5*A(I+1)**3
        IF (400>=Y) 8,4
4       PRINT I,999.
        GOTO 2
8       PRINT I,Y
11      STOP
```

Notera den eleganta behandlingen av inläsningen av vektorn `A` och att symbolen > fanns med från början. Utmatning av text fanns ej, varför 999 användes för att markera ett för stort tal. `DO`-slingan var mindre elegant, siffrorna anger startrad och slutrad för slingan samt vart exekveringen skulle hoppa när slingan var avslutad. Inte heller villkorssatsen kan sägas vara användarvänlig. Exponentiering kan ske med `**`.

E.3.2 FORTRAN I

```
C       THE TPK ALGORITHM
C       FORTRAN I STYLE
        FUNF(T)=SQRTF(ABSF(T))+5.0*T**3
        DIMENSION A(11)
    1   FORMAT(6F12.4)
        READ 1,A
        DO 10 J=1,11
        I=11-J
        Y=FUNF(A(I+1))
        IF(400.0-Y)4,8,8
    4   PRINT 5,I
    5   FORMAT(I10,10H TOO LARGE)
        GOTO 10
    8   PRINT 9,I,Y
    9   FORMAT(I10,F12.7)
   10   CONTINUE
        STOP 52525
```

Detta var det första programspråket med kommentarer! I övrigt användes en satsfunktion, och baklängesslingan är simulerad eftersom en `DO`-slinga ej fick gå baklänges och index noll var förbjudet. Villkorssatsen är den aritmetiska med hopp till tre olika ställen beroende på om uttrycket är mindre än noll, lika med noll eller större än noll. Alla funktionsnamn slutar på `F`. Text går att mata ut

om man kan räkna på fingrarna (10H anger att tio tecken följer). Strukturerad layout var ännu ej uppfunnen, den var dessutom bökig på hålkort.

E.3.3 FORTRAN IV eller Fortran 66

```
C       THE TPK ALGORITHM
C       FORTRAN IV STYLE
        DIMENSION A(11)
        FUN(T) = SQRT(ABS(T)) + 5.0*T**3
        READ (5,1) A
    1   FORMAT(5F10.2)
        DO 10 J = 1, 11
                I = 11 - J
                Y = FUN(A(I+1))
                IF (400.0-Y) 4, 8, 8
    4                   WRITE (6,5) I
    5                   FORMAT(I10, 10H TOO LARGE)
                GO TO 10
    8                   WRITE(6,9) I, Y
    9                   FORMAT(I10, F12.6)
   10     CONTINUE
        STOP
        END
```

Även här användes en satsfunktion och baklängesslingan är fortfarande simulerad. Strukturerad layout användes. Benämningen Fortran 66 började användas först efter 1978, dessförinnan var IBM-beteckningen FORTRAN IV den vanliga.

E.3.4 Fortran 77

```
        PROGRAM TPK
C       THE TPK ALGORITHM
C       FORTRAN 77 STYLE
        REAL A(0:10)
        READ (5,*) A
        DO 10 I = 10, 0, -1
                Y = FUN(A(I))
                IF ( Y .LT. 400) THEN
                        WRITE(6,9) I, Y
    9                   FORMAT(I10, F12.6)
                ELSE
                        WRITE (6,5) I
    5                   FORMAT(I10,' TOO LARGE')
                ENDIF
   10   CONTINUE
        END
        REAL FUNCTION FUN(T)
        REAL T
        FUN = SQRT(ABS(T)) + 5.0*T**3
        END
```

Även här borde en satsfunktion ha använts, men det hade nu blivit omodernt, varför en extern funktion användes. Baklängesslingan behöver inte längre simuleras och villkorssatsen kan nu ges på ett naturligt sätt. Strukturerad layout användes nu normalt. Texten kan ges inom apostrofer, dvs man behöver inte längre manuellt räkna antalet tecken i textsträngen. Satsen `END` i huvudprogrammet tjänstgör även som exekveringsslut, dvs `STOP` behövs ej. På motsvarande sätt behövs ej `RETURN` i funktionen `FUN`.

E.3.5 Fortran 90

```
      PROGRAM TPK
!     The TPK Algorithm
!     Fortran 90 style
      IMPLICIT NONE
      INTEGER               :: I
      REAL                  :: Y
      REAL, DIMENSION(0:10) :: A
      READ (*,*) A
      DO I = 10, 0, -1        ! Backwards
            Y = FUN(A(I))
            IF ( Y < 400.0 ) THEN
                  WRITE (*,*) I, Y
            ELSE
                  WRITE (*,*) I, ' Too large'
            END IF
      END DO
      CONTAINS                ! Local function
            FUNCTION FUN(T)
            REAL :: FUN
            REAL, INTENT(IN) :: T
            FUN = SQRT(ABS(T)) + 5.0*T**3
            END FUNCTION FUN
      END PROGRAM TPK
```

I stället för en extern funktion användes nu en intern funktion. Baklängesslingan behöver naturligtvis inte simuleras. Villkorssatsen kan nu ges på ett naturligt sätt och symbolen $<$ har återkommit från FORTRAN 0. Deklarationerna har ett nytt utseende. Strukturerad layout användes nu alltid, alla satsnummer är borta. Texten ges inom apostrofer, och standardformat och standardenheter utnyttjas. Kommentarer kan ges även direkt på raden. Ny symbol ! som inleder kommentarer. Implicit deklaration är lämpligen avstängd.

Detta exempel är oförändrat under Fortran 95!

E.3.6 F

Det nya språket F har utvecklats som en möjlig ersättare för Pascal vid programmeringsutbildning, men det är samtidigt en äkta delmängd av Fortran 90 och Fortran 95, se vidare Brainerd m. fl. [4].

```
      module Functions
      public :: fun
      contains
         function fun(t) result (r)
            real, intent(in) :: t
            real  :: r
            r = sqrt(abs(t)) + 5.0*t**3
         end function fun
      end module Functions


      program TPK
!     The TPK Algorithm
!     F style
      use Functions
      integer     :: i
      real                   :: y
      real, dimension(0:10) :: a
      read *, a
      do i = 10, 0, -1      ! Backwards
         y = fun(a(i))
         if ( y < 400.0 ) then
            print *, i, y
         else
            print *, i, " Too large"
         end if
      end do
      end program TPK
```

Nyckelord är reserverade ord och måste ges med små bokstäver (gemener), andra identifierare kan vara blandat stora och små bokstäver (men bara i en kombination per variabel, och på grund av att det är en delmängd av Fortran kan inte Var och vaR avse olika variabler). Funktioner och subrutiner tillåtes enbart via moduler, funktioner måste ha en resultatvariabel. Alla variabler måste deklareras explicit (satsen `IMPLICIT NONE` är därför implicit, och fick tidigare inte förekomma explicit, numera är den dock tillåten av kompatibilitetsskäl).

När den är tillgänglig är därför kompilatorväljaren `-u` mycket användbar vid test av F program med en Fortran 95 kompilator.

Bilaga F

Laborationer

Inledning

Dessa laborationer är små variationer av de som utnyttjas i Fortran-undervisningen vid Linköpings Tekniska Högskola.

När jag nedan skriver kursbiblioteket så är det en hänvisning till att respektive filer bör göras tillgängliga på lämpligt sätt för eleverna, men vid exempelvis självstudier kan man själv skriva in dem på ett lämpligt filbibliotek. Alternativt kan man hämta filerna från vårt kursbibliotek. Aktuella filer finns även bland övriga exempel i internetversionen under katalogen kod. Under UNIX är det lätt att införa symboliska namn på bibliotek, så jag kan kalla kursbiblioteket för `$KURSBIB` i stället för det fullständiga namnet på biblioteket på vårt lokala system, nämligen `/mailocal/lab/numt/TANA70/`

Lokalt gäller att både Fortran 77, Fortran 90 och Fortran 95 är tillgängliga på MAI:s Sun arbetsstationer.

Ytterligare information om kursen (även tidigare elevers synpunkter) finns på URL = `http://www.mai.liu.se/~boein/kurser/TANA70`

Lämplig miljö erhålles med kommandona

```
setenv KURSBIB /mailocal/lab/numt/TANA70/
set path=(/mailoca/lab/numt/TANA70 ./ $path)
module add workshop
```

eller något enklare

```
source /mailocal/lab/numt/TANA70/.cshrc
```

eller ännu enklare

```
TANA70setup
```

F.1 Labb 1, Runge-Kutta

Denna laboration är i första hand tänkt för dem som väl behärskar programspråket Pascal.

Pascal-programmet nedan för lösning av en ordinär differentialekvation med Runge-Kuttas metod finns på kursbiblioteket, nämligen på filen `lab1.p`. Din uppgift är att översätta det från Pascal till Fortran. Utskriften från programmet skall förutom siffervärden innehålla lämplig text. Det erfordras ej men betraktas som en fördel om programmet även ändras till att kunna använda andra startvärden på `x` och `y` än de i Pascal-programmet inbyggda, nämligen 1 och 2.

Följande skall lämnas in:

- Lista på det översatta programmet.
- Lista (vid UNIX från `script`) på körning av de fyra testfallen

```
Antal steg      Steglängd

     1              1
     2              0.5
     4              0.25
     8              0.125
```

Den som inte behärskar Pascal bör i stället leta rätt på en lärobok i numeriska metoder, till exempel [14, avsnitt 10.4], och direkt koda upp ett program för Runge-Kutta i Fortran.

Uppgiften är således att med Runge-Kuttas metod lösa differentialekvationen $y'(x) = x^2 + \sin(xy)$ med begynnelsevärdet $y(1) = 2$ fram till $x = 2$ med ovan angivna steglängder. Formlerna för Runge-Kuttas metod finns förutom i läroböcker i numeriska metoder bland annat i beskrivningen av laboration 8.

```
program RK1;
    (* Enkelt program i Pascal för Runge-Kuttas metod för
       första ordningens differentialekvation.
                dy/dx = x^2 + sin(xy)
                y(1) = 2                                    *)
var     antal, i : integer;
        h, k1, k2, k3, k4, x, y : real;
        function f(x,y : real) : real;
            begin
                  f := x*x + sin(x*y)
            end;
begin
      antal := 1;
        while antal > 0 do
        begin
            x := 1.0;
            y := 2.0;
            writeln(' Ge antal steg ');
            read(antal);
            if antal >= 1 then
               begin
                  writeln(' Ge steglängd ');
                  read(h);
```

```
                writeln('       x              y');
                writeln(x, y);
                for i := 1 to antal do
                begin
                     k1 := h*f(x,y);
                     k2 := h*f(x+0.5*h,y+0.5*k1);
                     k3 := h*f(x+0.5*h,y+0.5*k2);
                     k4 := h*f(x+h,y+k3);
                     x  := x + h;
                     y  := y + (k1+2*k2+2*k3+k4)/6;
                     writeln(x, y);
                end;
            end;
        end;
end.
```

Om Du önskar köra detta Pascal program måste Du eventuellt komplettera den första raden

```
program RK1;
```

till följande

```
program RK1(INPUT, OUTPUT);
```

eftersom en del system har detta krav.

F.2 Labb 2, Horners schema och filhantering

Programmet nedan är i Fortran 77 (eller fix form Fortran 90) och finns på kursbiblioteket på filen `lab2.f`. Rätta de fel, som finns i programmet. Det finns inget fel i själva den numeriska algoritmen (Horners schema, se [14, avsnitt 4.6] och [15, sid 19]) eller i kommentarerna i programmet. Däremot är det en del programmeringsfel inlagda, flera av dem beroende på en sammanblandning med Pascal. När programmet är korrekt (testa gärna med ekvationen $x^2 + 2 = 0$), ändra programmet så att det hämtar data från filen `lab21.dat`. Ändringarna skall göras så att användaren kan välja om data skall hämtas direkt från arbetsstation eller från fil, i det senare fallet skall användaren ge filens namn. Kolla att programmet även fungerar för filen `lab22.dat`. Notera att programmet använder komplexa tal, liksom att värdena på sådana tal läses in som talpar inom parentes. Programmet finns numera även i Fortran 90 fri form som `lab2.f90`.

Följande skall lämnas in:

- Lista på det rättade och modifierade programmet.
- Lista (vid UNIX från `script`) på körning av de tre testexemplen ($x^2+2 = 0$) respektive `lab21.dat` och `lab22.dat`.

Filen `lab2.f` till laboration 2. Hitta felen i programmet!

