FORTRAN 90 FIZIKUSOKNAK

Tóth Gábor

2002. szeptember 27.

Tartalomjegyzék

1. Bevezetés

1.1. Fortran

FORTRAN=FORmula TRANslator

IBM kezdte 1950-es évek végén. Standardok: Fortran 66, Fortran 77. Új kibővített nyelv: Fortran 90.

Fő alkalmazások: numerikus, mérnöki, tudományos

Összevetés C, C++, Java, Perl nyelvekkel.

1.2. A Fortran nyelv alapjai

A kis és nagy betűk nem különböznek!

Egy rövid program:

```
program test
   implicit none

end program test
```

Az `implicit none` utasítás arra utal, hogy minden változót deklarálni kell. Ebben a programban persze még egy változó sincs.

A változó nevek betűvel kell hogy kezdődjenek, amit betűkből számokból és az aláhúzás jelből álló legfeljebb 30 elemű karakter sor követhet. Formálisan:

```
ValtozoNev ::= [A-Z][A-Z,_,0-9]*
```

Összhossz maximum 31 karakter.

Utasítások elválasztása: sorvége vagy ";"

Egy sorban csak az első 132 karaktert olvassa el a fordító program. A képernyő ablakban elférő illetve a printereken kinyomtatható sorhosszak miatt ezt sem érdemes kihasználni, általában érdemes maximum 80 karakter hosszú sorokat írni. Fortran 77-nél a maximális sorhossz 72 karakter volt, ami a lyukkártyák méretéből öröklődött.

Nagyon hosszú sorokat folytatósorok segítségével lehet leírni. Ezeket a folytatandó sor végén illetve a folytató sor elején (ez utóbbi nem kötelező, de hasznos lehet) álló `&` jelöli:

```
a= 1+2+3+ &
   4+5+6
b="kicsi a bors, &
  &de eros"
```

A második példában a folytatósor elején lévő `&` jelre azért van szükség, hogy a szóközök száma egyértelmű legyen. Legfeljebb 39 folytatósort lehet egymás után írni.

Megjegyzéseket a `!` karakter mögé lehet írni a sor végéig:

```
! Adjunk ossze 6 szamot
a= 1+2+3+ &   ! az elso harom tag
              ! meg a folytatosorok kozott is lehet megjegyzes
   4+5+6      ! a masodik harom tag
```

1.3. Változók deklarálása és értékadás

Változó típusok és deklarációk:

```
integer :: i                    ! egesz
real    :: a                    ! valos
complex :: c                    ! komplex
logical :: IsDone               ! logikai
character :: letter             ! karakter
character (LEN=20) :: string    ! karakter lanc
type person                     ! tipus deklaracio eleje
  character (LEN=50) :: name       ! 1. mezo: nev
  integer            :: age        ! 2. mezo: kor
  real               :: height     ! 3. mezo: magassag
end type person                 ! tipus deklaracio vege
type(person) :: Mike            ! osszetett tipusu valtozo
```

!!! kimaradt: kind, double precision !!!

Konstans kifejezések használata

```
i      = -123                          ! egesz szam
```

```
a      = -123.0                          ! valos szam tizedes ponttal
a      = -1.23E2                         ! valos szam exponenssel
c      = (-123.0, 0.0)                   ! komplex valos es kepzetes resszel
letter = 'a'; letter = "b"               ! karakter szimpla/dupla idezojellel
string = "Mike Tyson"                    ! karakter lanc
IsDone = .true.                          ! hamis logikai ertek
IsDone = .false.                         ! igaz  logikai ertek
Mike   = person('Mike Tyson', 33, 178.4) ! osszetett tipusu konstans
```

1.4. Műveletek

Műveleti jelek: $+, -, *, /, **$

A $**$ a hatványozás jele. Az egész hatványokat a fordító szorzással helyettesíti, tehát felesleges az `a**2`-et `a*a` alakban írni. A műveleti sorrendet (precedencia) a következő órán tekintjük át.

Az egészek osztásával vigyázni kell! A két alábbi utasítás egyenértékű:

```
a = 3/5
a = 0.0
```

Kifejezések, melyek eredménye 0.6:

```
a = 3.0/5
a = 3/5.0
a = real(3)/real(5)
```

A `real` transzformációs függvény az egész számot valósra konvertálja.

1.5. Írás és olvasás

A kommunikáció a külvilággal az írás és olvasás parancsokon keresztül történik. Ezek legegyszerűbb formája:

```
read(*,*) a, b              ! beolvasas
write(*,*)'Ratio=',a/b      ! kiiras
```

A `read` és `write` utasítások első paramétere azt adja meg, hogy honnan olvas illetve hova ír, a második pedig a formátumot. A `*` az alapértelmezést jelenti,

azaz a képernyőre (STDOUT) ír, a billentyűzetről (STDIN) olvas, és a részben a fordító program által meghatározott formátumban ír és olvas. Az utasítások után tetszőleges sok változó (írás esetén kifejezés is lehet) sorolható fel vesszőkkel elválasztva.

!!! KIMARADT formátum, stringbe írás, fileba írás, unformatted, err=, end=, stb. !!!

1.6. UNIX alapok

Új jelszó megadása:

```
passwd
```

Állomány szerkesztése:

```
emacs test.f90 &
```

Az emacs a file nevéről felismeri annak típusát, és megfelelően alkalmazkodik hozzá. Néhány hasznos billentyű kombináció:

`Ctrl-A`	ugrás sor elejére
`Ctrl-E`	ugrás sor végére
`Ctrl-K`	a kurzortól a sor végéig törlés, lehet többször is
`Ctrl-Y`	utoljára törölt sorok visszaírása a kurzor pozíciójánál
`Ctrl-X U`	az utolsó utasítás visszavétele, lehet többször is
`TAB`	a sor megfelelő pozícióba tolása, "end" utasitások befejezése
`ESC Ctrl-Q`	a kurzorral kiválasztott program rész sorainak rendezése

A UNIX operációs rendszer egyszerre több programot hajt végre. Ezeket processzeknek hívjuk. A processzek indítása, megállítása, figyelése:

`emacs akarmi &`	program indítása háttérben
`emacs valami`	program indítása előtérben
`Ctrl-Z`	futó program felfüggesztése
`jobs`	az ablakban futó ill. felfüggesztett programok listája
`bg`	az utoljára felfüggesztett program háttérben futtatása
`kill fg`	a háttérben futtatott program előtérbe helyezése
`Ctrl-C`	előtérben futó program megállítása
`ps`	az összes futó process listája, elől az ID
`kill 14434`	az 14434 ID-jű process leállítása (ha figyel rá)
`kill -9 14434`	az 14434 ID-jű process leállítása (ha nem figyel)
`top`	a futo processek listaja CPU ido szerint rendezve

Fortran program fordítása és futtatása:

```
pgf90 test.f90
a.out
pgf90 -o test.exe test.f90
test.exe
```

A `-o` után írható a végrehajtjató program file neve. Ha ez nem szerepel, akkor az eredmény az `a.out` file lesz.

Fájlok másolása, mozgatása, törlése valamint alkönyvtárak létrehozása, használata, törlése:

`touch file1`	file1 megérintése/létrehozása
`cp file1 file2`	file1 másolása file2-be (copy)
`mv file2 file3`	file2 átnevezése file3-ra (move)
`rm file3`	file1 törlése (remove)
`mkdir dir1 dir2`	alkönyvtár(ak) létrehozása (make directory)
`mv file1 dir2 dir1`	file1 és dir2 berakása a dir1 alkönyvtárba
`cd dir1`	belépés a dir1 alkönyvtárba (change directory)
`mv dir2 dir3`	dir2 alkönyvtár átnevezése dir3-ra
`rmdir dir3`	dir3 alkönyvtár eltávolítása
`cd ..`	kilépés a dir1 alkönyvtárból
`rm -f dir1/*`	dir1 alkönyvtárban lévő fájlok azonnali törlése (forced)
`rm -rf dir1`	dir1 alkönyvtár rekurzív azonnali törlése

!!! TAB hasznalata tcsh alatt !!!