```
************************************************************
**      Programmet beräknar alla rötter (reella och komplexa)
**      till ett N:te-gradspolynom med komplexa koefficienter.
**      (N <= 10)
**
**                n        n-1
**       P(z) = a z  + a   z    + ... + a z + a
**               n      n-1              1     0
**
**      Avbrott sker om
**                1) Abs (Z1-Z0) < EPS     ==>
**                   Roten funnen = Z1
**                2) ITER > MAX  Långsam konvergens  ==>
**                   Avbrott
**
**      Programmet sätter EPS till 1.0E-7 och MAX till 30
**
**      Använd metod är NEWTON-RAPHSONS metod:
**      z1 = z0 - P(z0)/P'(z0)
**      Värdet av P(z0) och P'(z0) beräknas med hjälp av
**      HORNERS SCHEMA.
**
**      Fältet A(0:10) innehåller polynomets komplexa
**      koefficienter.
**      Fältet B(1:10) innehåller de komplexa
**      koefficienterna till polynomet Q(z),
**                  där P(Z) = (z-z0)*Q(z) + P(z0)
**      Koefficienterna till Q(z) erhålles med hjälp av
**      HORNERS SCHEMA.
**
**      När första roten erhållits med NEWTON-RAPHSONS metod,
**      divideras den bort (s.k. deflation).
**      Kvotpolynomet = Q(z).
**      Proceduren upprepas därefter med koefficienterna
**      till Q(z) som indata.
**      Som startapproximation används i samtliga fall
**      STARTV = 1+i
**      Z0 är föregående approximation till roten.
**      Z1 är senast beräknade approximation till roten
**      F0 = P(Z0)
**      FPRIM0 = P'(Z0)
************************************************************
      COMPLEX      A(0:10), B(1:10), Z0, Z1, STARTV
      INTEGER      N, I, ITER, MAX
      REAL         EPS
      DATA  EPS/1E-7/, MAX /30/, STARTV /(1,1)/
************************************************************
20    WRITE(6,*)  'Ange polynomets gradtal'
      READ (5,*) N
      IF (N .GT. 10) THEN
```

```
         WRITE(6,*) 'Polynomets gradtal får inte vara större',
     ,' än 10'
      GOTO 20
      WRITE (6,*) 'Ge polynomets koefficienter, som komplexa',
     ,' konstanter'
      WRITE (6,*) 'Högstagradskoefficienten först'
      DO 30 I = N, 0, -1
          WRITE (6,*) 'A(' , I, ') = '
          READ  (5,*)  A(I)
30    CONTINUE
      WRITE (5,*) '    Rötterna är','        Antal Iterationer'
*************************************************************
40    IF (N GT 0) THEN
C     ******     Finn nästa rot     ******
          Z0 = (0,0)
          ITER = 0
          Z1 = STARTV
50        IF (ABS(Z1-Z0) .GE. EPS) THEN
C         ++++++ Fortsätt iterera tills två efterföljande rötter
C         ++++++ är tillräckligt nära varandra.
              ITER = ITER + 1
              IF (ITER .GT. MAX) THEN
C             ------   För många iterationer  ==> Avbrott
                  WRITE (6,*) 'För många iterationer.'
                  WRITE (6,*) 'Senaste approximationen till',
     ,                        ' roten är ',Z1
              GOTO 200
              ENDIF
              Z0 = Z1
              HORNER (N, A, B, Z0, F0, FPRIM0)
C             ++++++   NEWTON-RAPHSONS METOD
              Z1 = Z0 - F0/FPRIM0
          GOTO 50
          ENDIF
C         ++++++++++++++++++++++++++++++++++++++++++++++++++++
100       WRITE (6,*) Z1, '         ',Iter
C         ****** Roten funnen. Dividera bort den och
C                sök nästa rot
          N = N - 1
          FOR I = 0 TO N DO
              A(I) = B(I+1)
      GOTO 40
      ENDIF
200   END

      SUBROUTINE  HORNER(N, A, B, Z, F, FPRIM)
******    Parametrarnas betydelse - se huvudprogrammet   ******
******    BI och CI hjälpvariabler.                      ******

      INTEGER N, I
```

```
      COMPLEX A(1:10), B(0:10), Z, F, FPRIM, BI, CI

      BI = A(N)
      B(N) = BI
      CI = BI

      DO 60  I = N-1, 1, -1
             BI = A(I) + Z*BI
             CI = BI + Z*CI
             B(I) = BI
C            ++++++ BI = B(I) för beräkning av P(Z) ++++++
C            ++++++ CI  för beräkning av P'(Z)      ++++++
60    CONTINUE

      F = A(0) + Z*BI
      FPRIM = CI

      RETURN
      END
******   SLUT  HORNERS SCHEMA

******   Programmet är komponerat av Ulla Ouchterlony 1984
```

Kommentarer.
Jag rekommenderar följande process vid lösningen av denna uppgift.

1. Läs programmet och rätta de fel Du hittat.
2. Kör programmet och se om det fungerar till belåtenhet.
3. Om det ej fungerar, rätta de fel som Du hittat nu och gå till punkt 2. Om Du inte hittar något fel kompilerar Du om programmet med påslagen kontroll av odeklarerade variabler. Detta sker under vissa system med

   ```
   f95 -u lab2.f
   a.out
   ```

   Under moderna system, till exempel Fortran 90/95, kan man i stället lägga in satsen `IMPLICIT NONE` först i både huvudprogram och subrutin. Hos Absoft har `-u` en helt annan betydelse, jag har inte hittat motsvarande kommando under denna kompilator.
4. Rätta de fel som Du hittat nu och gå till punkt 2. Om Du inte hittar något fel kompilerar Du nu om programmet med påslagen kontroll av fältindex. På vissa system kan detta ske med följande kommandon.

   ```
   f95 -C lab2.f
   a.out
   ```

   Under Absoft ger man istället `f90 -Rb` och vid Intel ifort ger man `ifort -CB`.

5. Rätta de fel som Du hittat nu och gå till punkt 2.

6. Om Du inte hittar felet, det blir till exempel fel resultat eller utskriften "segmentation error", så kan Du försöka med avlusaren eller debuggern. På de flesta UNIX-system startar Du denna med kommandot

   ```
   dbx a.out
   ```

   om Ditt kompilerade program heter `a.out`. Som första kommando ger Du run och kör sedan som vanligt. Eventuella fel under exekveringen kommer nästan ut i klartext! För att komma ut från debuggern skriver Du `quit`. Se även kapitel 15.

7. När programmet fungerar börjar Du med att modifiera programmet till att kunna hantera indata även från fil.

8. Hur man öppnar en fil framgår av avsnittet 7.1, sid 72.

9. Vid liststyrd inmatning av text bör texten inneslutas inom apostrofer (kräves på vissa system). Detta är obekvämt för användaren. Man bör därför i stället använda formatstyrd inmatning. Om man skall mata in ett tecken användes lämpligen `FORMAT(A1)`, om det gäller 13 tecken användes lämpligen `FORMAT(A13)`.

10. Kom ihåg att en läs- eller skrivsats i en explicit `DO`-slinga alltid börjar med en ny post (ny rad). Om man har flera data på samma rad måste dessa därför läsas med samma lässats, varvid man kan använda en implicit `DO`-slinga, se avsnitt 6.5, sid 65.

    Den explicita `DO`-slingan nedan kommer att skriva sju rader, var och en med ett värde av `X`

    ```
    DO I = 1, 13, 2
       WRITE(*,*) X(I)
    END DO
    ```

    Den implicita `DO`-slingan nedan kommer att skriva en rad med sju värden av `X`

    ```
    WRITE(*,*) ( X(I), I = 1, 13, 2)
    ```

11. Kom ihåg att eftersom Newton-Raphson bara är garanterat konvergent nära roten finns det ingen garanti för att det rättade programmet fungerar på godtyckliga indata.

12. Enligt standarden skall komplexa tal skrivas med parentes om realdel och imaginärdel, även om imaginärdelen är noll, som i (1.0, 0.0). Vissa kompilator, bland annat Intel ifort tillåter att man i detta fall bara skriver realdelen enligt 1.0, varvid imaginärdelen automatiskt blir noll.

    Prova gärna om ditt system tillåter detta förenklade skrivsätt, men använd det då bara vid inmatning via tangentbordet och aldrig vid inmatning från fil (av portabilitetsskäl).

F.3 Labb 3, Fakultet och Bessel

Detta är laborationsuppgift nummer tre, och den består i att skriva två små program i programspråket Fortran.

- **Deluppgift a) Fakultet**
  Skriv en funktion i Fortran med tillhörande huvudprogram för beräkning av fakulteten. Använd genomgående heltal. Skriv gärna huvudprogrammet så att det ber om ett värde på det heltal för vilket fakulteten önskas beräknad. Provkör och beräkna 10!

- **Deluppgift b) Bessel**
  Skriv ett program för tabellering av Bessel-funktionen $J_0x)$. Använd därvid NAG:s Fortran 77 bibliotek eller NAG:s Fortran 90 bibliotek, (eller något motsvarande bibliotek) för att få tag på denna funktion. Skriv gärna huvudprogrammet så att det ber om det intervall för vilket funktionen önskas beräknad. Mata ut en tabell med x och $J_0(x)$ för $x = 0.0, 0.1, ..., 1.0$. Se till att utmatningen ser snygg ut! Provkör!

  Vid användning av NAG:s Fortran 77 bibliotek har man nytta av NAG User Notes, vilka beskriver de maskinspecifika delarna.

  Aktuell Bessel-funktion i NAG-biblioteket heter `S17AEF` och har argumenten `X` och `IFAIL` och i denna ordning, där `X` är argumentet för Besselfunktionen i enkel eller dubbel precision, och där `IFAIL` är en felparameter (heltal). Denna senare sättes lämpligen till 1 vid ingången och avläses vid återhoppet. Om den då är noll så är allt OK, om den är 1 så är argumentet för stort (till beloppet).

  Om NAG-biblioteket är installerat på normalt vis länkar man in det med kommandot

  ```
  f77 prog.f -lnag
  ```

  där `prog.f` är Ditt program.

  Det är även tillåtet att använda NAG:s nya Fortran 90 bibliotek. Notera att funktionerna har andra namn i detta bibliotek (vår Bessel-funktion heter `nag_bessel_j0)`. Du måste använda en eller flera moduler, till exempel

  ```
  USE nag_bessel_fun, ONLY : nag_bessel_j0
  ```

  samt länka med

  ```
  f90 prog.f90 -lnagfl90
  ```

  Felhanteringen är helt annorlunda i detta bibliotek, det finns ingen parameter `IFAIL`. Du kan därför överlåta felhanteringen till systemet, som automatiskt signalerar om något gått fel.

  Man kompilerar och länkar in erforderliga bibliotek med ett implementationsberoende kommando, till exempel på Sun

  ```
  f90 prog.f -lnag -lF77
  ```

eller på DEC ULTRIX

```
f90 prog.f -lnag -lfor -lutil -li -lots
```

På DEC UNIX är `-lfor -lutil -lots` tillgängliga, men inget av dem behövs.

Observera i vilken precision NAG-biblioteket finns på just Din maskin! Gör Du fel med precisionen blir resultatet oftast fullständigt galet.

Lokalt gäller att NAG-biblioteket ej finns på Sun-systemet vid MAI.

- **Deluppgift c) Bosses funktion**
  Som ett alternativ till NAG-biblioteket har jag gjort i ordning en alternativ laborationsuppgift, som utnyttjar ett minibibliotek som jag själv gjort.

  Skriv ett program för tabellering av Bosses funktionen $Bo(x)$. Använd därvid biblioteket `libbosse.a` på kursbiblioteket. Skriv gärna huvudprogrammet så att det ber om det intervall för vilket funktionen önskas beräknad. Mata ut en tabell med x och $Bo(x)$ för $x = 0.0, 0.1, ..., 1.0$. Se till att utmatningen ser snygg ut! Provkör! Undersök även vad som händer vid argumenten $+800$ och -900!

  Aktuell Bosse-funktion heter `BO` och har argumenten `X` och `IFAIL` och i denna ordning, där `X` är argumentet för Bosse-funktionen i enkel eller dubbel precision, och där `IFAIL` är en felparameter (heltal). Denna senare sättes lämpligen till 1 vid ingången och avläses vid återhoppet. Om den då är noll så är allt OK, om den är 1 så är argumentet för stort, om den är 2 så är argumentet för litet. Denna felhantering är samma som hos NAG, om `IFAIL` är 0 vid ingången kommer programmet vid ett fel att stanna med en felutskrift från felhanteringsfunktionen, och aldrig komma tillbaks till Ditt program!

  Om Bosses bibliotek är installerat på normalt vis länkar man in det med kommandot

  ```
  f77 prog.f -L$KURSBIB -lbosse
  ```

  där `prog.f` är Ditt program.

  Om Du däremot använder Fortran 90 men Du fortfarande har Bosses bibliotek i Fortran 77 måste man kompilera och länka in erforderliga bibliotek med ett implementationsberoende kommando, se ovan i deluppgift b). För att göra livet lättare har jag dock gjort även en Fortran 90 version av mitt bibliotek, vilket användes utnyttjande

  ```
  f90 prog.f90 -L$KURSBIB -lbosse90
  ```

  Anmärkning 1. Notera att på vissa system måste man kasta om ordningen på filkatalogen och biblioteket, som

  ```
  f90 prog.f90 -lbosse90 -L$KURSBIB
  ```

  Anmärkning 2. Observera i vilken precision Bosses bibliotek finns på just Din maskin! Gör Du fel med precisionen blir resultatet oftast fullständigt galet.

  Lokalt gäller att Bosses bibliotek är i dubbel precision på Sun-systemet.

F.4 Labb 4, Fakultet och Runge-Kutta

Detta är laborationsuppgift nummer fyra, och den består i att skriva två små program i programspråket Fortran. Nu duger dock inte längre Fortran 77, utan egenskaper som kom först med Fortran 90 behövs.

- **Deluppgift a) Fakultet**
  Skriv en rekursiv funktion i Fortran 90 med tillhörande huvudprogram för beräkning av fakulteten. Provkör och beräkna 10! en gång till (jämför med laboration 3 a, även här i laboration 4 a skall heltal användas).

- **Deluppgift b) Runge-Kutta**
  Skriv ett program i Fortran 90 för lösning av ett system av ordinära differentialekvationer med Runge-Kuttas klassiska metod. Givet är systemet

```
y'(t) = z                          y(0) = 1
z'(t) = 2*y*(1+y^2) + t*sin(z)     z(0) = 2
```

  Beräkna en approximation till $y(0.2)$ genom att använda steglängden 0,04. Upprepa med steglängderna 0,02 och 0,01. Låt respektive derivata-funktioner finnas som interna (lokala) funktioner (à la Fortran 90). Det är således ej tillåtet att använda vanliga (externa) funktioner. Startvärden på t, y och z skall ges interaktivt, liksom steglängden h.

  Notera att det av mig införda förbudet att i just denna laboration använda externa funktioner försvårar användning av underprogram för utförande av Runge-Kutta stegen. Enda syftet med detta förbud är att tvinga till övning i lokala funktioner.

F.5 Labb 5, Linjärt ekvationssystem och filhantering

Detta är laborationsuppgift nummer fem. Den består i att självständigt i Fortran 90/95 skriva ett huvudprogram och ett antal underprogram, samtliga i dubbel precision, för att lösa ett vanligt förekommande beräkningsproblem. Problemet går ut på att lösa ett linjärt ekvationssystem $Ax = b$, utnyttjande en färdig rutin för det numeriska arbetet. Minst två laborationstillfällen åtgår normalt för att lösa denna uppgift på ett tillfredsställande sätt.

Kommentarer:

- Det är meningen att huvudprogrammet skall ha en meny (gärna utnyttjande `SELECT CASE`) där man väljer vilket av nedanstående underprogram som skall utföras. När ett underprogram har exekverats kommer man tillbaks till menyn för ett nytt val.

- Huvudprogrammet utnyttjar `LASMAT, LASVEK, MATIN` och `VEKIN` för att få in matrisen `A` och vektorn `b`, anropar lösningsprogrammet `LOS` och skriver ut matris `A`, högerled `b` och lösning `x` med hjälp av `MATUT` och `VEKUT`.

- Programmet skall kontrollera att man exempelvis inte försöker skriva ut en lösning med `VEKUT` förrän en sådan har beräknats med `LOS`.

- Om dimensionen på matrisen och vektorn ej överensstämmer skall programmet (underprogrammet eller huvudprogrammet) ge en felutskrift, liksom om den givna rutinen upptäcker singularitet för matrisen eller något annat problem.

- Dimensioneringen skall utnyttja de dynamiska möjligheterna i Fortran 90/95. En lämplig metod kan vara att utnyttja pekare vid deklarationen, se avsnitt 3.3.2, sid 37, eller (vilket är att föredraga) att utnyttja en modul, se slutet av nämnda avsnitt, sid 38.

- Programmet skall efter avslutad beräkning och utmatning gå tillbaks till startpunkten och begära ny matris och/eller nytt högerled. Det skall vidare vara möjligt att utnyttja huvudprogrammet för att (utan förnyad manuell inmatning) utnyttja redan på filer undanlagrade matriser och vektorer. I uppgiften ingår bland annat att göra ytterligare nödvändiga specifikationer, testa rimligheten av inlästa värden, samt att hitta på de testmatriser och testvektorer som skall utnyttjas för att testa programmet.

1. Det första underprogrammet `LASMAT` skall interaktivt läsa in flyttalsvärdena i en kvadratisk matris. Programmet skall fråga efter dimension samt ge användaren möjlighet att individuellt mata in enbart de värden som är skilda från noll. Därefter skall programmet lagra undan den fulla matrisen (den eventuella glesa egenskapen får inte utnyttjas vid lagringen) på en fil utnyttjande maximal noggrannhet och minimalt lagringsutrymme. Underprogrammet skall fråga användaren efter namnet på filen, men filtypen skall vara `.mat`. Användaren får ej ge filtypen.

   Kommentarer till `LASMAT`:

   - Det är lämpligt att använda två olika sätt att läsa in matrisen, nämligen dels en metod som läser in matrisen rad för rad, dels en som läser in rad, kolumn och värde för de nollskillda elementen. Den första metoden är bra för att läsa in de båda testmatriserna!
   - Kravet med "maximal noggrannhet och minimalt lagringsutrymme" innebär att man skall använda oformatterad utmatning, jämför med inledningen till kapitel 6 om in- och utmatning samt sektion 16.3, sid 143.

2. Det andra underprogrammet `LASVEK` skall interaktivt läsa in flyttalsvärdena i en vektor. Programmet skall fråga efter dimension samt ge användaren möjlighet att individuellt mata in enbart de värden som är skilda från noll. Därefter skall programmet lagra undan den fulla vektorn (den eventuella glesa egenskapen får inte utnyttjas vid lagringen) på en fil utnyttjande maximal noggrannhet och minimalt lagringsutrymme. Underprogrammet skall fråga användaren efter namnet på filen, men filtypen skall vara .vek. Användaren får ej ge filtypen.

3. Det tredje underprogrammet `MATIN` läser en matris från filen med angivet namn och lagrar den så att den är tillgänglig i det anropande programmet. Användaren får ej ge filtypen.

4. Det fjärde underprogrammet `VEKIN` läser en vektor från filen med angivet namn och lagrar den så att den är tillgänglig i det anropande programmet. Användaren får ej ge filtypen.

5. Det femte underprogrammet `MATUT` skriver ut en matris på papper (via en fil), med rader och kolumner på normalt sätt om dimensionen är högst 10, och på ett begripligt sätt annars. Radskrivarens 132 positioner i bredd skall utnyttjas väl. Notera att godtyckliga flyttal skall kunna skrivas ut. Utskrift på papper sker normalt via en fil, som sedan skrives ut med ett lpr under UNIX eller motsvarande kommando under Windows.

6. Det sjätte underprogrammet `VEKUT` skriver ut en vektor på papper (via en fil), helst dock i transponerad (radvis) form. Notera att det finns både högerled och lösning att skriva ut! Man behöver därför komplettera denna rutin med information om vilken vektor som skrives ut.

7. Det sjunde underprogrammet `LOS` löser ekvationssystemet genom att anropa den givna rutinen `LOES_LIN_EKV_SYSTEM`, som är i dubbel precision. Denna rutin får ej ändras och får ej heller läggas in i en modul.

Obligatoriska testexempel är följande

$$A = \begin{pmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix} \qquad b = \begin{pmatrix} -359 \\ 281 \\ 85 \end{pmatrix}$$

och

$$C = \begin{pmatrix} 2 & 3 & 0 & 0 \\ 0 & 0 & 1.8 & 10.1 \\ 0.2 & 4.3 & 5 & 0 \\ 0 & 0.8 & 7 & 7 \end{pmatrix} \qquad d = \begin{pmatrix} 1 \\ 11 \\ 42 \\ 109 \end{pmatrix}$$

Inlämning skall ske av fullständig programlistning, körexempel med gles matris, resultatutskrift av matris med dimensionerna 3, 4, 10 och 12.

Tips:

1. Testa även på det triviala fallet med dimensionen 1 på både matris och vektor.

2. Den givna rutinen får ej ändras. Anropet måste därför utnyttja ett så kallat gränssnitt, `INTERFACE`.

3. Utför gärna den första kompileringen av programmet på en programenhet i taget, för att inte bli översvämmad med felutskrifter. När alla programenheterna kan kompileras var för sig kan man förfara på fyra sätt. Jag antar att huvudprogrammet finns på filen `huvud.f90`, subrutinen `LASMAT` på filen `lasmat.f90`, osv, samt ett eventuellt fullständigt program på filen `labb5.f90`.

1. Kompilera alla rutinerna var för sig men samtidigt och med samma kommando,

```
f90 huvud.f90 lasmat.f90 lasvek.f90 ... loes.f90
```

Exekvera med

```
a.out
```

2. Kompilera alla rutinerna var för sig helt separat och utnyttja objektmodulerna vid länkning,

```
f90 -c huvud.f90
f90 -c lasmat.f90
f90 -c lasvek.f90
...
f90 -c loes.f90
f90 huvud.o lasmat.o lasvek.o ... loes.o
```

Exekvera med

```
a.out
```

Vid ändring av en rutin behöver bara den kompileras om, varvid länkning av samtliga följer. Denna metod går mycket fortare!

3. Kombinera alla filerna till en stor fil och kompilera,

```
f90 labb5.f90
```

Exekvera med

```
a.out
```

Denna metod går mycket långsamt!

4. Kompilera alla rutinerna var för sig men utnyttjande kommandot `make`. Exekvera med `labb5`. Detta är en överkurs i UNIX! En lämplig `makefile` finns nedan och på kursbiblioteket. Detta är den bästa metoden!

```
labb5:  huvud.o lasmat.o lasvek.o matin.o vekin.o \
                matut.o vekut.o los.o loes.o
        f90 -o labb5 huvud.o lasmat.o lasvek.o matin.o \
                vekin.o matut.o vekut.o los.o loes.o
huvud.o: huvud.f90
        f90 -c huvud.f90
lasmat.o: lasmat.f90
        f90 -c lasmat.f90
lasvek.o: lasvek.f90
        f90 -c lasvek.f90
matin.o: matin.f90
```

```
        f90 -c matin.f90
vekin.o: vekin.f90
        f90 -c vekin.f90
matut.o: matut.f90
        f90 -c matut.f90
vekut.o: vekut.f90
        f90 -c vekut.f90
los.o:  los.f90
        f90 -c los.f90
loes.o: loes.f90
        f90 -c loes.f90
```

Ovanstående utnyttjas på så sätt att man flyttar filen `makefile` till aktuell area och när man vill kompilera skriver man `make` i terminalfönstret. Då kompileras enbart de filer som ändrats sedan föregående `make` (eller alla om det är första gången), samt länkas alla programenheterna samman till ett körklart program, i detta fall med namnet `labb5`. Om man har valt andra filnamn än ovan måste naturligtvis filen `makefile` justeras, liksom om kompilatorn har annat namn (här `f90`).

I princip fungerar `make` så att om det som står efter kolon har en senare tidsmarkering än det som står före, så utföres den påföljande raden. Det finns en bok [25] som beskriver `make`.

Anm. Bakåtstrecket \ markerar fortsättningsrad i UNIX.

4. Den rutin `LOES_LIN_EKV_SYSTEM` som skall användas i dubbel precision ligger på kursbiblioteket i enkel precision under namnet `loes1.f90`, och i dubbel precision under namnet `loes.f90` och ser ut som följer.

```
SUBROUTINE LOES_LIN_EKV_SYSTEM(A, X, B, FEL)
  IMPLICIT NONE
  ! Fältdeklarationer
  DOUBLE PRECISION, DIMENSION (:, :), INTENT (IN)  :: A
  DOUBLE PRECISION, DIMENSION (:),    INTENT (OUT) :: X
  DOUBLE PRECISION, DIMENSION (:),    INTENT (IN)  :: B
  LOGICAL, INTENT (OUT)                            :: FEL

  ! Arbetsarean M är A utvidgad med B
  DOUBLE PRECISION, DIMENSION (SIZE (B), SIZE (B) + 1) :: M
  INTEGER, DIMENSION (1)                                :: MAX_LOC
  DOUBLE PRECISION, DIMENSION (SIZE (B) + 1)            :: TEMP_ROW
  INTEGER                                               :: N, K

        ! Initiera M
        N = SIZE (B)
        M (1:N, 1:N) = A
        M (1:N, N+1) = B

        ! Triangulariseringsfas
        FEL = .FALSE.
        TRIANG_SLINGA: DO K = 1, N - 1
```

```
               ! Pivotering
               MAX_LOC = MAXLOC (ABS (M (K:N, K)))
               IF ( MAX_LOC(1) /= 1 ) THEN
                   TEMP_ROW (K:N+1 ) = M (K, K:N+1)
                   M (K, K:N+1) = M (K-1+MAX_LOC(1), K:N+1)
                   M (K-1+MAX_LOC(1), K:N+1) = TEMP_ROW (K:N+1)
               END IF

               IF (M (K, K) == 0.0D0) THEN
                  FEL  = .TRUE.     ! Singulär matris A
                  EXIT TRIANG_SLINGA
               ELSE
                  TEMP_ROW (K+1:N) = M (K+1:N, K) / M (K, K)
                  M (K+1:N, K+1:N+1) = M (K+1:N, K+1:N+1)  &
                     - SPREAD( TEMP_ROW (K+1:N), 2, N-K+1) &
                     * SPREAD( M (K, K+1:N+1), 1, N-K)
                  M (K+1:N, K) = 0 ! Dessa värden används ej
               END IF
         END DO TRIANG_SLINGA

         IF (M (N, N) == 0.0D0) FEL  = .TRUE.  ! Singulär matris A

         ! Återsubstitution
         IF (FEL) THEN
            X = 0.0D0
         ELSE
            X (N) = M (N, N+1)/ M (N, N)
            DO K = N-1, 1, -1
               X (K) = M (K, N+1) - SUM (M (K, K+1:N) * X (K+1:N))
               X (K) = X (K) / M (K, K)
            END DO
         END IF

END SUBROUTINE LOES_LIN_EKV_SYSTEM
```

Anm: Rutinen ovan utnyttjade i en tidigare version summation av tomma vektorer, dvs i princip `SUM(vektor(N+1:N))`, där vektorn är dimensionerad att ha högst `N` element, dvs vi har indexöverskridande. Regeln för summationsrutinen `SUM` är att i sådant fall (på grund av att summation baklänges är begärd) att summera inget element, och således få värdet noll. Jämför med `DO`-slingan baklänges. Om man kompilerar med indexkontroll, till exempel med `f90 -C loes.f90` erhålles naturligtvis felutskrift om otillåtet index vid exekvering. Indexkontroll kan därför ej användas framgångsrikt på den tidigare versionen:

```
         IF (FEL) THEN
            X = 0.0D0
         ELSE
            DO K = N, 1, -1
               X (K) = M (K, N+1) - SUM (M (K, K+1:N) * X (K+1:N))
!              Ovanstående summation är tom om K = N
```

```
            X (K) = X (K) / M (K, K)
         END DO
      END IF
```

Laboration 5, Deluppgift a)

Detta är laboration nummer 5 som ovan, men kravet är dessutom att utnyttja pekare vid deklarationen av fältet för matrisen, i enlighet med metoden i avsnitt 3.3.2, sid 37.

Laboration 5, Deluppgift b)

Detta är laboration nummer 5 som ovan, men kravet är dessutom att utnyttja en modul med ett allokerbart och sparat fält vid deklarationen av fältet för matrisen, i enlighet med metoden som nämns sist i avsnittet på sid 38. Det är här inte tillåtet att använda pekare.

Laboration 5, Deluppgift c)

Detta är laboration nummer 5 som ovan, men i stället för att utnyttja den där givna rutinen `loes.f90` för lösning av ekvationssystemet skall valfri rutin från NAG-biblioteket i Fortran 77 utnyttjas. NAG:s Webb plats kan utnyttjas. Man länkar normalt med

```
f90 prog.f90 -lnag
```

När NAG-biblioteket är kompilerat med Fortran 77, måste man ibland kompilera och länka in erforderliga bibliotek med ett implementationsberoende kommando. Försök dock gärna först utan några extra bibliotek.

Vid deklarationen av fältet för matrisen är det i detta fall valfritt att utnyttja någon av metoderna i a) och b) ovan, dvs utnyttjande antingen en pekare eller en modul med allokerat och sparat fält, eller eventuellt att hitta en egen fungerande metod.

Vid eventuell användning av `makefile` måste denna naturligtvis justeras.

Laboration 5, Deluppgift d)

Detta är laboration nummer 5 som ovan, men i stället för att utnyttja den där givna rutinen `loes.f90` för lösning av ekvationssystemet skall valfri rutin från NAG-biblioteket i Fortran 90 utnyttjas. NAG:s Webb plats kan utnyttjas. Man länkar normalt med