1.7. Programozási feladat: hello world

Írjon egy programot, ami kiírja a képernyőre, hogy "Hello world!". Megoldás:

```
program hello
  ! This is a little test program
  implicit none
  character (LEN=30) :: Text

  Text="Hello &
       &world!"
  write(*,*)Text
end program hello
```

2. Változók és kifejezések

2.1. Programozási feladat: másodfokú egyenlet

Másodfokú egyenlet megoldása. Beolvassa az A, B, C együtthatókat és kiírja az X1 és X2 gyököket, melyek komplexek is lehetnek. Az aritmetikai műveletek elvégzési sorrendje (felülről lefelé):

```
**
* , /
+ , -
```

Az azonos szinten álló műveletek a kifejezésben balról jobbra hajtódnak végre. A végrehajtási sorrendet zárójelekkel lehet megváltoztatni. Pl:

```
D=B**2 - 4*A*C; Q = A**(2*B); X1 = (-B + SQRT_D)/(2*A);
```

A diszkrimináns kiszámításánál szükség van a `cmplx` transzformációs függvényre! A négyzetgyökvonáshoz érdemes az `sqrt` függvényt használni, bár a 0.5-ik hatványra emelés is jó megoldás.

Megoldás:

```
program solve2
  ! Solve second order equation a*x**2 + b*x + c = 0
  implicit none

  real :: a, b, c
  complex :: Discriminant, x1, x2
  !-----------------------------------------------
  write(*,*)'Give a, b and c:'
  read(*,*)a,b,c
  Discriminant = sqrt(cmplx(b**2 - 4*a*c))
  x1= (-b + Discriminant)/(2*a)
  x2= (-b - Discriminant)/(2*a)
  write(*,*)'Root1=',x1,' Root2=',x2
end program solve2
```

Megjegyzés: a program $a = 0$ esetén nem működik! Feltételes utasításokra van szükség!

2.2. Operátorok

Numerikus kifejezések között:
`>, <, ==, >=, <=, /=`

Az eredmény egy logikai kifejezés, melynek értéke `.true.` vagy .false.

Ugyanezen relációs operátorok régi Fortran 77-es jelölése:
`.gt., .lt., .eq., .ge., .le., .ne.`

Logikai kifejezések közti logikai operátorok végrehajtási sorrendje (felülről lefelé):

```
    .not.
    .and.
    .or.
.eqv.  .neqv.
```

Logikai kifejezésekben is zárójelezéssel lehet a műveleti sorrendet megváltoztatni. Példák:

```
A/=0.0
A>0.0 .and. B<0.0 .or. A<0.0 .and. B>0.0
(A==B .or. A==C) .and. .not. B==C
```

A `.not.B==C` helyett természetesen `B/=C` is írható.

2.3. Műveletek karakterláncokkal

Karakterláncok közötti műveletből csak egy van, ez az összefűzés (concatenation) operátor: `//`. Karakterláncok (string) részeit (substrings) a következő módon jelöljük:

```
str(I:J)
str(:J)   ! str(1:J)
str(I:)   ! str(I:len(str))
str(I:I)
```

Az első karakter indexe 1. Megjegyzés: Az I-ik karakter nem jelölhető `str(I)`-vel, csak a `str(I:I)` jelölés helyes. A `len` függvény a karakterláncnak a deklarációban szereplő hosszát adja meg. Ha a záró szóközök nélküli hosszra van szükségünk, akkor a `len_trim` függvényt kell használni.

Példák:

```
str="kicsi"//" kocsi"
write(*,*)str(:5)   ! kicsi
write(*,*)str(6:)   ! kocsi
str(1:5)="nagy"
write(*,*)str       ! nagy  kocsi
str(1:5)=str(3:7)
write(*,*)str(1:7)  ! gy  k kocsi
```

Az utolsó értékadás mutatja, hogy megengedett az átfedés, és a szabály az, hogy az eredeti részstring másolódik be.

2.4. Tömbök

Az indexek alapértelmezésben 1-gyel kezdődnek (ellentétben a C-vel, ahol 0-val). A memóriában az egymás melletti tömb elemek az első indexben különböznek (ellentétben más programozási nyelvekkel). Legfeljebb 7 indexe lehet egy tömbnek. Szükség esetén ez összetett típusokkal növelhető.

Példák tömb deklarációkra:

```
real :: a(10,20), b(-1:12,-10:10)
real, dimension(-1:12, -1:12) :: c, d, e
type(person) :: people(100)
character (len=80), dimension(25) :: page
```

A `page` változó például tartalmazhatja a képernyőn megjelenített szöveget. A 10-dik sor első 40 betűjét a `page(10)(1:40)` tartalmazza.

Tömb elemeket skalár indexekkel, míg résztömböket a `minimum:maximum:lépésköz` formában lehet megadni. A lépésköz negatív is lehet. Példák:

```
a(1,3)       ! egy elem
a(1:5,3)     ! 5 elemu vektor
a(:,:)       ! 10x20 -as tomb
b(1:5:2,0)   ! 3 elemu vektor: b(1,0), b(3,0), b(5,0)
a(1:10:-1,1) ! 10 elemu vektor: a(10,1), a(9,1), ..., a(1,1)
a(2:1,:)     ! 0 elembol allo tomb
```

Tömb konstansokat a `(/ ... /)` kifejezéssel lehet megadni:

```
a(1:5,1)=(/ 0.0, 1.0, 2.0, 3.0, 4.0 /)
a(1:5,1)=(/ (I-1.0, I=1,5) /)            ! ugyanaz mint az elozo
a(1:5,1)=(/ (I*0.1, I=1,9,2) /)          ! 0.1, 0.3, 0.5, 0.7, 0.9
```

Fontos, hogy ilyen módon csak konstans értékek adhatók meg, azaz változókat nem tartalmazhatnak a kifejezések. Többdimenziós tömkonstansokat a

```
reshape(elemek,alak)
```

konverziós függvény segítségével lehet megadni, pl.

```
a(1:2,1:2) = reshape( (/1.0, 2.0, 3.0, 4.0/), (/2, 2/))
```

!!! KIMARADT !!! Indirekt indexek

2.5. Összetett típusú változó: `derived type`

Már adtunk példát összetett típusú változóra és konstans értékadásra. Most bővítsük ki ismereteinket tömbökkel:

```
type person                              ! szemely tipus deklaracio eleje
  character (LEN=50) :: name               ! karakterlanc: nev
  integer            :: age                ! egesz szam:   kor
  real, dimension(2) :: lArm               ! valos tomb:   karhossz
end type person                          ! szemely tipus deklaracio vege
type(person) :: Mike                     ! szemely tipusu valtozo: Mike
Mike = person('Mike Tyson', 33, (/80.4, 80.5/))   ! ertekadas
```

Mint látható, az összetett típusú változókra teljes egészükben is lehet hivatkozni, de egyes részeik is elérhetőek a `%` karakter segítségével. Például

```
 write(*,*) Mike%name(1:4), Mike%age, Mike%lArm(2)  ! Mike  33 80.5
```

Összetett típusú változónak a mezője is lehet összetett típusú, de rekurzív típus definíció nem megengedett. Például egy ember 2 kocsijának adatait a következkezőképpen tárolhatjuk

```
type CarType                           ! kocsi tipus deklaracio eleje
  character (len=50) :: type, color      ! karakterlanc: tipus, szin
  integer            :: age              ! egesz:        kor
```

```
  end type CarType                    ! kocsi tipus deklaracio vege
  type PersonType                     ! szemely tipus deklaracio eleje
    character (len=50) :: name          ! karakterlanc:      nev
    type(CarType)      :: car(2)        ! kocsi tipusu tomb: kocsik
  end type PersonType                 ! szemely tipus deklaracio vege
  type(PersonType) :: Bob             ! szemely tipusu valtozo: Bob
  Bob%car(1)%color = 'red'            ! ertekadas: Bob elso kocsija piros
  Bob%car(2)%type  = 'Porsche'        ! ertekadas: Bob masodik kocsija Porsche
```

Eddig csak statikus adatstruktúrákat definiáltunk. Dinamikus adatstruktúrákról a későbbiekben lesz szó. !!!