```
f90 prog.f90 -lnagfl90
```

Vid deklarationen av fältet för matrisen är det i detta fall valfritt att utnyttja någon av metoderna i a) och b) ovan, dvs utnyttjande antingen en pekare eller en modul med allokerat och sparat fält, eller eventuellt att hitta en egen fungerande metod.

Vid eventuell användning av `makefile` måste denna naturligtvis justeras.

Laboration 5, Deluppgift e)

Detta är laboration nummer 5b som ovan, men dessutom gäller att förbudet att utnyttja den glesa egenskapen hos matrisen hävs. I stället är det nu ett krav att utnyttja den glesa egenskapen hos matrisen, men bara i underprogrammen `LASMAT` och `MATIN`. Detta kan ske som i kapitel 9, avsnitt 9.7, sid 94, med datatypen `NONZERO`. Filen (matrisen) skall nu lagras med filtypen `.gls`, även nu utnyttjande maximal noggrannhet och minimalt lagringsutrymme.

F.6 Labb 6, Fakultet

Detta är laborationsuppgift nummer sex, och den består i att skriva ett antal varianter av ett program för beräkning av fakulteten.

Skriv en funktion i Fortran med tillhörande huvudprogram för beräkning av fakulteten. Skriv gärna huvudprogrammet så att det ber om värden på de heltal för vilket fakulteten önskas beräknad. Provkör med olika heltal som indata.

1. Använd genomgående heltal i funktionen. Notera vilket som är det största heltal som fungerar!

2. Använd nu flyttal `REAL` i funktionen, inargumentet skall dock behållas som heltal. Notera vilket som är det största heltal som fungerar! Det finns nu två svar, dels det största tal som ger helt rätt resultat, dels det största som ger approximativt rätt svar. Man kan även behöva skilja på exakt rätt svar i binär form eller i decimal form!

3. Upprepa med flyttal i dubbel precision, `DOUBLE PRECISION` i funktionen.

4. Upprepa med flyttal i fyrdubbel precision, om du har tillgång till detta.

F.7 Labb 7, Bessel-funktionen

Detta är laborationsuppgift nummer sju, och den består i att skriva ett program för beräkning av Bessel-funktionen $J_0(z)$, se även tabellverket [32, sid 266].

Besselfunktioner är lösningar till Bessels differentialekvation

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} + (z^2 - \nu^2)y = 0$$

som har fått sitt namn efter Friedrich Wilhelm Bessel (1784 - 1846), som var den förste att studera lösningarna till dem som ett led i hans forskning inom celest mekanik. Konstanten $\nu = 0$ anger det man kallar ordningen hos ekvationen. Man kan visa att i fallet $\nu = 0$ utgör potensserien

$$y(z) = 1 + \sum_{1}^{\infty} \frac{(-1)^m z^{2m}}{2^{2m}(m!)^2}$$

en av de linjärt oberoende lösningarna, den betecknas $J_0(z)$.

Potensserien är konvergent i hela det komplexa talplanet, men på grund av att kancellation kan uppträda nöjer vi oss med beräkning på skivan $|z| <= 1$.

Uppgiften är nu att i Fortran skriva en funktion i enkel precision som beräknar $J_0(z)$ för sådana argument, och som ser till att varna användaren om argumentet blir för stort (till beloppet).

Vidare skall funktionen skrivas så att den är generisk, det vill säga en version för komplexa argument och en för reella argument, aktuell version väljes automatiskt. Lämpligt avbrottsvillkor vid summationen är då den tillkommande termen inte ger något ytterligare bidrag till summan. Funktionen skall läggas i en modul, som användes av nedan nämnda huvudprogram.

Skriv ett huvudprogram som använder funktionen för att generera en tabell över $J0(z)$ på de tre linjerna $Re(z) = -0,5; Re(z) = +0,5$ och $Im(z) = 0$ med en lämplig stegning.

Program och körningsresultat skall redovisas. Du skall dessutom ge två bra skäl till varför man **inte** bör använda en funktion (eller subrutin) för beräkning av fakulteten.

Laboration 7 b innebär att komplettera programmet till att inte bara klara enkel precision, utan även dubbel och fyrdubbel. Programmet skall bestå av modul(er) och ett huvudprogram, inga andra programenheter.

F.8 Labb 8, Runge-Kutta

Detta är laborationsuppgift nummer åtta, och den består i att skriva ett program i Fortran för lösning av ett system av ordinära differentialekvationer med Runge-Kuttas klassiska metod. För systemet nedan, där ett streck över en bokstav markerar att det är en vektor,

$$\frac{d\bar{y}}{dt} = \bar{f}(t, \bar{y}), \quad \bar{y}(t_0) = \bar{y}_0$$

lyder Runge-Kuttas metod

$$\begin{aligned}
\bar{k}_1 &= h \cdot \bar{f}(t_n, \bar{y}_n) \\
\bar{k}_2 &= h \cdot \bar{f}(t_n + h/2, \bar{y}_n + \bar{k}_1/2) \\
\bar{k}_3 &= h \cdot \bar{f}(t_n + h/2, \bar{y}_n + \bar{k}_2/2) \\
\bar{k}_4 &= h \cdot \bar{f}(t_n + h, \bar{y}_n + \bar{k}_3) \\
\bar{y}_{n+1} &= \bar{y}_n + (\bar{k}_1 + 2\bar{k}_2 + 2\bar{k}_3 + \bar{k}_4)/6 \\
t_{n+1} &= t_n + h
\end{aligned}$$

Uppgiften är att skriva en subrutin med inparametrarna `tn, yn, h` och `f` samt utparametrarna `tnp1` och `ynp1`. Här är $\bar{y}$ en vektor och $\bar{f}(t, \bar{y})$ en vektorvärd funktion av det skalära argumentet t och vektorn $\bar{y}$.

Tillämpa ditt program på två valfria av nedanstående tre problem! Beräkna approximationer till $\bar{y}(0.2)$ genom att använda steglängderna $h = 0.04; 0.02$ och 0.01. Gör gärna dessutom en manuell Richardson-extrapolation.

$$(1) \qquad \frac{dy}{dt} = t \cdot \sin y, \qquad y(0) = 1$$

$$(2) \qquad \frac{d\bar{y}}{dt} = \begin{pmatrix} y_2 \\ 2y_1 \cdot (1 + y_1^2) + t \cdot \sin y_2 \end{pmatrix}$$

$$\bar{y}(0) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\frac{d\bar{y}}{dt} = \begin{pmatrix} 1 & 2 & 3 \\ -2 & 1 & 2 \\ -3 & -2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{3}$$

$$\bar{y}(0) = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Program och körningsresultat skall redovisas.

Kommentarer:

- Det är naturligt att programmet kommer att bestå av ett huvudprogram, tre funktioner (för de tre högerleden i differentialekvationen) samt en subrutin (för Runge-Kutta beräkningen). Jag rekommenderar att de tre funktionerna läggs i en modul, och subrutinen i en annan modul.
- I subrutinen för Runge-Kutta behövs ett formellt argument för funktionen, som vi kan kalla `F(T,Y)`. Problemet här är att dimensionen på `Y` kan vara ett, två eller tre! Man behöver där ett `INTERFACE` som kan se ut ungefär så här

```
INTERFACE
FUNCTION F(T,Y) RESULT (FUT)
    REAL                       :: T
    REAL, DIMENSION(:)         :: Y
    REAL, DIMENSION(SIZE(Y)) :: FUT
END FUNCTION F
END INTERFACE
```

  Programmet kommer då att vid exekveringen känna av den aktuella dimensionen av `Y` och tilldela samma dimension till funktionen `F` (och variabeln `FUT`). De tre funktionerna måste skrivas på motsvarande sätt, även den första funktionens `Y` måste skrivas som en vektor (med verklig längd 1) och inte som en skalär.

- Om man inte använder sig av moduler behövs två `INTERFACE` till!

F.9 Labb 9, Egna datatyper

Del 1.
Skriv ett program som definierar två datatyper. Den första innehåller relevant personinformation, som tilltalsnamn, förnamnsinitial, efternamn, kön, ålder, yrke, och övrigt som Du finner lämpligt. Den andra innehåller en adress i lämplig form. Lagra dina egna data i dessa datatyper och skriv ut på ungefär följande sätt

Mitt namn är Bo G. Einarsson. Jag är en 68 år gammal manlig politiker, och jag bor på Ekholmsvägen 249 i Linköping.

Del 2.
Använd dessa två datatyper för att skapa en ny datatyp "familj", som innehåller namnen på fader, moder och två barn, och deras gemensamma adress. Skapa test-data (helst fiktiva) och skriv ut en familjesammanfattning. Notera speciellt problemet med avstavning och ny rad!

Anmärkning: Egendefinierade datatyper diskuteras i bokens avsnitt 9.7 på sid 94, 11.2 på sid 102 och B.12 på sid 170.

Du får använda en statisk största längd på alla textsträngar. Textsträngar med variabel längd är ett speciellt tillägg till Fortran, se [21, Part 2]. Vid utmatning får däremot inte onödigt många blanka skrivas ut. Funktionen `LEN_TRIM` ger längden av en textsträng utan eventuell(a) avslutande blank(a).

Du får vidare antaga att tilltalsnamnet är första förnamnet och att det bara finns ett extra förnamn vars initial skall med. Program och körningsresultat skall redovisas.

F.10 Labb 10, Egen sinus

Skriv ditt eget program i form av en funktion för beräkning av sinus-funktionen för små värden på beloppet av argumentet `X`, dvs för $-\pi/2 < x < \pi/2$. Använd Maclaurin-serier. Önskad noggrannhet skall anges som ett frivilligt andra argument enligt `MIN_SIN(X, NOGR=Y)`. Precisionen skall kunna vara enkel, dubbel, eller fyrdubbel. Funktionen skall skrivas generisk. Om det andra argumentet saknas skall en lämplig noggrannhet väljas!

Skriv ett testprogram och jämför med den inbyggda `SIN(X)`. Precisionen skall kunna vara enkel, dubbel, eller fyrdubbel. Utvidga programmet till att klara godtyckliga reella värden på `X`, men i detta fall utan samma höga krav på noggrannheten.

Du skall dessutom ge två bra skäl till varför man **inte** bör använda en funktion (eller subrutin) för beräkning av fakulteten.

F.11 Labb 11, Instabil algoritm

Denna laboration analyserar en instabil algoritm. Skriv ett program för att beräkna integralerna $I_0, I_1, ..., I_{20}$, där

$$I_n = \int_0^1 x^n e^{-x} dx$$

med rekursionsformeln (som enkelt visas med partiell integration)

$$I_0 = 1 - 1/e$$

$$I_n = -1/e + n \cdot I_{n-1} \qquad (n = 1, 2, 3, ...)$$

Skriv programmet så att det använder antingen enkel eller dubbel precision, beroende på hur modulen som definierar precisionen ser ut.

Betrakta resultatet och jämför med enkla analytiska uppskattningar!

Det finns ett tricks för att beräkna dessa integraler korrekt! Minns du det? Om man startar med ett begynnelsevärde med sex signifikanta siffror kan vi få ett nonsensresultat. Om vi däremot startar med ett värde med inga signifikanta siffror kan vi få sex signifikanta siffror i resultatet! Skriv även detta program!

F.12 Labb 12, Intervallaritmetik

I kompilatorn från Sun finns ett inbyggt system för intervallaritmetik. Ett exempel från [11, avsnitt 9.5] på användning av intervallaritmetik vid Newtons metod finns i kursbiblioteket på filen `ia_newton.f90`. Din uppgift är att modifiera detta program till att bestämma samtliga nollställen till funktionen

$$f(x) = \exp(-10 \cdot x^2) + \cos(\pi * x)$$

i intervallet [-10; 10]. Använd därefter den metodoberoende feluppskattningen till att verifiera de två till beloppet minsta rötterna. Kompilering sker med kommandot

```
f95 -xia fil.f90
```

för att få med Sun:s tillägg för intervallaritmetik. Ytterligare information finns på Sun:s sida. Jag rekommendar att de båda noggrannhetsparametrarna `f_eps` och `root_eps` sänks från nuvarande 10^{-5} till högst 10^{-10}.

F.13 Labb 13, Binär stjärna

Ljusstyrkan hos en binär stjärna varierar på följande sätt. Vid tiden $t = 0$ är dess magnitud 2.5, och den förblir vid denna nivå till $t = 0.9$ dagar. Dess magnitud är sedan bestämd av formeln

$$3.355 - \ln(1.352 + \cos(\pi(t - 0.9)/0.7))$$

till $t = 2.3$ dagar. Dess magnitud är sedan 2.5 till $t = 4.4$ dagar, och bestäms sedan av formeln

$$3.598 - \ln(1.998 + \cos(\pi(t - 4.4)/0.4))$$

till $t = 5.2$ dagar. Dess magnitud är sedan 2.5 till $t = 6.4$ dagar, efter vilket cykeln upprepas med en period av 6.4 dagar.

Skriv en funktion som har tiden t som indata och ljusstyrkan (magnituden) vid denna tidpunkt som utdata. Skriv ett huvudprogram som plottar ljusstyrkan som en funktion av tiden för intervallet $t = 0$ till $t = 25$ dagar.

Anm. Exemplet kan förefalla löjligt, men strålningen från dubbelstjärnan[1] Algol β i stjärnbilden Perseus har faktiskt ett liknande utseende, periodisk med 68 h 50 min, se [33, figur 40].

[1] En dubbelstjärna kan enkelt beskrivas som två närbelägna stjärnor som roterar "kring varandra". Om ur observatörens synpunkt båda syns uppfattas de som en enda ljusstark stjärna, men om den ena är framför den andra når inte ljuset från den bakre fram och de båda upplevs som en svagare stjärna.

Om du saknar tillgång till plottning i Fortran kan du i stället använda ett annat paket för detta, till exempel Matlab. Råd om plottning med Matlab från Fortran ges i slutet av laboration F.16.

F.14 Labb 14, Logisk funktion med textsträngs-argument

Skriv en logisk funktion som har två textsträngs-argument av typ `CHARACTER`, och som returnerar värdet `.TRUE.` om det andra argumentet finns i det första, och `.FALSE.` i annat fall. Således, om funktionen kallad `WITHIN` anropas med