3. Kontrol utasítások

3.1. Ugrás: `GOTO`

A `GOTO` egy legfeljebb 5 jegyű pozitív egész számmal, úgynevezett numerikus cimkével megjelölt sorra ugrik, és onnan folytatja a végrehajtást. Például:

```
    a=1
    goto 100
    a=2
100 write(*,*)a  ! 1 lesz az eredmeny
```

A `GOTO` utasítást nagy előszeretettel használták a Fortran 77-ben, ahol viszonylag keves kontrol utasítás állt rendelkezésre. A sok GOTO utasítással megírt program áttekinthetetlen és a hibák keresése rendkívül nehéz. Fortran 90-ben szinte soha nincs szükség a GOTO utasításra, ritka kivétel lehet a hibák kezelése.

3.2. Megállás: `stop`

A `STOP` utasítás megállítja a végrehajtást. Több formája létezik:

```
  stop                        ! megall
  stop 100                    ! megall es a 100 as szamot irja ki
  stop "Ezert alltam meg!"    ! megall es szoveget ir ki
```

A szam csak pozitiv egész es legfeljebb 5 jegyű lehet, a karakter lánc pedig egy konstans, azaz változót nem tartalmazhat. Ha változót kell kírnunk a megállás előtt, a `write` utasítást használhatjuk.

3.3. Feltételes utasítás: `if..then..else`

A feltételes utasítás legegyszerűbb formája az `IF` utasításból, egy zárójelben álló logikai kifejezésből és az azt követő végrehajtandó utasításból áll. Például:

```
  if(a>0.0) write(*,*)'a is positive'
```

Ha több mint egy utasítást akarunk végrehajtani, akkor az `if(..)then...end if` formát kell alkalmazni:

```
if(a>0.0)then
  write(*,*)'a was positive'
  a=-a
end if
```

Ha a logikai kifejezés hamis értékéhez is rendelni akarunk végrehajtandó utasítást, akkor az `if(..)then...else...end if` kontrol utasítást kell használni:

```
if(a>0.0)then
  write(*,*)'a is positive'
else
  write(*,*)'a is not positive'
end if
```

Végül egy egész feltétel lánc is megadható a `if(..)then...else if(...)then...else if(...)then...else...end if` alakkal:

```
if(a>0.0)then
  write(*,*)'a is positive'
else if (a<0.0) then
  write(*,*)'a is negative'
else
  write(*,*)'a is zero'
end if
```

Természetesen az utolsó `else` rész nem kötelező. A feltételes utasítások egymásba ágyazhatóak:

```
  if(a>0.0)then
     if(b>0.0)then
        write(*,*)'a and b are positive'
```

```
    else
       write(*,*)'a is positive but b is not'
    end if
  else
    write(*,*)'a is not positive'
  end if
```

A sok feltételes utasítás egymásba ágyazásánál fontos a sorok megfelelő tördelése. Tovább segítheti az összetartozó `if`, `else if`, `else` és `end if` részek megtalálását, ha egy szöveges cimkét használunk, például:

```
SIGNA:if(a>0.0)then
  SIGNB:if(b>0.0)then
          write(*,*)'a and b are positive'
        else SIGNB
          write(*,*)'a is positive but b is not'
        end if SIGNB
      else SIGNA
        write(*,*)'a is not positive'
      end if SIGNA
```

3.4. Kapcsoló: `select case`

A `SELECT CASE` kontrol utasítás egész illetve karakter(lánc) típusú konstans értékek közötti választást tesz lehetővé. Az utasítás általános formája

```
  select case(i+1)
  case(:-1)
     write(*,*)'i+1 is negative'
  case(1)
     write(*,*)'i+1 is 1'
  case(2:3)
     write(*,*)'i+1 is 2 or 3'
  case(4,6,8,10:)
     write(*,*)'i+1 is 4, 6, 8, 10 or larger'
  case default
     write(*,*)'i+1 is something else'
  end select
```

Természetesen a `case default` rész nem kötelező. Ha hiányzik, és az egyik ágban szereplő `case(..)` sem tartalmazza a szelekciós változó értékét, akkor a vérgehajtás egyszerűen az `end select` után folytatódik.

Fontos, hogy az egyes ágakban szereplő értékek, értékhalmazok nem fedhetnek át egymással. Különösen hasznos a `select case` utasítás karakterlánc típusú változóknál:

```
select case(Name)
case('kovacs','KOVACS')
  Name='Kovacs'
  write(*,*)'Name is corrected to Kovacs'
case('szabo','SZABO')
  Name='Szabo'
  write(*,*)'Name is corrected to Szabo'
case('Kovacs','Szabo')
  ! nincs semmi teendo
case default
  write(*,*)'Unknown Name=',Name
end select
```

3.5. Ciklus: do..cycle..exit

A `DO` ciklus utasítás az egyik leggyakrabban használt kontrol utasítás. A ciklus utasítás lehetővé teszi, hogy ugyanazt a programrészt többször végrehajtsuk. Ha ismert számú iterációt hajtunk végre, akkor a
`DO valtozo=kezdoertek,vegertek,lepeskoz`
formát érdemes használni. A változó csak egész típusú lehet, a lépésközt nem kötelező megadni, alapértelmezésben az értéke 1. Példa az első 100 egész szám kiírására:

```
integer :: i
do i=1,100
  write(*,*)i
end do
```

Példa minden második elem összeadására az n-ikig:

```
real :: a(10), sum
integer :: i, n
a = (/2, 3.1, 4.0, -6.1, 2e-3, 1.0, 5.0 ,9.4, 0.0, -1.e+3/)
read(*,*) n
sum=0.0
do i=1,n,2
   sum = sum + a(i)
end do
```

Példa negatív lépésköz alkalmazására:

```
do i=10,2,-1
   a(i)=a(i)-a(i-1)
end do
```

Vegyük észre, hogy az eredmény egészen más lenne, ha az indexeken növekvő sorrendben mennénk végig. Ha a lépésközt nem írtuk volna ki, akkor a ciklus egyszer sem hajtódna végre, hiszen a 10-es kezdő érték nagyobb mint az 2-es végérték!

A ciklusból bármikor ki lehet lépni az EXIT utasítással, illetve a következő iterációra lehet ugrani a CYCLE utasítással. Például vonnjunk gyököt az a tömb pozitív elemeiből, egészen a -666. értékű elemig:

```
do i=1,10
   if(a(i)==-666.0) EXIT
   if(a(i)<0.0) CYCLE
   a(i)=sqrt(a(i))
end do
```

Ha az iterációk száma nem ismert előre, akkor ciklus változó nélküli DO utasítást lehet használni. A ciklusból az EXIT utasítással kell kilépni. Például olvassunk be egy pozitív számot:

```
do
   write(*,*)'Please give me a positive integer number:
   read(*,*) n
   if( n > 0) EXIT
   write(*,*)'This is not a positive integer !'
end do
```

A ciklus addig fut amíg pozitív számot nem adunk meg.