```
WITHIN("Jag testar", "test")
```

skall `.TRUE.` returneras, medan

```
WITHIN("Jag testar", "Test")
```

skall returnera `.FALSE.`

Använd gärna en eller flera av de inbyggda ("intrinsic") funktionerna! Testa din funktion med ett huvudprogram som har inmatning av textsträngar från tangentbordet och använder resultatet av funtionen för att ge en av följande utskrifter

```
Frasen 'test' finns i "Jag testar"
Frasen 'Test' finns inte i "Jag testar"
```

F.15 Labb 15, Skalär- och vektor-produkter

Skriv två generiska funktioner för skalärprodukten och vektorprodukten av två tre-dimensionella vektorer. Som argument skall flyttal i enkel, dubbel eller fyr-dubbel precision accepteras, men för enkelhets skull bara samma typ vid ett specifikt anrop.

Använd dessa för att skriva en tredje generisk funktion som beräknar den skalära trippel-produkten, definierad som

```
[abc] = a . (b x c)
```

Skriv sedan ett huvudprogram som läser in de tre vektorerna a, b och c och som skriver ut skalärprodukten $a \cdot b$ och vektorprodukten $b \times c$ samt den skalära trippelprodukten $[abc]$.

Testa på följande vektorer:

```
a = (  1 10  20 )'
b = (  5  8   9 )'
c = ( 12 36 108 )'
```

och

```
a = (  1 sqrt(2)  sqrt(3) )'
b = (  5    8       9     )'
c = (  2    3       8     )'
```

där `sqrt` betyder kvadratroten.

Anm. Vektorprodukten definieras av

```
c = a x b

c1 = a2 b3 - a3 b2
c2 = a3 b1 - a1 b3
c3 = a1 b2 - a2 b1
```

F.16 Labb 16, Ekologisk differentialekvation

I denna laboration skall vi betrakta ett system av andra ordningen av differentialekvationer av första ordningen. Systemet liknar [15, uppgift 10.15], det ekologiska problemet med harar, bytesdjur, och rävar, rovdjur.

$$u'(t) = u(t) \cdot [A - 0.1 \cdot v(t)] + 0.015 \cdot t, \quad u(0) = \alpha$$

$$v'(t) = v(t) \cdot [0.02 \cdot u(t) - 1] + 0.0075 \cdot t, \quad v(0) = \beta$$

Samtliga storheter är normaliserade, varför den verkliga tidsskalan är okänd, vi kan dock kalla tidsenheten för år. Vi betecknar antalet harar med $u(t)$ och antalet rävar med $v(t)$. Tänk Dig gärna antalen som tusental!

Lösning av detta system av differentialekvationer skall nu programmeras i Fortran, utnyttjande Runge-Kuttas metod och dubbel precision. Förutom beräkning av antalet harar och rävar skall programmet beräkna det fel i respektive antal som beror på att parametern `A` inte är exakt 1 utan i stället `1 + dellta`. Själva beräkningen skall ske med `A = 1` och `dellta = 0.02`. Aktuell formel blir analog till den i svaret till 10.15 b, dvs (kolla den gärna):

```
DELTAunp1 = [1 + h*(1 - 0.1*vn)]*DELTAun
             - 0.1*h*un*DELTAvn + h*un*dellta

DELTAvnp1 = [1 + h*(0.02*un - 1)]*DELTAvn
             + 0.02*h*vn*DELTAun

DELTAu0 = 0
DELTAv0 = 0
dellta = 0.02
```

Den avser egentligen Eulers metod, men Du får använda den här vid felberäkningen, för att minska programmeringsarbetet.

Anmärkning: Att jag kallat felet ovan för `dellta` beror på att *delta* är ett reserverat namn vid fixpunktsaritmetik i Ada.

Din uppgift är att köra programmet med två val av startvärdena α (antalet tusental harar vid starttidpunkten) och β (antalet tusental rävar vid starttidpunkten). Dessa skall båda väljas mellan 20 och 30.

Gör två körningar, och välj ett lämpligt värde på steglängden h. Du skall i båda fallen gå från $t = 0$ till $t = 50$.

För varje fall skall Du plotta vardera $u \pm \Delta u$ och $v \pm \Delta v$ som funktion av t. Dessutom skall Du separat plotta v som funktion av u. Den senare plotten kallas fasplanet.

Plottningen kan ske antingen direkt i Fortran (kräver i så fall tillgång till plottrutiner), eller genom att Fortran-programmet genererar en fil som kan utnyttjas av Matlab för plottning. Man kommer åt till exempel den andra kolumnen av en matris y genom att skriva $y(:, 2)$. En möjlighet är att skriva ut $t, u, v, \Delta u$ och Δv som fem separata vektorer, som klistras (målas) in i Matlab efter till exempel `x = [`, avsluta med `]`. Alternativt kan man i Fortran skapa en fil med namnet `filnamn.m` som innehåller erforderlig matris eller vektorer och som kan hämtas in i Matlab med kommandot `filnamn` där. Som vanligt är det lämpligt att undvika `filnamn` som användes för olika funktioner i Matlab!

Kommandot `load filnamn.mat` arbetar däremot normalt med binära filer som har filändelsen `.mat` och passar bäst ihop med filer skapade i Matlab med kommandot `save filnamn.mat`. Kommandot `load filnamn.txt` fungerar däremot utmärkt i vårt fall.

Redovisning sker genom inlämning av programlistning och plottbilder samt svar på följande frågor.

1. Hur varierar antalet harar och rävar i förhållande till varandra?

   $$\alpha = \ldots\ldots\ldots\ldots \qquad \beta = \ldots\ldots\ldots\ldots$$

2. Hur varierar antalet harar och rävar i förhållande till varandra?

   $$\alpha = \ldots\ldots\ldots\ldots \qquad \beta = \ldots\ldots\ldots\ldots$$

3. Vad händer om man byter tecken på de båda termerna `0.015*t` och `0.0075*t`?

4. Vad händer om man tar bort de båda termerna `0.015*t` och `0.0075*t`?

Bilaga G

Ordförklaringar

G.1 Fält

- allokerbart fält (allocatable array)
  Fältet deklareras som `ALLOCATABLE` med en viss typ och en viss rang. Det kan sedan tilldelas utrymme med `ALLOCATE` till ett bestämt omfång, och av-allokeras med `DEALLOCATE`.

- automatiskt fält (automatic array)
  Fält i underprogram där aktuellt fält är lokalt men parametrarna i dimensioneringen finns med bland de formella argumenten.

```
SUBROUTINE sub (i, j, k)
REAL, DIMENSION (i, j, k) :: x
```

- fält (array)
  Dimensionerad storhet.

- fält med antagen dimension (assumed-size array)
  Fältet har en variabel dimension liksom i Fortran 77 genom att både fältnamn och aktuell dimension finns med i argumentlistan, utom den sista dimensionen, vilken ges med en asterisk.

```
SUBROUTINE sub (a, na1, na2)
REAL, DIMENSION (na1,na2,*) :: a
```

  I Fortran 66 fanns inte begreppet "assumed-size array", men det simulerades genom att placera siffran "1" där stjärnan "`*`" är i Fortran 77. Denna sed bryter mot eventuell index-kontroll och är naturligtvis förbjuden av moderna kompilatorer, som NAG Fortran 90 kompilatorn. Många gamla program utnyttjar fortfarande denna metod att simulera dynamisk minnestilldelning.

- fält med antaget mönster (assumed-shape array)
  Fält i subprogram där aktuellt fält ej är lokalt men har en viss typ och en viss rang.

```
SUBROUTINE sub (a)
REAL, DIMENSION (:, :, :) :: a
```

  I detta fall kräves ett explicit gränssnitt i den anropande programenheten, förutom en explicit deklaration där av fältet `A`.

```
INTERFACE
    SUBROUTINE SUB (A)
       REAL, DIMENSION (:, :, :) :: A
    END SUBROUTINE SUB
END INTERFACE
```

- fältfunktion (array function)
  Funktion som opererar på ett fält och returnerar ett fält eller en skalär, eller opererar på en skalär och returnerar ett fält.

- fältsektion (array section)
  Del av ett fält (kan ha rang större än 1).

- fältuttryck (array assignment)
  Tilldelning av ett helt fält till ett annat är tillåtet om de båda fälten har samma mönster (liksom om uttrycket till höger är skalärt, i vilket fall alla element i fältet tilldelas skalärens värde).

- justerbart fält (adjustable array)
  Fältet har en variabel dimension liksom i Fortran 77 genom att både fältnamn och aktuell dimension finns med i argumentlistan.

```
SUBROUTINE sub (a, na)
REAL, DIMENSION (na) :: a
```

- ledande
  Vid endimensionella fält (vektorer) är den ledande dimensionen ointressant, vid tvådimensionella fält (matriser) är den ledande dimensionen den första av de båda dimensionernas omfång. Begreppet användes normalt inte vid fält av högre ordning, men avser då samtliga dimensioners omfång utom den sista.

- mönster (shape)
  Mönstret för ett fält består av rang och omfång.

- omfång (extent)
  Antalet element längs respektive dimension.

- rang (rank)
  Antalet dimensioner. VARNING: Ej den matematiska rangen.

G.2 Övrigt

- attribut (attribute)
  Vid deklaration av en variabel kallas tilläggs-egenskaper som `PARAMETER`, `PUBLIC`, `PRIVATE`, `INTENT`, `DIMENSION`, `SAVE`, `OPTIONAL`, `POINTER`, `TARGET`, `ALLOCATABLE`, `EXTERNAL` och `INTRINSIC` för attribut. Dessa kan även i stället ges som egna satser.

- `BLOCK DATA` programenhet (`BLOCK DATA` program unit)
  En `BLOCK DATA` programenhet innehåller definitioner (begynnelsevärden) som skall användas av de andra programenheterna. Ersättes nu av det generellare begreppet modul.

- extern (`EXTERNAL`)
  De funktioner och subrutiner som man själv skriver, eller som ingår i tillämpningsbibliotek. Motsats är inbyggd. Om externa funktioner eller subrutiner användes som argument vid anrop av något annat underprogram skall de deklareras `EXTERNAL`.

- funktion (function)
  En programenhet som returnerar ett funktionsvärde (eventuellt flera, dvs ett fält) i sitt namn och som har ett antal parametrar. Det är tillåtet med funktioner utan parametrar (argument), det är likaså tillåtet men ej tillrådligt att funktionen ändrar värdet på ett eller flera av argumenten.

- generisk (generic)
  En generisk funktion kan vara sådan att typen på argumentet ger samma typ på funktionen. Värdet `SIN(1.0D0)` är dessutom noggrannare än det av `SIN(1.0)`. Dessutom finns inbyggda generiska funktioner, till exempel `REAL()`, som kan ha argument av olika typer men alltid returnerar ett värde av en viss typ. För generiska subrutiner är det bara argumenten som anpassas.

- gränssnitt (interface)
  Eftersom de olika programenheterna i ett Fortran-program behandlas helt självständigt måste all information om argumenten föras över manuellt. I Fortran 77 skedde detta med otympliga argumentlistor. I Fortran 90 kan i stället ett gränssnitt kallat `INTERFACE` användas. Detta måste användas vid

  1. moduler som använder exempelvis egna datatyper,
  2. anrop med nyckelordsargument eller underförstådda argument
  3. egna generiska rutiner
  4. fält med antaget mönster
  5. fält deklarerade med pekare
  6. vid definition av ny betydelse hos en `OPERATOR`
  7. om avsikt `INTENT` skall få verkan vid användning av kompilatorn från NAG
  8. vid anrop av en fältfunktion med ett fält som resultat (behövs ej vid inbyggda fältfunktioner)

9. användning av subrutiner eller funktioner som argument (om man använder `IMPLICIT NONE`)

- huvudprogram (main program)
  Varje Fortranprogram måste bestå av exakt ett huvudprogram, samt eventuellt en eller flera subrutiner, funktioner, moduler och `BLOCK DATA` programenheter. Ett huvudprogram kan inledas med satsen `PROGRAM namn`, och avslutas med `END` eller `END PROGRAM` eller `END PROGRAM namn`.

- inbyggd (`INTRINSIC`)
  De funktioner och subrutiner som följer med kompilatorn kallas inbyggda. De har vissa speciella egenskaper, bland annat behöver eventuella generiska egenskaper inte deklareras. I undantagsfall levererar en del leverantörer ett par av de normalt inbyggda rutinerna som externa. Motsatsen till inbyggd är extern. Om inbyggda funktioner eller subrutiner användes som argument vid anrop av något annat underprogram skall de deklareras `INTRINSIC`. Observera skillnaden mellan inbyggda och interna underprogram.

- intern funktion (internal function)
  En funktion som är lokal till en viss programenhet, som finnes efter en `CONTAINS` sats. Alla variabler i den överordnade programenheten är direkt tillgängliga. En intern funktion kan ej användas utanför den direkt överordnade programenheten. Detta kan vara bra för att undvika namnkonflikter mellan funktioner och subrutiner från olika bibliotek. Interna funktioner är generellare än satsfunktioner. En intern funktion kan ej användas som argument! Det finns även interna subrutiner.

- kapslad (nested)
  Inuti en `DO`-slinga kan finnas ytterligare en `DO`-slinga (eller flera), inuti en `IF...THEN...ELSE...ENDIF`-konstruktion kan finnas ytterligare en eller flera, samt inuti en `CASE`-konstruktion kan finnas ytterligare en eller flera. Det är naturligtvis även möjligt att till exempel ha en `CASE`-konstruktion inne i en `DO`-slinga. Ibland kallas även ovan nämnda konstruktioner för slingor.

- lokal funktion
  Sammanfattning av interna funktioner och satsfunktioner.

- modul (module)
  En modul innehåller deklarationer (specifikationer) och definitioner som skall användas av de andra program- enheterna. Ersätter bland annat `BLOCK DATA`.

- programenhet (program unit)
  Sammanfattande namn på huvudprogram, subrutin, funktion, modul och `BLOCK DATA` programenhet.

- rekursiv (recursive)
  En funktion eller en subrutin som anropar sig själv kallas rekursiv, och är tillåten från Fortran 90.

- resultat-variabel (result variable)
  En funktion anropas med sitt funktionsnamn, som vid återhoppet innehåller funktionsvärdet. Om funktionen är rekursiv måste dock en speciell resultat-variabel användas internt inne i funktionen för att lagra funktionsvärdet (och skilja det från rekursivt anrop). Utifrån användes fortfarande det "vanliga" funktionsnamnet. Resultat-variabeln (som i funktionsdefinition finns inom parentes efter ordet `RESULT`) är praktisk vid en fältvärd funktion, även om denna ej är rekursiv.

- satsfunktion (statement function)
  En funktion som är lokal till en viss programenhet, finnes efter deklarationerna och före de exekverbara satserna. Mycket enkel form, fanns redan i FORTRAN I. En generellare form är den interna funktionen efter `CONTAINS`. En satsfunktion kan ej användas som argument!

- slinga (loop)
  Med slinga avses en följd av satser som upprepas, det vanligaste fallet är en DO-slinga, men det kan även vara en IF-sats (aritmetisk eller logisk eller mindre vanligt den fullständiga `IF...THEN...ELSE...ENDIF`-konstruktionen) som orsakar ett återhopp med en `GOTO`-sats till en tidigare del av programmet. Ibland kallas även en `CASE`-eller en `IF...THEN...ELSE ...ENDIF`-konstruktion för slinga. Explicita och implicita slingor diskuteras i avsnitt 6.5, sid 65, samt i ett enkelt exempel sist i laboration F.2, sid 219.

- subrutin (subroutine)
  En programenhet som ej returnerar något funktionsvärde genom sitt namn och som har ett antal parametrar. Det är tillåtet att subrutinen ändrar värdet på ett eller flera av argumenten. En subrutin kan dock ha en annan uppgift, till exempel att läsa in eller skriva ut data. En subrutin till skillnad från en funktion har ingen typ. Den anropas med `CALL subrutinnamnet`, medan en funktion anropas med bara `funktionsnamnet`. I båda fallen tillkommer naturligtvis eventuella argument.

Bilaga H

Svar och kommentarer till övningarna

Inledning

Det förutsättes att när inget annat angetts så gäller de implicita reglerna om heltal och flyttal, dvs `IMPLICIT NONE` har ej använts.

H.1 Svar

(2.1) Följande variabelnamn är tillåtna under Fortran 77 och Fortran 90:

```
A2  OOOOOO  DO      EIIR        Bettan
```

Följande variabelnamn är ej tillåtna under Fortran 77 (för långa och/eller innehåller understrykningstecken) men väl under Fortran 90:

```
GUSTAVUS        ADOLFUS             GUSTAV_ADOLF
HEJ_DU_GLADE    GOETEBORG
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Följande variabelnamn är inte tillåtna under Fortran 77 och inte heller under Fortran 90:

```
2C          inleds av siffra
2_CESAR     inleds av siffra
ÅKE         ej tillåtet med Å
$KE         ej tillåtet med $
C-B         ej tillåtet med minustecken
            (men korrekt som uttryck)
DOLLAR$     ej tillåtet med $
K**2        ej tillåtet med * (men korrekt som uttryck)
_STOCKHOLM_ ej tillåtet med inledande _ (understrykningstecken)
```

(2.2)

```
SUMMA = 0.0
DO I = 1, 100
        IF ( X(I) == 0.0 ) EXIT
        IF ( X(I) <  0.0 ) CYCLE
        SUMMA = SUMMA + SQRT(X(I))
END DO
```

Om man i stället använder en `FORALL`-konstruktion måste man notera att i `DO`-satsen tolkas "aktuellt" värde som det första med angiven egenskap (nämligen noll), varvid beräkningen skall avbrytas enligt specifikationen av uppgiften. Detta fungerar då inte vid den mer parallella `FORALL`-konstruktionen, som är olämplig i detta fall.

Anm. Uppgiften är inte helt entydig, eftersom man kan tänkas utföra `DO`-satsen i annan ordning!

(2.3)

```
SELECT CASE (N)
CASE(:-1)
        ! Fall 1
CASE(0)
        ! Fall 2
CASE(3,5,7,11,13)
        ! Fall 3
END SELECT
```

(2.4)

```
      PROGRAM SINUS
!     TABELLERING AV SINUS-FUNKTIONEN
      IMPLICIT NONE
      DOUBLE PRECISION :: A, B, H, X
      INTEGER :: N, I
      INTRINSIC SIN
  1   WRITE(*,*) ' Ge önskat intervall och antal delintervall!'
      READ(*,*) A, B, N
      IF ( A > B .OR. N <= 0 ) GO TO 1
      H = (B - A)/DBLE(N)
      WRITE(*,*)
      WRITE(*,*) ' Tabellering av sinus-funktionen sin(x) '
      WRITE(*,*)
      WRITE(*,*) '      x               sin(x)'
      WRITE(*,*)
      DO I = 0, N
         X = A + DBLE(I)*H
         WRITE(*,20) X, SIN(X)
      END DO
 20   FORMAT(1X,F12.6,F26.16)
      WRITE(*,*)
      STOP
      END
```

Som synes valde jag att göra denna beräkning i dubbel precision. Körning med enkla indata gav följande resultat.

```
  Ge önskat intervall och antal delintervall!
0 1 10
  Tabellering av sinus-funktionen sin(x)
       x               sin(x)
     0.000000        0.0000000000000000
     0.100000        0.0998334166468282
     0.200000        0.1986693307950612
     0.300000        0.2955202066613396
     0.400000        0.3894183423086505
     0.500000        0.4794255386042030
     0.600000        0.5646424733950354
     0.700000        0.6442176872376911
     0.800000        0.7173560908995228
     0.900000        0.7833269096274834
     1.000000        0.8414709848078965
```

(2.5)

```
      PROGRAM ETTAN
!     Kontroll av de trigonometriska funktionerna SIN och COS
      IMPLICIT NONE
      REAL :: A, B, H, X
      REAL :: Y, Z, ETTA, MAXFEL
      INTEGER :: N, I
      INTRINSIC SIN, COS
  1   WRITE(*,*) ' Ge önskat intervall och antal delintervall!'
      READ(*,*) A, B, N
      IF ( A > B .OR. N <= 0 ) GO TO 1
      H = (B - A)/REAL(N)
      WRITE(*,*)
      WRITE(*,*) ' Beräkning av felet vid beräkning av ', &
                 ' sinus och cosinus'
      WRITE(*,*)
      MAXFEL = 0.0
      DO I = 0, N
         X = A + REAL(I)*H
         Y = SIN(X)
         Z = COS(X)
         ETTA = Y**2 + Z**2
         MAXFEL = MAX(MAXFEL, ABS(ETTA-1.0))
      END DO
      WRITE(*,20) MAXFEL
 20   FORMAT(1X,E14.6)
      WRITE(*,*)
      END PROGRAM ETTAN
```

Resultatet av beräkningen på Sun blev föga överraskande $2 \cdot \mu = 0.1192 \cdot 10^{-6}$.

(2.6) Under fix form betyder den `LOGICAL L`, dvs variabeln `L` deklareras som logisk. Under fri form blir det syntaxfel.

(2.7) Nej, den är inte korrekt eftersom ett kommatecken fattas mellan `REAL` och `DIMENSION`. I den form som den skrivits tolkas satsen som en deklaration (gamla typen) av flyttalsmatrisen `DIMENSION` och en implicit deklaration (nya typen) av ett skalärt flyttal `AA`. Strikt formellt är det dock en korrekt deklaration. Variabelnamnet `DIMENSION` är tillåtet under Fortran 90, men det är naturligtvis för långt under Fortran 77.

(2.8) Ja, den är korrekt men den är mindre lämplig eftersom den dödar den inbyggda funktionen `REAL` för explicit konvertering av en variabel av annan typ till typen `REAL`. Det är dock inget som hindrar att man har en variabel av typ `REAL` med namnet `REAL`, eftersom Fortran inte har reserverade ord.

(2.9) Nej, den är inte korrekt, vid `COMMON` användes inte dubbelkolon vid deklaration, i princip eftersom `COMMON` är ett utgående kommando. Den rätta deklarationen är den gamla hederliga

```
        COMMON A
```

(3.1) Detta är en mycket avancerad variant till lösning!

```
SUBROUTINE LOES_LIN_EKV_SYSTEM(A, X, B, FEL)
IMPLICIT NONE
! Fältdeklarationer
REAL, DIMENSION (:, :), INTENT (IN)  :: A
REAL, DIMENSION (:),    INTENT (OUT) :: X
REAL, DIMENSION (:),    INTENT (IN)  :: B
LOGICAL, INTENT (OUT)                :: FEL

! Arbetsarean M är A utvidgad med B
REAL, DIMENSION (SIZE (B), SIZE (B) + 1) :: M
INTEGER, DIMENSION (1)                   :: MAX_LOC
REAL, DIMENSION (SIZE (B) + 1)           :: TEMP_ROW
INTEGER                                  :: N, K, I

! Initiera M
N = SIZE (B)
M (1:N, 1:N) = A
M (1:N, N+1) = B

! Triangulariseringsfas
FEL = .FALSE.
TRIANG_SLINGA: DO K = 1, N - 1
        ! Pivotering
        MAX_LOC = MAXLOC (ABS (M (K:N, K)))
        IF ( MAX_LOC(1) /= 1 ) THEN
                TEMP_ROW (K:N+1 ) = M (K, K:N+1)
                M (K, K:N+1)= M (K-1+MAX_LOC(1), K:N+1)
                M (K-1+MAX_LOC(1), K:N+1) = TEMP_ROW( K:N+1)
```

```
        END IF

        IF (M (K, K) == 0) THEN
                FEL  = .TRUE. ! Singulär matris A
                EXIT TRIANG_SLINGA
        ELSE
                TEMP_ROW (K+1:N) = M (K+1:N, K) / M (K, K)
                DO I = K+1, N
                        M (I, K+1:N+1) = M (I, K+1:N+1) - &
                          TEMP_ROW (I) * M (K, K+1:N+1)
                END DO
                M (K+1:N, K) = 0  ! Dessa värden användes ej
        END IF
END DO TRIANG_SLINGA
IF (M (N, N) == 0) FEL  = .TRUE.  ! Singulär matris A

! Återsubstitution
IF (FEL) THEN
        X = 0.0
ELSE
        DO K = N, 1, -1
                X (K) = ( M (K, N+1) - &
                  SUM (M (K, K+1:N) * X (K+1:N)) ) / M (K, K)
                END DO
        END IF
END SUBROUTINE LOES_LIN_EKV_SYSTEM
```

Ingående matris `A` och vektorer `B` och `X` har deklarerats med antaget mönster (assumed-shape array), dvs typ, rang och namn ges här, medan omfånget ges i anropande rutin, och då i ett explicit gränssnitt.

Notera att den inbyggda funktionen `MAXLOC` ger som resultat en heltalsvektor, med samma antal element som rangen (antalet dimensioner) hos argumentet. I vårt fall är argumentet en vektor, varför aktuell rang är ett och `MAXLOC` således ger en vektor med ett enda element. Detta är anledningen till att den lokala variabeln `MAX_LOC` deklarerats som en vektor med ett element. Om man deklarerar `MAX_LOC` som ett skalärt heltal fås kompileringsfel. Värdet blir naturligtvis index för det största elementet (det första av de största om flera är lika).

Notera även att numreringen börjar på 1 trots att jag tittar på en vektor med element som numreras från `K` till `N`. Jag har valt att inte utföra pivoteringen (dvs själva radbytet) om rutinen finner att raderna redan ligger rätt, nämligen när `MAX_LOC(1)` blir 1.

Beräkningen avbrytes så snart som singularitet konstateras, observera att denna kan inträffa så sent att den inte noteras inne i slingan.

Vid pivoteringen användes vektorn `TEMP_ROW` dels vid radbytet, dels utnyttjas den även för multiplikatorerna i Gausseliminationen.

Jag använder här tills vidare bara fältsektioner av vektortyp vid beräkningarna, men skall nu utnyttja funktionen `SPREAD` för att kunna använda fältsektioner av matristyp, varvid den innersta slingan försvinner (`DO I = K+1, N`).

Funktionen `SPREAD (SOURCE, DIM, NCOPIES)` returnerar ett fält av sam-

ma typ som argumentet `SOURCE`, men med rangen ökad med ett. Parametrarna `DIM` och `NCOPIES` är heltal. Om `NCOPIES` är negativ så användes i stället värdet noll. Om `SOURCE` är en skalär så blir `SPREAD` helt enkelt en vektor med `NCOPIES` element som alla har samma värde som `SOURCE`. Parametern `DIM` anger vilket index som skall utökas, det måste vara mellan 1 och `1+(rangen hos SOURCE)`, om `SOURCE` är en skalär måste således `DIM` vara ett. Parametern `NCOPIES` ger antalet element i den nya dimensionen, således ej antalet nya kopior. Om variabeln `A` svarar mot ett fält `(/ 2, 3, 4 /)` så blir

```
SPREAD (A, DIM=1, NCOPIES=3)        SPREAD (A, DIM=2, NCOPIES=3)

          2  3  4                          2  2  2
          2  3  4                          3  3  3
          2  3  4                          4  4  4
```

Jag går nu över till fältsektioner av matristyp, vilka kan kompileras effektivare på parallella maskiner, genom att byta ut den innersta explicita slingan, dvs

```
DO I = K+1, N
   M (I, K+1:N+1) = M (I, K+1:N+1) - &
     TEMP_ROW (I) * M (K, K+1:N+1)
END DO
```

mot

```
M (K+1:N, K+1:N+1) = M (K+1:N, K+1:N+1)  &
  - SPREAD( TEMP_ROW (K+1:N), 2, N-K+1) &
  * SPREAD( M (K, K+1:N+1), 1, N-K)
```

Anledningen till att vi måste "krångla tilldet med funktionen `SPREAD` är att i den explicita slingan är (vid fixt `I`) variabeln `TEMP_ROW(I)` en konstant som multipliceras med `N-K+1` olika värden `M`. Å andra sidan användes samma vektor ur `M` för alla värden på `I`, vilka är `N-K` stycken. Omformningen av matriserna skall ske till två matriser med samma mönster som delmatrisen `M (K+1:N, K+1:N+1)`, dvs `N-K` rader och `N-K+1` kolumner, eftersom alla de fyra räknesätten på fält är element för element.

Det kan tyvärr vara litet besvärligt att få parametrarna till `SPREAD` helt rätt. Till hjälp kan vara att utnyttja funktionerna `LBOUND` och `UBOUND` för att få lägre och övre dimensioneringsgränser, liksom att skriva ut det nya fältet med satsen

```
WRITE(*,*) SPREAD(SOURCE,DIM,NCOPIES)
```

Notera då att utmatningen sker kolonnvis (dvs första index varierar snabbast, som vanligt i Fortran). Man kan använda de lägre och övre dimensioneringsgränserna för en mer explicit utmatningssats som ger en utmatning bättre anpassad till hur fältet ser ut. Här måste man dock först ha gjort en tilldelning till ett vanligt deklarerat fält med rätt mönster för att kunna utnyttja index på vanligt sätt. Notera vidare att indexen i en konstruktion som `SPREAD` automatiskt kanar ner till 1 som lägre gräns, även när man som `SOURCE` har något som `A(4:7)`.

I den sista av `DO`-slingorna i programmet erhålles en tom summa vid index `N`, nämligen från `N+1` till `N`. Enligt Fortrans regler blir denna summa noll, vilket är korrekt ur matematisk synpunkt. Om man har indexkontroll påslagen är det dock risk för att denna larmar. Jag har därför bytt ut

```
DO K = N, 1, -1
        X (K) = ( M (K, N+1) - &
          SUM (M (K, K+1:N) * X (K+1:N)) ) / M (K, K)
        END DO
END IF
```

mot

```
X (N) = M (N, N+1) / M (N, N)
DO K = N-1, 1, -1
        X (K) = ( M (K, N+1) - &
          SUM (M (K, K+1:N) * X (K+1:N)) ) / M (K, K)
        END DO
END IF
```

i programmet `loes1.f90` i källkodsbiblioteket.

(4.1) Jag byter ut

```
IF ( F(MITT) > 0.0 ) THEN
   V = MITT
ELSE
   H = MITT
END IF
```

mot

```
IF ( F(MITT) == 0.0 ) THEN
    NOLL = MITT
    RETURN
ELSE IF ( F(MITT) > 0.0 ) THEN
    V = MITT
ELSE
    H = MITT
END IF
```

(4.2) En första ändring är att `IF ( H - V >= EPS )` nu måste skrivas som det fullständiga `IF ( ABS(H - V) >= EPS )` eftersom vi inte längre kan vara säkra på vilken punkt som är längst åt höger. En felutgång måste läggas in för det fall att `F(A)` och `F(B)` har samma tecken. I testen `IF ( F(MITT) > 0.0 )` byter vi ut `F(MITT)` mot `F(MITT)*F(V)`.

(4.3) Om man vill använda en mer avancerad algoritm kan man byta ut satsen `MITT = 0.5*(H-V)`, man kan därvid fortfarande kalla den nya punkten för `MITT`, trots att den nu inte längre ligger mitt mellan `V` och `H`.

(4.4)

```
RECURSIVE FUNCTION TRIBONACCI(N) RESULT (T_RESULTAT)
IMPLICIT NONE
INTEGER, INTENT(IN) :: N
INTEGER             :: T_RESULTAT
IF ( N <= 3 ) THEN
```

```
        T_RESULTAT = 1
ELSE
        T_RESULTAT = TRIBONACCI(N-1) + &
        TRIBONACCI(N-2) + TRIBONACCI(N-3)
END IF
END FUNCTION TRIBONACCI
```

Anropande program kan skrivas

```
IMPLICIT NONE
INTEGER :: N, M, TRIBONACCI
N = 1
DO
        IF ( N <= 0 ) EXIT
                WRITE (*,*) ' GE N '
                READ(*,*) N
                M = TRIBONACCI(N)
                WRITE(*,*) N, M
END DO
END
```

och ger resultatet `TRIBONACCI(15) = 2209`.

(4.5) Filen `kvad.f90` för adaptiv numerisk kvadratur (integration) ges nedan. Jag använder trapetsformeln, halverar steget och gör Richardsonextrapolation. Metoden blir därför ekvivalent med Simpsons formel. Som feluppskattning använder jag Linköpingsmodellen, där felet högst är beloppet av skillnaden mellan de båda icke extrapolerade värdena. Om felet är för stort tillämpas rutinen var för sig på de båda delintervallen, varvid tillåtet fel i vart och ett av delintervallen blir hälften av det tidigare tillåtna.

```
RECURSIVE FUNCTION ADAPTIV_KVAD (F, A, B, TOL, ABS_FEL) &
                 RESULT (RESULTAT)
IMPLICIT NONE

        INTERFACE
                FUNCTION F(X) RESULT (FUNKTIONSVAERDE)
                REAL, INTENT(IN)        :: X
                REAL                    :: FUNKTIONSVAERDE
                END FUNCTION F
        END INTERFACE

        REAL, INTENT(IN)  :: A, B, TOL
        REAL, INTENT(OUT) :: ABS_FEL
        REAL              :: RESULTAT

        REAL              :: STEG, MITT
        REAL              :: EN_TRAPETS_YTA, TVAA_TRAPETS_YTA
        REAL              :: VAENSTER_YTA, HOEGER_YTA
        REAL              :: DIFF, ABS_FEL_V, ABS_FEL_H
```

```
        STEG = B - A
        MITT = 0.5 * (A + B)

        EN_TRAPETS_YTA   = STEG * 0.5 * (F(A) + F(B))
        TVAA_TRAPETS_YTA = STEG * 0.25 * (F(A) + F(MITT)) + &
                           STEG * 0.25 * (F(MITT) + F(B))
        DIFF = TVAA_TRAPETS_YTA - EN_TRAPETS_YTA

        IF ( ABS(DIFF)  <  TOL ) THEN
                RESULTAT = TVAA_TRAPETS_YTA + DIFF/3.0
                ABS_FEL = ABS(DIFF)
        ELSE
                VAENSTER_YTA = ADAPTIV_KVAD (F, A, MITT, &
                        0.5*TOL, ABS_FEL_V)
                HOEGER_YTA   = ADAPTIV_KVAD (F, MITT, B, &
                        0.5*TOL, ABS_FEL_H)
                RESULTAT = VAENSTER_YTA + HOEGER_YTA
                ABS_FEL = ABS_FEL_V + ABS_FEL_H
        END IF
END FUNCTION ADAPTIV_KVAD
```

Filen `test_kva.f90` för test av ovanstående rutin för adaptiv numerisk kvadratur behöver `INTERFACE` både för funktionen `F` och för den egentliga kvadraturrutinen `ADAPTIV_KVAD`. Notera att för den senare måste funktionen `F` deklareras både med typ `REAL` och med `EXTERNAL`.

Jag har valt att komplettera detta program med mätning av exekveringstiden, utnyttjande de i Fortran 90 inbyggda subrutinerna för detta, se Bilaga D.21.1, sid 204.

```
PROGRAM TEST_ADAPTIV_KVAD
IMPLICIT NONE

        INTERFACE
                FUNCTION F(X) RESULT (FUNKTIONSVAERDE)
                REAL, INTENT(IN) :: X
                REAL             :: FUNKTIONSVAERDE
                END FUNCTION F
        END INTERFACE

        INTERFACE
                RECURSIVE FUNCTION ADAPTIV_KVAD &
                        (F, A, B, TOL, ABS_FEL) RESULT (RESULTAT)
                REAL, EXTERNAL    :: F
                REAL, INTENT(IN)  :: A, B, TOL
                REAL, INTENT(OUT) :: ABS_FEL
                REAL              :: RESULTAT
                END FUNCTION ADAPTIV_KVAD
        END INTERFACE

        REAL            :: A, B, TOL
        REAL            :: ABS_FEL
```

```
        REAL            :: RESULTAT, PI
        INTEGER         :: I

        INTEGER         :: COUNT1, COUNT2, COUNT_RATE
        REAL            :: TID

        PI = 4.0 * ATAN(1.0)
        A = -5.0 ; B = +5.0
        TOL = 0.1

        CALL SYSTEM_CLOCK(COUNT=COUNT1, COUNT_RATE=COUNT_RATE)

        DO I = 1, 5
                TOL = TOL/10.0
                RESULTAT =  ADAPTIV_KVAD (F, A, B, TOL, ABS_FEL)
                WRITE(*,*)
                WRITE(*,"(A, F15.10, A, F15.10)") &
                 "Integralen är approximativt  ", &
                RESULTAT, " med approximativ feluppskattning ", &
                  ABS_FEL
                WRITE(*,"(A, F15.10, A, F15.10)") &
                "Integralen är mer exakt      ", &
                  SQRT(PI), " med verkligt fel                ",&
                RESULTAT - SQRT(PI)
        END DO

        CALL SYSTEM_CLOCK(COUNT=COUNT2)
        TID = REAL(COUNT2 - COUNT1)/REAL(COUNT_RATE)
        WRITE(*,*) ' Beräkningen tar ', TID, ' sekunder'

END PROGRAM TEST_ADAPTIV_KVAD
```

Vi får naturligtvis inte glömma bort integranden, vilken jag dock låter ligga i samma fil som huvudprogrammet. Deklarationen är av den nya typen, speciellt vad gäller att resultatet returneras i en särskild variabel.

```
        FUNCTION F(X) RESULT (FUNKTIONSVAERDE)
        IMPLICIT NONE
        REAL, INTENT(IN)        :: X
        REAL                    :: FUNKTIONSVAERDE
        FUNKTIONSVAERDE = EXP(-X**2)
        END FUNCTION F
```

Nu är det dags att provköra på Sun-datorn. Jag har redigerat utmatningen något för att få den mer koncentrerad. Feluppskattningen är relativt realistisk i varje fall med denna integrand, den onormaliserade felfunktionen.

Huvudprogrammet och funktionen $f(x)$ finns i filen `test_kva.f90`, medan den rekursiva funktionen finns i `kvad.f90`. Dessa kan hämtas med WWW från internetversionen och kan därför enkelt användas för egna tester.

```
f90 test_kva.f90 kvad.f90
```

```
test_kva.f90:
kvad.f90:
a.out
Integralen är 1.7733453512 med feluppskattning 0.0049186843
                          med verkligt fel    0.0008914471
Integralen är 1.7724548578 med feluppskattning 0.0003375171
                          med verkligt fel    0.0000009537
Integralen är 1.7724541426 med feluppskattning 0.0000356939
                          med verkligt fel    0.0000002384
Integralen är 1.7724540234 med feluppskattning 0.0000046571
                          med verkligt fel    0.0000001192
Integralen är 1.7724539042 med feluppskattning 0.0000004876
                          med verkligt fel    0.0000000000
```

Jag har kört programmet på ett antal olika system och fått följande resultat, se tabell H.1. Vid upprepad körning varierar körtiden något.

Dator	Tid sekunder	Precision decimala siffror
PC Intel 486 SX 25	74,8	6
PC Intel 486 DX 50	2,75	6
PC Intel Pentium 200	0,26	6
Sun SPARC SLC	2,50	6
Sun SPARC station 10	0,58	6
Cray Y-MP	0,19	14
Cray C 90	0,13	14
DEC Alpha 3000/900	0,06	6
DEC Alpha 3000/900	0,06	15
DEC Alpha 3000/900	3,32	33

Tabell H.1: **Körtid av kvadraturprogrammet på olika system.**

Tiden på Cray är inte speciellt liten, eftersom detta program inte kunde utnyttja vektorisering, något som förutom den rena snabbheten är Cray's stora fördel. Vi noterar även att den fyrdubbla precisionen är ganska långsam på DEC Alpha, där dock enkel och dubbel precision tar samma tid!

Anm. Ovanstående program är skrivet på ett mycket direkt sätt för att illustrera rekursiv och adaptiv teknik. Det är därför inte optimerat, en enkel optimering föreslås nedan. Vi optimerar det adaptiva programmet för numerisk kvadratur genom att minska antalet funktionsanrop av funktionen $f(x)$.

Vi noterar att vid varje anrop av den rekursiva funktionen beräknas funktionsvärdet i de båda ändpunkterna samt i mittpunkten (det senare till och med två gånger)!

Det modifierade huvudprogrammet och funktionen $f(x)$ finns i internetversionen i filen `test_kv2.f90`, liksom den modifierade rekursiva funktionen i filen `kvad2.f90`.

Följande ändringar har gjorts:

1. Anropslistan till den rekursiva funktionen har utökats med funktionsvärdena i ändpunkterna. Denna ändring måste naturligtvis även ske i huvudprogrammet vid anropet av funktionen.

2. Funktionsvärdet i mittpunkten beräknas direkt vid inträdet i den rekursiva funktionen.

3. I huvudprogrammet måste funktionsvärdena i ändpunkterna beräknas före anropet av den rekursiva funktionen.

4. De nya variablerna `FA, FB` och `FMITT` måste naturligtvis deklareras.

Efter dessa ändringar exekverar programmet ungefär tre gånger snabbare! Funktionsberäkning sker nu bara en gång per anrop av den rekursiva funktionen (plus två initiala).