A ciklusok egymásba ágyazhatóak. Példa mátrixok szorzására:

```
do i=1,n
   do j=1,n
      a(i,j)=0.0
      do k=1,n
         a(i,j)=a(i,j)+b(i,k)*c(k,j)
      end do
   end do
end do
```

Az `i` ciklus változó értékét a `do i=j,k,l` ciklus utasítás után a Fortran 90 a következőképpen definiálja:

- ha $j > k$ és $l > 0$ vagy $j < k$ és $l < 0$ (azaz a ciklus egyszer sem hajtódik végre) akkor $i = j$, azaz a kezdőérték
- ha a ciklusból EXIT utasítással léptünk ki, akkor megmarad i pillanatnyi értéke az EXIT végrehajtásakor.
- ha a ciklus végigfut, akkor $i = k + l$ lesz, azaz a végérték után következő érték.

3.6. Programozási feladat: prím számok

Keressük meg és írjuk ki az első 1000 szám közül a prímszámokat az Eratosztenészi szita módszerrel. A szita algoritmusa az, hogy először minden számról feltételezzük hogy prímszám, majd sorban kihúzzuk a 2, 3, stb. 1-nél nagyobb egész számú többszöröseit, pl. 2-re a 4,6,8,...,1000 számokat.

Optimalizációk az algoritmusban:

- elegendő elmenni a legnagyobb szám gyökéig.
- a már kihúzott (nem prím) számok többszöröseivel nem kell foglalkozni

A programban használjuk a `parameter` deklarációt, hogy ha 1000 helyett 100-ig keresünk prímeket, akkor csak egy helyen kelljen változtatni. Továbbá szükség lesz a `floor` függvényre, ami a valós paramétérhez a legközelebbi, a valós számnál nem nagyobb egész számot adja vissza.
Például a `write(*,*)floor(3.2), floor(1.0), floor(-0.1)` eredménye `3 1 -1` lesz.

Megoldás:

```
program prime
  implicit none

  integer, parameter :: n=1000       ! konstans parameter
  logical :: IsPrime(n)              ! .true. a primekre
  integer :: i                       ! index valtozo
  !-------------------------------------------------------------
  IsPrime=.true.                     ! mindrol feltesszuk hogy prim
  do i=2,floor(sqrt(real(n)))        ! osztokkal gyok n-ig megyunk
     if(IsPrime(i)) &                ! ami meg nincs kihuzva, annak
```

```
        IsPrime(2*i : n : i)=.false. ! kihuzzuk a tobbszoroseit
  end do
  do i=2,n                           ! kiirasnal n-ig megyunk
    if(IsPrime(i)) write(*,*)i       ! kiirjuk a primeket
  end do
end program prime
```

4. Dinamikus változók

4.1. Foglalható tömbök: `allocatable`

Dinamikus méretű tömböket az `allocatable` tulajdonsággal kell deklarálni, és a dimenziók számát kettőspontokkal kell jelölni. Például

```
real, allocatable :: a(:), b(:,:)
integer, allocatable, dimension(:,:) :: c, d
```

Az így deklarált tömbök számára az `allocate` utasítással lehet memóriát foglalni. A tömbök méretét ugyanúgy kell megadni, mint a szokásos statikus deklarációnál, azonban most tetszőleges egész változók és kifejezések is használhatóak. Például:

```
integer :: n
read(*,*)n
allocate(a(n), b(-n:n,-n:n), c(0:n+1,0:5), d(0:5,0:n+1))
```

Ha a tömb számára nincs elég hely a memóriában, a program futása hibajelzéssel leáll. Ez elkerülhető, ha használjuk az `allocate` második `STAT` paraméterét, ami egy egész változóba ír 0-t sikerese foglalás esetén, és 0-tól különböző számot egyébként. Például:

```
integer :: s
do
  write(*,*)'Size of array='
  read(*,*) n
  allocate(b(n,n), STAT=s)
  if(s==0) EXIT
  write(*,*)'Could not allocate array b'
end do
```

Amíg a tömb nincs lefoglalva, addig nem lehet a tömb elemekhez hozzáférni. Azt, hogy a tömb már le van-e foglalva, az `allocated` függvénnyel lehet eldönteni. Például:

```
if(.not.allocated(c)) allocate(c(0:n+1,0:n+1))
```

Ha már nincs szükség a foglalható tömbre, érdemes törölni a memóriából a `deallocate` utasítással:

```
deallocate(a,b,c,d)
```

Fontos megjegyezni, hogy egy foglalható tömb számára csak akkor lehet memóriát foglalni, ha jelenleg nem `allocated`. Ha például meg akarjuk növelni egy tömb méretét, akkor először törölni kell, majd újra foglalni a tárterületet. Ha nem akarjuk, hogy elvesszenek a korábbi adatok, akkor egy másik változóba kell menteni.

Például olvassunk be ismeretlen mennyiségű adatot az a tömbbe. Az utolsó adatot a -1 érték jelölje.

```
integer, allocatable :: a(:), b(:)
integer :: n, dn, i
n=100; dn=100;               ! kezdeti tomb meret es novekmeny
allocate(a(n))               ! "a" tomb foglalas
i=0                          ! adat szamlalo 0-zasa
do                           ! adatok olvasasi ciklus eleje
   i=i+1                          ! adat szamlalo novelese
   if(i>n)then                    ! tobb mint n adat?
      allocate(b(n)); b=a;             ! eddigi adatok mentese b-be
      deallocate(a); allocate(a(n+dn)); ! "a" tomb meretet noveljuk
      a(1:n)=b                         ! az adatok visszairasa
      deallocate(b)                    ! "b" tomb torlese
      n=n+dn;                          ! uj tomb meret
   end if                         ! felteteles utasitas vege
   read(*,*)a(i)                  ! adat olvasas
   if( a(i) == -1) EXIT           ! kilepes -1 eseten
end do                       ! adat olvasasi ciklus vege
```

4.2. Programozási feladat: tetszőleges számú prím szám

Írja át a prím szám kereső programot úgy, hogy az első n szám között keressen prímeket, ahol n egy beolvasható pozitív egész szám.

Megoldás:

```
program prime
  implicit none

  logical, allocatable :: IsPrime(:) ! .true. a primekre
  integer :: i, n, s                 ! index, meret, statusz
  !-------------------------------------------------------------
  do
    write(*,*)'n='
    read(*,*) n                      ! n beolvasasa
    if(n<2) CYCLE                    ! n<2 nem jo
    allocate(IsPrime(n),stat=s)      ! IsPrime tomb foglalasa
    if(s==0) EXIT                    ! EXIT ha sikeres a foglalas
  end do
  IsPrime=.true.                     ! mindrol feltesszuk hogy prim
  do i=2,floor(sqrt(real(n)))        ! osztokkal gyok n-ig megyunk
     if(IsPrime(i)) &                ! ami meg nincs kihuzva, annak
        IsPrime(2*i : n : i)=.false. ! kihuzzuk a tobbszoroseit
  end do
  do i=2,n                           ! kiirasnal n-ig megyunk
     if(IsPrime(i)) write(*,*)i      ! kiirjuk a primeket
  end do
end program prime
```

4.3. Mutató: `pointer`

Még a foglalható méretű tömböknél is rugalmasabb adatszerkezetek hozhatók létre mutató típusú változókkel. Ezeket a változókat a `pointer` tulajdonsággal kell deklarálni:

```
  real, pointer :: a(:), b(:)
```

A mutatók valójában csak egy memória címet tartalmaznak, ami egy adott típusú változóra, jelen esetben egy dimenziós valós tömbre mutat. A deklarálással még nem hoztuk létre a változót, sőt a mutató értéke is definiálatlan. A mutató típusú változók kétféleképpen kaphatnak értéket:

- `allocate` utasítással
- `=>` mutató hozzárendeléssel

Az mutatók esetében az `allocate` utasítás a foglalható tömbhöz hasonlóan működik, például az

```
allocate(a(100))
```

létrehoz egy 100 dimenziós tömböt amire az 'a' mutató mutat. A tömb elemei ugyanúgy érhetők el, mintha 'a' egy egyszerű tömb lenne. A mutató hozzárendelésre példa:

```
b => a
```

Ezután a b mutató ugyanoda mutat, mint ahova az a. Ha most megváltoztatjuk az a elemeit, megváltoznak b elemei is és fordítva:

```
a(1)=1; b(2)=2;
write(*,*)'b(1)=',b(1),' a(2)=',a(2) ! eredmeny: 1 2
```

Lényeges különbség a foglalható tömbökhöz képest, hogy az a mutató számára új memória tömb foglalható anélkül, hogy a korábbi foglalást törölnénk! Például ha az eddigi utasítások után következik egy

```
allocate(a(-20:20))
```

akkor a egy új és definiálatlan 41 elemű tömbre mutat, míg b arra az 1000 elemű tömbre, amit az első foglalással hoztunk létre. Vigyázni kell arra, hogy egy lefoglalt tárterületre mindig mutasson mutató. A lefoglalt terület a `deallocate` utasítással szabadítható fel:

```
deallocate(a)
```

A mutatókról az `associated` függvénnyel dönthetjük el, hogy mutatnak-e valamire (ez hasonló szerepet játszik, mint az `allocated` a foglalható tömböknél. Például:

```
if(associated(a)) write(*,*)'a is associated'
```

A `nullify` utasítással érhetjük el, hogy a mutató ne mutasson semmire:

```
nullify(a,b)
```

Ez abban különbözik a `deallocate` utasítástól, hogy az a tárterület, amire a mutatók eddig mutattak nem szabadulnak fel. Ha már egyetlen mutató sem mutat egy lefoglalt tárterületre, akkor az elérhetetlenné és felszabadíthatatlanná válik,

és a memória terület csak a program lefutása után szabadul fel (amennyiben az operációs rendszer jól működik). Ezért a mutatók manipulációinál különösen vigyázni kell.

Mutatók mutathatnak közönséges változókra is, ha azokat a `TARGET` tulajdonsággal ruháztuk fel. Például:

```
integer, target       :: a(10)
integer, pointer      :: b(:), c(:), d, e
type(person), target  :: Mike
a    = (/ i, i=1,10 /)
Mike = (/ 'Mike Tyson', 33, 176.5 /)
b => a
c => a(3:10)
d => a(1)
e => Mike%age
write(*,*) b(2), c(2), d, e   ! eredmeny: 2 4 1 33
```

Mint látható, a mutató az egész tömbre, a tömb egy részére, illetve egy elemre is mutathat, valamint egy összetett típus egy mezőjére is.

4.4. Mutatók használata összetett típusokban

A mutatók egyik nagy előnye, hogy összetett típusokban önmagára mutató változót lehet létrehozni. Adjuk meg például egy személy nevét, nemét és házastársát a `person` típusban:

```
type person
  character (len=50)    :: Name      ! nev
  logical               :: IsMale    ! .true. ha ferfi
  type(person), pointer :: Spouse    ! mutato a hazastarsra
end type person
type(person), targer :: Bob, Mary
Bob%Name =  'Mr. Bob Marley';  Bob%IsMale=.true.;   Bob%Spouse => Mary
Mary%Name = 'Mrs. Bob Marley'; Mary%IsMale=.false.; Mary%Spouse => Bob
write(*,*) Bob%Spouse%Name        ! Mrs. Bob Marley
write(*,*) Bob%Spouse%Spouse%Name ! Mr. Bob Marley
```

Ha az összetett típus egyik mezőjében dinamikus méretű tömböt akarunk létrehozni, akkor nem használhatunk foglalható (`allocatable`) tömböket, mert a fordítóprogram akkor nem tudná előre, hogy az összetett típushoz mekkora tárterület tartozik, ami elrontaná az optimalizációt. Ugyanakkor lehetőség van

tömbre mutató mezőt megadni, hiszen ennek mérete ismert (egy memória cím mérete).

Például visszatérve a !!! részben hozott példához, az összetett típust definiálhatjuk tetszőleges számú kocsira:

```
type CarType
  character (len=50) :: Type, Color  ! tipus, szin
  integer            :: Age          ! kor
end type CarType
type PersonType
  character (len=50)     :: Name
  integer                :: nCar
  type(CarType), pointer :: Car(:)
end type PersonType
type(PersonType) :: Bob

Bob%nCar = 2; allocate(Bob%Car(2)) ! Bobnak ket autoja van
```

Ennél is érdekesebb egy olyan adattípus, melyben ismeretlen méretű tömb szerepel, mely saját típusára mutató elemekből áll. A Fortran 90 nem engedi meg, hogy mutatókból tömböt hozzunk létre. Az eddigi példákban mindig egy skalár mutató mutatott az egész tömbre (vagy annak egy részére).

Közvetett úton azonban megoldható a probléma. Létrehozható egy új tipus, ami csak egy mutatót tartalmaz, és ebből már lehet tömböt definiálni:

```
type PersonPtr
   type(PersonType), pointer :: Ptr
end type PersonPtr

type PersonType
   character (len=50) :: Name
   type(PersonPtr), pointer :: Children(:)
end type PersonType

type(PersonType), target :: Alice, Bob, Charlie

allocate(Alice%Childen(2))
Alice % Childer(1) % Ptr => Bob
Alice % Childer(2) % Ptr => Charlie
Bob % Name = 'Bob Marley'
write(*,*) Alice % Children(1) % Ptr % Name  ! Bob Marley
```

Mint látható egy extra `Ptr` mezőn keresztül érhetők el a gyerekek adatai. Ilyenmódon egy egész családfa felépíthető.

4.5. Programozási feladat: nevek listája

Írjon egy programot, ami sorban beolvas neveket, és ezt egy listában tárolja, majd a neveket sorban kiírja. Az utolsó nevet egy üres karakterlánc jelöli. Nem tudjuk előre, hány név lesz, a láncot dinamikusan kell felépíteni.

A lista minden eleme egy összetett típus, mely két mezőből áll: a név és egy mutató a következő elemre. Fontos, hogy az első elem elérhető maradjon!

A nevek beolvasásánál szükség lesz a

```
read(*,'(a)') Name
```

utasításra. Az `'(a)'` egy formautasítás, ami annyit mond, hogy karakterláncot olvasunk be. A `*` most nem működne jól, mert nem tudnánk vele üres nevet megadni.