(4.6)

```
        SUBROUTINE SOLVE(F, A, B, TOL, EST, RESULTAT)
        REAL, EXTERNAL                 :: F
        REAL, OPTIONAL, INTENT (IN)    :: A
        REAL, OPTIONAL, INTENT (IN)    :: B
        REAL, OPTIONAL, INTENT (IN)    :: TOL
        REAL, INTENT(OUT), OPTIONAL    :: EST
        REAL, INTENT(OUT)    :: RESULTAT
        IF (PRESENT(A)) THEN
                TEMP_A = A
        ELSE
                TEMP_A = 0.0
        END IF
        IF (PRESENT(B)) THEN
                TEMP_B = B
        ELSE
                TEMP_B = 1.0
        END IF
        IF (PRESENT(TOL)) THEN
                TEMP_TOL = TOL
        ELSE
                TEMP_TOL = 0.001
        END IF
! Här skall den verkliga beräkningen finnas, men den
! är här ersatt med enklast tänkbara approximation,
! nämligen mittpunktsapproximation utan feluppskattning.
                RESULTAT = (TEMP_B - TEMP_A) &
                 * F(0.5*(TEMP_A+TEMP_B))
                IF (PRESENT(EST)) EST = TEMP_TOL
        RETURN
        END SUBROUTINE SOLVE
```

Den förenklade integralberäkningen ovan kan exempelvis ersättas med den adaptiva kvadraturen i föregående övning.

(4.7) Det exakta utseendet för gränssnittet beror naturligvis på integrationsrutinens utformning.

```
        INTERFACE
```

```
SUBROUTINE SOLVE (F, A, B, TOL, EST, RESULTAT)
        REAL, EXTERNAL               :: F
        REAL, INTENT(IN), OPTIONAL   :: A
        REAL, INTENT(IN), OPTIONAL   :: B
        REAL, INTENT(IN), OPTIONAL   :: TOL
        REAL, INTENT(OUT), OPTIONAL  :: EST
        REAL, INTENT(OUT)            :: RESULTAT
        END SUBROUTINE SOLVE
END INTERFACE SOLVE
```

(6.1) Satsen är ej tillåten, vilket dock visar sig först vid exekvering. Man kan i stället skriva

```
WRITE(*,*) ' HEJ '
```

eller

```
WRITE(*,'(A)') ' HEJ '
```

vilka båda skriver ut texten `HEJ` på standardenheten för utmatning. Om man vill ge den text man vill mata ut direkt på den plats där utmatningsformatet skall ges kan man antingen använda apostrofeditering

```
WRITE(*, "(' HEJ ')")
```

eller den föråldrade Hollerith-räknaren

```
WRITE(*, "(5H HEJ )")
```

(6.2) De skriver ut till beloppet stora eller små tal med en heltalssiffra, sex decimaler och exponent, medan tal mittemellan skrivs på ett naturligt sätt. Vi får således i detta fall

```
1.000000E-03
1.00000
1.000000E+06
```

Talen från 0,1 till 100 000 skrivs ut på det naturliga sättet, och med sex signifikanta siffror.

(8.1) Variablerna `A` och `B` tilldelas angivna värden, men hela resten av raden blir kommentar!

(8.2) Nej, på den andra raden är det blanka tecknet efter `&` ej tillåtet. Det bryter nämligen identifieraren `ATAN` i två identifieare. Om det blanka tecknet tas bort blir raderna korrekta. Fri form förutsättes, eftersom `&` ej är fortsättningstecken i fix form.

(9.1)

```
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(15,99)
```

(9.2)

```
REAL (KIND=DP) :: E, PI
```

(9.3)

```
REAL (KIND=DP), PARAMETER :: E = 2.718281828459045_DP, &
PI = 3.141592653589793_DP
```

(11.1) Jag antar att vektorn har en fix dimensionering, samt gör en kontrollutmatning av ett par värden.

```
REAL, TARGET, DIMENSION(1:100) :: VEKTOR
REAL, POINTER, DIMENSION(:)    :: UDDA, JAEMNA
UDDA   => VEKTOR(1:100:2)
JAEMNA => VEKTOR(2:100:2)
JAEMNA = 13   ;       UDDA   = 17
WRITE(*,*) VEKTOR(11), VEKTOR(64)
END
```

(11.2) Jag antar att den givna vektorn även här har en fix dimensionering.

```
REAL, TARGET, DIMENSION(1:10) :: VEKTOR
REAL, POINTER, DIMENSION(:)   :: PEKARE1
REAL, POINTER                 :: PEKARE2
PEKARE1 => VEKTOR
PEKARE2 => VEKTOR(7)
```

(13.1) Använd funktionerna `MIN` och `MAX` för att finna minsta respektive största värde av alla kombinationer `A%LAEGRE * B%LAEGRE, A%LAEGRE * B%OEVRE, A%OEVRE * B%LAEGRE` och `A%OEVRE * B%OEVRE` vid multiplikation, och motsvarande vid division.

(13.2) Testa om `B%LAEGRE <= 0 <= B%OEVRE`, i vilket fall felutskrift skall ske.

(13.3) Eftersom vi inte har direkt tillgång till maskinaritmetiken (dvs kommandon av typ avrunda nedåt respektive avrunda uppåt) så kan man få en hyfsad simulering genom att minska respektive öka med avrundningskonstanten. I princip dubbleras då effekten från avrundningen.

(14.1) Om detta misslyckas är det troligen något fel redan i det ursprungliga Fortran 77 programmet, eller Du har utnyttjat någon utvidgning från standard Fortran 77.

(14.2) Om detta misslyckas beror det troligen på att felaktiga kommandon tolkades som variabler under fix form, men nu när blanka är signifikanta upptäcks den felaktiga syntaxen.

Notera även att eventuell text i positionerna 73 till 80 under fix form normalt behandlas som kommentar (till exempel radnummer).

(14.3) Fortran 77 ger inget fel varken under kompilering eller exekvering. Kompilering under Fortran 90 fix form ger en varning från kompilatorn att variabeln `ZENDIF` användes utan att ha tilldelats något värde. Programmet tolkas nämligen så att det förutsättes att när inget annat angetts så gäller de implicita reglerna om heltal och flyttal, dvs `IMPLICIT NONE` har ej använts.

Litteraturförteckning

[1] Jeanne C. Adams och Walter S. Brainerd och Richard A. Hendrickson och Richard E. Maine och Jeanne T. Martin och Brian T. Smith. *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures.* Springer, 2008. ISBN 978-1-84628-378-9. (Citerad på sidorna 6, 145.)

[2] Katarina Blom. *Fortran 90 - en introduktion.* Studentlitteratur, 1994. ISBN 91-44-478814-X. (Citerad på sidan 6.)

[3] Walter S. Brainerd. *Guide to Fortran 2003 Programming.* Springer, 2009. ISBN 978-1-84882-542-0. (Citerad på sidan 6.)

[4] Walt Brainerd och Charlie Goldberg och Jeanne Adams. *Programmer's Guide to F.* The Fortran Company, 2001. ISBN 0-9640135-1-7. (Citerad på sidan 211.)

[5] Stephen J. Chapman. *Fortran 95/2003 for Scientists and Engineers.* McGraw-Hill, New York, N.Y., 3:e edition, 2007. ISBN 978-0-07-319157-7. (Citerad på sidan 145.)

[6] Ian D. Chivers och Jane Sleightholme. Compiler Support for the Fortran 2003, 2008, TS29113, and 2018 Standards, revision 24. *ACM SIGPLAN Fortran Forum*, 37(2):19–41, August 2018. (Citerad på sidorna 5, 145, 155, 156.)

[7] Viktor K. Decyk. A method for passing data between C and Opaque Fortran 90 pointers. *ACM SIGPLAN Fortran Forum*, 27(2):2–7, Augusti 2008. (Citerad på sidan 153.)

[8] Aleksandar Donev. Rationale for co-arrays in Fortran 2008. *ACM SIGPLAN Fortran Forum*, 26(3):9–19, December 2007. (Citerad på sidan 155.)

[9] Bo Einarsson. En jämförelse av FORTRAN IV och Fortran 77. ITM arbetsrapport 51, Institutet för Tillämpad Matematik, Stockholm, April 1978. (Citerad på sidan 16.)

[10] Bo Einarsson. Mixed Language Programming, Part 4, Mixing ANSI-C with Fortran 77 or Fortran 90; Portability or Transportability? In *Current Directions in Numerical Software and High Performance Computing*, International Workshop, Kyoto, Japan, Oktober 1995. `https://wg25.taa.univie.ac.at/ifip/kyoto/einarsson.html` . (Citerad på sidan 146.)

[11] Bo Einarsson, editor. *Accuracy and Reliability in Scientific Computing.* Software-Environments-Tools. SIAM, Philadelpahia, Pennsylvania, 2005. ISBN 0-89871-584-9. (Citerad på sidorna 128, 233.)

[12] Bo Einarsson och Richard J Hanson och Tim Hopkins. Standardized mixed language programming for Fortran and C. *ACM SIGPLAN Fortran Forum*, 28(3):8–22, December 2009. (Citerad på sidan 153.)

[13] Torgil Ekman och Göran Eriksson. *Programmering i Fortran 77.* Studentlitteratur, Lund, 3:e edition, 1984. ISBN 91-44-16663-X. (Citerad på sidan 6.)

[14] Lars Eldén och Linde Wittmeyer-Koch. *Numeriska beräkningar – analys och illustrationer med MATLAB.* Studentlitteratur, Lund, 4:e edition, 2001. ISBN 91-44-02007-4. (Citerad på sidorna 94, 214, 215.)

[15] Tommy Elfving och Jan Eriksson och Ulla Ouchterlony och Ingegerd Skoglund. *Numeriska beräkningar – en exempelsamling.* Studentlitteratur, Lund, 3:e edition, 2002. ISBN 91-44-02448-7. (Citerad på sidorna 215, 235.)

[16] Lloyd D. Fosdick och Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976. (Citerad på sidan 129.)

[17] Stefan Goedecker och Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, Philadelphia, 2001. ISBN 0-89871-484-2. (Citerad på sidan 135.)

[18] William Gropp och Ewing Lusk och Anthony Skjellum. *Using MPI, Portable Parallel Programmering with the Message-passing Interface.* The MIT Press, Cambridge, Massachusetts, 2:a edition, 2000. ISBN 0-262-57132-6. (Citerad på sidan 169.)

[19] Richard J Hanson och Tim Hopkins. *Numerical Computing with Modern Fortran.* SIAM Press, Philadelphia, 2013. ISBN 978-1-611973-11-2. (Citerad på sidan 155.)

[20] ISO. Standard for binary floating-point arithmetic, 1989. IEC 559:1989, även känd som IEC 60559 and IEEE 754-1985, Reviderad version IEEE 754-2008, ISBN 978-0-7361-5753-5. (Citerad på sidorna 87, 89, 89, 91, 125, 127, 145, 153, 190.)

[21] ISO. *ISO/IEC 1539-1:2010 Information technology – Programming languages – Fortran - Part 1: Base language*, 2010. Se även
ISO/IEC 1539-2:2000 (*Part 2: Varying length character*)
ISO/IEC 1539-3:1999 (*Part 3: Conditional compilation*)
TR 15580:2001 (*Floating-point exception handling*)
TR 15581:2001 (*Enhanced data type facilities*)
TR 19767:2005 (*Enhanced module facilities*)
. (Citerad på sidorna 3, 6, 60, 145, 155, 167, 172, 187, 188, 194, 194, 195, 195, 202, 232.)

[22] Donald E. Knuth och Luis Trabb Pardo. The early development of programming languages. In N. Metropolis och J. Howlett och Gian-Carlo Rota, editor, *A History of Computing in the Twentieth Century*, pages 197 – 273. Academic Press, 1980. ISBN 0-12-491650-3. (Citerad på sidan 207.)

[23] Charles H. Koelbel och David B. Loveman och Robert S. Schreiber och Guy L. Steele och Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Massachusetts, 1994. ISBN 0-262-61094-9. (Citerad på sidorna 14, 17, 29, 122.)

[24] John M. Levesque och Joel W. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, San Diego, CA, 1989. ISBN 0-12-444760-0. (Citerad på sidorna 135, 140.)

[25] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly, 2004. ISBN 0-596-00610-1. (Citerad på sidan 226.)

[26] Michael Metcalf. *Fortran Optimization*. Academic Press, London och New York, 1982. ISBN 0-12-492480-8. (Citerad på sidan 135.)

[27] Michael Metcalf och John Reid och Malcolm Cohen. *Modern Fortran explained*. Oxford University Press, 2011. ISBN 978-0-19-960141-7. (Citerad på sidorna 6, 145, 147, 153, 187.)

[28] Robert W Numrich och John K Reid. *Coarrays, Parallel Programming in Fortran*. Chapman & Hall/CRC, 2016. ISBN 978-1439840047. (Citerad på sidan 155.)

[29] John Reid. The new features of Fortran 2003. *ACM SIGPLAN Fortran Forum*, 26(1):10–33, April 2007. (Citerad på sidan 145.)

[30] John Reid. Coarrays in the next Fortran standard. ISO/IEC, Mars 2008. `ftp://ftp.nag.co.uk/sc22wg5/N1710-N1750/N1724.pdf`. (Citerad på sidan 155.)

[31] John Reid. The new features of Fortran 2018. *ACM SIGPLAN Fortran Forum*, 37(1):5–43, April 2018. (Citerad på sidan 156.)

[32] Lennart Råde och Bertil Westergren. *Beta, Mathematics Handbook for Science and Engineering*. Studentlitteratur, Lund, 4:e edition, 1998. ISBN 91-44-00839-2. (Citerad på sidan 229.)

[33] Simon Singh. *Big Bang*. HarperCollins Publishers, London, 2005. ISBN 0-00-715252-3 Engelsk pocket, ISBN 91-7343020-X Svensk inbunden. (Citerad på sidan 233.)

[34] Christoph Überhuber och Peter Meditz. *Software-Entwicklung in Fortran 90*. Springer, Wien, 1993. ISBN 3-211-82450-2. (Citerad på sidan 95.)

Sakregister