Megoldás:

```
Program List
  implicit none
  Type Person                                              ! tipus deklaracio
     Character (Len=50)     :: Name
     Type (Person), Pointer :: Next
  End Type Person
  Type(Person), Pointer :: First, Thisperson, Lastperson  ! mutatok
  Character (Len=50) :: Newname                            ! uj nev
  !-------------------------------------------------------------------------
  Do                                                 ! olvasasi ciklus
     Read(*,'(a)')Newname                               ! uj nev olvasasa
     If(Newname=='') exit                               ! kilep ha ures
     Allocate(Thisperson)                               ! memoria foglalas
     Thisperson%Name=Newname                            ! nev beirasa
     If (Associated(First)) then                        ! ha mar volt elotte nev
        Lastperson%Next => Thisperson                     ! ez az utolso utani
        LastPerson => ThisPerson                          ! most mar ez az utolso
     Else                                               ! ha ez az elso nev
        First      => Thisperson                          ! ez az elso
        LastPerson => ThisPerson                          ! ez az utolso is
     Endif
  End Do                                             ! olvasasi  ciklus vege
```

```
  ! Kiiras
  ThisPerson => First                          ! Mutatot elsore allit
  Do                                           ! kiirasi ciklus
     If(.not. Associated(ThisPerson)) exit       ! kilep ha nincs kovetkezo
     write(*,*)ThisPerson%Name                   ! nev kiiras
     Thisperson=>Thisperson%Next                 ! ugras a kovetkezore
   End Do                                      ! kiirasi ciklus vege
End Program List
```

5. Alprogramok és modulok

Ezen az órán az eljárásokról és függvényekről, közös nevükön alprogramokról (subprogram) lesz szó. A főprogramon (main program) és az alprogramon kívül még egy fajta programegységgel, a modullal (module) is megismerkedünk.

5.1. Eljárás: `subroutine`

Eddig minden program egyetlen egységből, az úgynevezett főprogramból állt, melynek szerkezete

```
program ProgramName
  implicit none
  ! valtozo deklaraciok
  ...
  ! vegrehajthato utasitasok
  ...
end ProgramName
```

Ahogy a program egyre hosszabb lesz, szükségessé válik, hogy kisebb egységekre vágjuk fel. Az egyik ilyen egység az eljárás, azaz `subroutine`, melynek szerkezete

```
subroutine SubroutineName(var1, var2, ...)
  implicit none
  ! valtozo deklaraciok
  ...
  ! vegrehajthato utasitasok
  ...
end SubroutineName
```

Mint látható az eljárások szerkezete hasonló a főprogram szerkezetéhez. Lényeges különbség, hogy az eljárásnak paramétereket – más néven argumentumokat – adhatunk át. Az argumentumkat a szubrutin nevét követő zárójelben felsorolt belső változónevekkel (dummy parameters) jelöljük, a fenti példában `var1, var2 ...`. Ezeket a változókat ugyanúgy deklarálni kell, mint a szubrutin saját változóit. A szándék (`intent`) tulajdonsággal adhatjuk meg, hogy az egyes argumentumok információt adnak át a szubrutinnak (`intent(in)`, vagy információt adnak vissza a meghívó program egységnek (`intent(out)`, esetleg mindkettőt (`intent(inout)`.

Az argumentumok a szokásos változótípusokon kívül lehetnek eljárások vagy függvények is. Ezeknél az argumentumoknál a szándék tulajdonság nem adható meg. Ugyanez a helyzet mutató típusú argumentumoknál is, melyeknél nem lenne egyértelmű, hogy a szándék a mutatóra magára, vagy a mutató által elérhető adatra vonatkozik-e.

Például

```
subroutine sub(a,b,c,d,e,f)
  implicit none
  real, intent(in)                  :: a    ! bejovo informacio
  real, intent(out)                 :: b(3) ! kimeno informacio
  character (len=10), intent(inout) :: c    ! be- es kimeno informacio
  real, pointer                     :: d    ! mutato, nincs szandek
  external                          :: e    ! eljaras
  real, external                    :: f    ! fuggveny
  integer :: i                              ! belso valtozo
  !-------------------------------------------------------
  write(*,*)'a=',a               ! a hasznalhato, de nem valtoztathato
  b = (/ 1.0, 2.0, 3.0 /);       ! b bejovo erteke nem hasznalhato
  c(1:1) = 'X'                   ! c erteke valtoztathato
  d = d + 1                      ! d erteke valtoztathato
  call e(5)                      ! e eljaras meghivasa
  read(*,*) i                    ! i belso valtozo olvasasa
  write(*,*)'f(i)=',f(i)         ! f fuggveny hasznalata
end subroutine sub
```

Az `external` deklaráció, illetve tulajdonság azt jelenti, hogy a belső argumentum egy eljárás, illetve valamilyen függvény amit ezen az eljáráson kívül definiálunk. Ennél precízebben az `interface` segítségével lehet eljárás illetve függvény argumentumokat deklarálni, erre rövidesen kitérünk !!!

Egy eljárást a `call` utasitással kell meghívni, például:

```
call sub(a1+3.5, b1, c1, d1, e1, f1)
```

A meghivásnál szereplő argumentumokat külső (actual) argumentumoknak nevezzük. Mint látható, az `intent(in)` szándékú argumentum esetében nemcsak változó, de tetszőleges kifejezés is átadható az eljárás számára. Az `intent(out)`, `intent(inout)` szándékú belső argumentumoknak megfelelő külső argumentumok viszont csak változók lehetnek. Fontos még megemlíteni, hogy `d1` csak mutató típusú lehet, mivel a belső argumentum `d` is az. Viszont fordítva ez nem szükségszerű, például `b1` vagy `c1` lehetnek valós tömbre vagy karakterláncra mutató változók annak ellenére, hogy az eljáráson belül közönséges változókként lettek deklarálva. Ilyen esetben természetesen a belső argumentum a külső argumentum által mutatott változóval asszociálódik.

Ha egy eljárásnak nincs argumentuma, akkor definiálása illetve meghívása a következőképpen néz ki:

```
subroutine SimpleSub
  write(*,*)'I have no arguments'
end subroutine SimpleSub
...
call Simple
```

Egy eljárásból általában akkor térünk vissza, amikor a végrehajtás az `end subroutine SubroutineName` sorhoz ér. A `return` utasítással azonban mód van arra, hogy hamarabb visszaadjuk a vezérlést a meghívó programegységnek. Egy eljárásban tetszőlegesen sok `return` utasítás szerepelhet.

5.2. Függvény: `function`

Az eljárásoknál valamivel bonyolultabbak a függvények, melyek egy értéket adnak vissza. A függvény argumentumainak általában mind `intent(in)` szándékúnak illik lennie, hiszen az eredményt a függvény adja vissza. Ez ugyan nem kötelező, de célszerű betartani. A visszaadott érték típusa megadható a `function` előtt:

```
real function integral(xMin,xMax)    ! valos erteku fuggveny eleje
  real, intent(in) :: xMin, xMax       ! integralasi hatarok
  ...
  integral = ...                       ! fuggveny ertek megadasa
  ...
end function                         ! fuggveny vege
```

vagy a fuggveny változói között (természetesen szándékot nem kell és nem is lehet megadni, hiszen az értelemszerűen csak `intent(out)` lehet):

```
function integral(xMin,xMax)         ! fuggveny eleje
   real             :: integral         ! az integral valos tipusu
   real, intent(in) :: xMin, xMax       ! integralasi hatarok
...
```

Mint látható a függvénynek értéket kell adni a függvényen belül. A függvény argumentumlistája mögé írt `result(nev)` megadásával lehetőségünk van arra, hogy a függvényen belül más néven hivatkozzunk a függvény eredményére:

```
function integral(xMin,xMax) result(res)  ! fuggveny eleje
   real             :: res                  ! az eredmeny valos tipusu
   real, intent(in) :: xMin, xMax           ! integralasi hatarok
   ...
   res = ...                                ! eredmeny megadasa
end function integral
```

A függvényben is szerepelhet `return` utasítás. Ügyelnünk kell arra, hogy a függvény mindig értéket kapjon, pl.

```
function integral(xMin,xMax) result(res)  ! fuggveny eleje
   real             :: res                  ! az eredmeny valos tipusu
   real, intent(in) :: xMin, xMax           ! integralasi hatarok
   !-------------------------------------------------------------------
   if(xMin >= xMax)then                     ! feltetel
     res = 0.0                                 ! eredmeny megadasa
     return                                    ! visszateres
   end if
   ...
   res = ...                                ! eredmeny megadasa
end function integral
```

Egy függvény értéke lehet tömb vagy akár összetett típus is. A függvényérték ugyan nem lehet foglalható (`allocatable`) tömb, viszont lehet mutató típusú. Például:

```
function compact(a)
  implicit none
  real, intent(in) :: a(100)
  real, pointer :: compact(:)
  integer :: i, n
  ! find the number of different elements
  ...
  allocate(compact(n))
```

```
  do i=1,n
    ...
    compact(i) = ...
  end do
end function compact
```

Egy ilyen függvényt például egy mutató értékadással lehet használni:

```
  y => compact(x)
```

Pointer típusú függvényt nem lehet `external` tulajdonsággal deklarálni. Vagy explicit interfész kell (lásd 5.4 részt), vagy belső illetve modulfüggvénynek kell lennie (lásd 5.5 illetve 5.6 részeket).

Ha egy függvénynek nincs argumentuma, akkor – ellentétben az argumentum nélküli eljárásokkal – továbbra is ki kell írni az argumentumlistát tartalmazó zárójeleket:

```
real function SimpleFun()
  SimpleFun = 1.0
end function SimpleFun

program Simple
  implicit none
  real, external :: SimpleFun
  write(*,*) SimpleFun()
end program Simple
```

A függvények esetében a meghívó programegységben mindig deklarálni kell a függvény típusát, és célszerű az `external` tulajdonságot is odaírni, ami megkülönbözteti egy közönséges változótól. Ennél pontosabban interfész segítségével lehet egy függvényt deklarálni, amiről a 5.4 részben lesz szó.

5.3. Rekurzió: `recursive`

Mind az eljárások, mind a függvények meghívhatják önmagukat akár közvetlenül, akár közvetett módon más eljárásokon és függvényeken keresztül. A rekurzív alprogramok esetében a `subroutine` illetve `function` elé kell írni, hogy `recursive`. Egy klasszikus, de egyáltalán gyakorlatilag nem túl hasznos példa a faktoriális kiszámítása rekurzív módon:

```
recursive function factorial(n) result(res)
```

```
  integer             :: res
  integer, intent(in) :: n
  if(n<=1)then
     res = 1
  else
     res = n * factorial(n-1)
  end if
end function factorial
```

A `result` segítségével jól elkülönül a függvény értéke (`res`) a rekurzív függvény hívástól (`factorial(n-1)`). Természetesen ügyelni kell arra, hogy a rekurzió ne legyen végtelen. Egy alprogramban lokálisan deklarált változók minden rekurzív hívásnál újra foglalódnak, ezért csak akkor érdemes rekurzív algoritmust használni, ha a rekurzív szintek száma nem túl nagy, és a rekurzív algoritmus lényegesen egyszerűbb mint egy nem rekurzív forma. A faktoriálist nyilván sokkal egyszerűbb és hatékonyabb egy ciklussal kiszámolni.

5.4. Interfész: `interface`

Egy programegység egy másik programegységről az interfész (`interface`) segítségével kaphat az `external` tulajdonság megadásánál pontosabb információt. Az interfész az `interface` és `end interface` utasítások között egy vagy több alprogram deklarációs részéből áll. A deklarációs résznek az eljárások nevét, a függvények nevét és típusát, valamint az argumentumlistában szereplő változók nevét és típusát kell tartalmaznia. A lokális változók deklarákására az interfészben nincsen szükség. Például:

```
interface
  recursive function factorial(n) result(res)
     implicit none
     integer             :: res
     integer, intent(in) :: n
  end function factorial
  subroutine copy(a,b)
    implicit none
    real, intent(in) :: a
    real, intent(out):: b
  end subroutine copy
  ...
end interface
```

Egy tipikus alkalmazás a függvény vagy eljárás típusú argumentum deklarálása:

```
subroutine find_max(xMin,xMax,Func)
  ! Find maximum of Func between xMin and xMax
  implicit none
  real, intent(in) :: xMin, xMax
  interface
    real function Func(x)
      implicit none
      real, intent(in):: x
    end function Func
  end interface
  ....
end function find_max
```

Általában célszerű az interfészt közvetlenül az eljárás vagy függvény programszövegéből átmásolni. Egy alprogramot vagy `external` tulajdonsággal vagy interfésszel deklarálhatunk, a kettőt egyszerre nem lehet megtenni. Az interfész mindenképpen pontosabb leírást ad a külső programegységről, így a fordítóprogram könnyebben észreveszi a tipikus programhibákat, például a rosszul megadott argumentumlistát. A későbbiekben még sok olyan nyelvi eszközzel fogunk még találkozni, ami megköveteli az interfész használatát.

5.5. Belső alprogramok: `contains`

Mindeddig külső (external) alprogramokról volt szó, melyek között az adatok az argumentumokon keresztül adhatók át. A külső programegység végére írt `contains` utasítás után lehetőség van belső (internal) alprogramok definiálására:

```
subroutine ExternalSub(var1,var2,...)
  implicit none
  ! valtozo deklaraciok
  real :: HostVar
  ! utasitasok
  ...
  call InternalSub1(4)
  ...
  write(*,*)InternalFunc1()
  ...
contains
  subroutine InternalSub1(Arg)   ! belso eljaras eleje
     real, intent(in) :: Arg       ! argumentum
     integer          :: Local     ! lokalis valtozo
     write(*,*) HostVar            ! gazda programegyseg valtozoja
     write(*,*) InternalFunc1()    ! masik belso programegyseg
```

```
  end subroutinne InternalSub1   ! belso eljaras vege

  real function InternalFunc1()  ! belso fuggveny eleje
     InternalFunc1= HostVar**2     ! gazda programegyseg valtozoja
  end function InternalFunc1     ! belso fuggveny vege
  ...
end subroutine ExternalSub
```

Egy belső alprogram már nem tartalmazhat újabb belső alprogramot!

A belső alprogram az őt tartalmazó gazda- (host) programegység valamennyi változóját látja. Így az argumentum listában elegendő azokat a belső változókat felsorolni, melyek értékét a különböző meghívásoknál más-más külső változóhoz vagy kifejezéshez asszociáljuk. A rövidebb argumentumlisták használáta nemcsak a programírónak kényelmes, de hatékonyabb programot is eredményez, mert kevesebb változót kell átadni, illetve a fordító program gyakran a belső alprogram meghívását az alprogram utasításaival tudja helyettesíteni (in-line), ami még hatékonyabb végrehajtást eredményez.

A gazda programegység illetve a többi belső alprogram természetesen nem látja egy belső alprogram lokális változóit. Ha a belsőprogramegységben deklarált lokális változó neve megegyezik egy a gazdaprogramegységben deklarált változó nevével, akkor a belsőprogramegység csak a saját változójához fér hozzá. Érdemes még megemlíteni, hogy a belső alprogramra is vonatkozik a gazdaprogramegységben szereplő `implicit none` utasítás, így azt nem szükséges megismételni, és így a lokális változókat természetesen deklarálni kell.

Minden külső programegység csak a saját belső alprogramjait tudja használni, más külső programegységek belső eljárásaihoz és függvényeihez nincs hozzáférés. Ez lehetőséget ad arra, hogy ugyanazon a néven többféle belső alprogramot használjunk. Természetesen egy gazdaprogramegységen belül a belső alprogramok egymást meg tudják hívni.

Belső alprogramok természétesen nem deklarálhatók `external` tulajdonsággal, és interfészre sincs szükség, hiszen a belső alprogramról a meghíváshoz szükséges minden információ rendelkezésre áll a programegység fordításakor.

5.6. Modul: module

A külső programegységek közti kommunikációt az argumentum lista mellett modulokkal tehetjük egyszerűbbé. Egy modul ugyanazon a szinten áll mint a többi programegység, és szerkezete is hasonló:

```
module ModExample
```

```
  implicit none
  ! tipus valtozo es interfesz deklaraciok
  ! tetszoleges szamban es sorrendben
  type ModType
     real :: a
     integer :: i(4)
  end type ModType

  real          :: ModVar1
  type(ModType) :: ModVar2

  interface
     real function erf(x)
        implicit none
        real, intent(in) :: x
     end function erf
  end interface
  ...
contains
  ! modul programegysegek
  function ModFunc(FuncVar)
     ...
  end function ModFunc
  subroutine ModSub
     ...
  contains
    subroutine InternalSub
      ...
    end subroutine InternalSub
  end subroutine ModSub
  ...
end module ModExample
```

Mint látható a modulok a változó, típus és interfész deklarációkon kívül moduleljárásokat és modulfüggvényeket, azaz modulalprogramokat is tartalmazhatnak. Ezeknek lehetnek belső alprogramjai. A modulalprogramok természetesen hozzáférnek a modulban deklarált változókhoz és típusokhoz és egymást is meg tudják hívni.

A modult a `use` utasítással lehet egy programegységbe vagy egy másik modulba beolvasni. A `use` utasításoknak a programegység illetve modul legelején, még az `implicit none` előtt kell szerepelnie. Például

```
program Example
  use ModExample
```

```
  implicit none
  ...
  ModVar1 = erf(3.0)
  call InternalSub
  ...
  contains
    subroutine InternalSub
      use MyMod
      write(*,*)ModVar1, MyModvar1
    end subroutine InternalSub
end program
```

A programegység a használt modulok összes változójához, típusához, interfészéhez, és modulalprogramjához hozzá tud férni. Ugyanez vonatkozik a programegység belső alprogramjaira is, azaz ezekben nem szükséges megismételni a use utasítást. Természetesen egy belső alprogram is használhat modulokat. Ennek tartalmához a gazdaprogramegység és a többi belső alprogram csak akkor férnek hozzá, ha ezek is használják ugyanezt a modult.

Maguk a modulok is használhatnak más modulokat, de természetesen a rekurzió nem megengedett, azaz egy modul nem használhatja önmagát. A gyakorlatban viszonylag gyakran szükség van arra, hogy egy modul egy másik modult használjon. Például az egyik modul a típusokat deklarálja, a másik a változókat, amik ilyen típusúak:

```
module ModPersonType
  implicit none
  type PersonType
     character (len=50) :: name
     integer            :: age
  end type PersonType
end module ModPersonType

module ModPeople
  use ModPersonType
  implicit none
  integer, parameter :: nPeople
  type(PersonType)   :: People(nPeople)
end module ModPeople

program list_people
  use ModPeople
  implicit none
  integer :: iPeople
  do iPeople=1,nPeople
```

```
    call read_person(People(iPeople))
  end do
  call sort_people
  ...
end program list_people

subroutine read_person(SomeOne)
  use ModPersonType
  implicit none
  type(PersonType), intent(out) :: SomeOne
  write(*,*) 'Name='
  read(*,*)  SomeOne%Name
  ...
end subroutine read_person

subroutine sort_people
  use ModPeople
  implicit none
  type(PersonType) :: Tmp
  ! sort people by name
  ...
end subroutine sort_people
```

Mint látható a `read_person` eljárásnak csak a `ModPersonType` modulban deklarált `PersonType` típusra, míg a `sort_people` eljárásnak a `ModPeople` modulban deklarált `People` tömbre is szüksége van. Mivel a `ModPeople` modul használja a `ModPersonType` modult, a `sort_people` eljárásban a `ModPeople` modulon keresztül hozzaférünk a `PersonType` típushoz is. A modulok egymásbaágyazhatósága hasonlatos (de nem azonos) az objektumorientált nyelvekben szokásos öröklődéshez (inheritance).

A program fordításánál lényeges, hogy egy modul előbb szerepeljen mint a modult használó programegység. Mint a 6.2 részben látni fogjuk, erre akkor is ügyelni kell, ha a modul és a modult felhasználó programegységet külön fordítjuk le.

5.7. Programozási feladat: emberek listája alprogramokkal

Fejezze be az előző részben vázolt `list_people` programot. Írjon a `read_person` eljárást kiegészítő `write_person` eljárást, ami kiírja egy személy adatait és ezt hívja meg a főprogramból a `sort_people` rendezés után. Próbálja ki hogyan lehetne a `read_person` és `write_person` eljárásokat külső, a főprogramban definiált belső, illetve `ModPersonType` modulban definiált moduleljárás formájában

megírni.

Megoldás külső eljárásokkal:

```
program list_people
  use ModPeople
  implicit none
  integer :: iPeople
  do iPeople=1,nPeople
    call read_person(People(iPeople))
  end do
  call sort_people
  do iPeople=1,nPeople
     call write_person(people(iPeople))
  end do
end program list_people

subroutine sort_people
  use ModPeople
  implicit none
  integer :: i,j
  type(PersonType) :: TmpPerson
  do i=2,nPeople
     do j=1,i-1
        if(people(i)%Name < people(j)%Name)then
           TmpPerson=people(i)
           people(i)=people(j)
           people(j)=TmpPerson
        end if
     end do
  end do
end subroutine sort_people

subroutine write_person(Someone)
  use ModPersonType
  implicit none
  type(PersonType), intent(in) :: SomeOne
  write(*,*) 'Name=',SomeOne%Name
  write(*,*) 'Age=',SomeOne%Age
end subroutine write_person
```

A megoldás belső illetve moduleljárásokkal csak annyiban különbözik, hogy az eljárások a megfelelő `contains` utasítás mögé kerülnek, valamint a `use` és `implicit none` utasítások az eljárásokból törölhetők.

6. Alprogramok és modulok használata

Ennek az órának a célja az eddigi ismeretek elmélyítése, az alulról építkező (bottom-up) programozási stílus alkalmazása, valamint a `Makefile` és a `make` program használatának elsajátítása.

6.1. Programozási feladat: integrálás

Írjon egy `integral` programot, ami az `exp` függvényt kiintegrálja az `xMin` és `xMax` változók által definiált intervallumban. A numerikus integráláshoz ossza fel az intervallumot `n` részre. Olvassa be az `xMin, xMax, n`, számítsa ki az integrált, majd írja ki a numerikus és összehasonlításképpen a numerikus eredményt is.

```
program integral
  implicit none
  real :: xMin, xMax, Dx, Result
  integer :: n, i
  !---------------------------
  write(*,*)'xMin, xMax, n='
  read(*,*)xMin, xMax, n
  Result=0.0
  Dx = (xMax-xMin)/n
  do i=1,n
     Result = Result + Dx*exp(xMin + i*Dx)
  end do
  write(*,*)'Numerical result=',Result
  write(*,*)'Analytic  result=',exp(xMax)-exp(xMin)
end program integral
```

Próbálja ki a programot, például:

```
a.out
 xMin, xMax, n=
0 1 100
 Numerical result=   1.7268876
 Analytic  result=   1.7182817
```

Az alulról építkező programozás lényege, hogy egy működő programot fejlesztünk tovább, miközben folyamatosan ellenőrizzük, hogy a program továbbra is működik.

Az `xMin, xMax, n` változók beolvasását tegye át a `read_parameters` belső eljárásba, az integrálást pedig az `integrate` külső függvénybe. A beolvasásnál ellenőrizze, hogy $xMax > xMin$ és hogy $n > 0$.

```
program integral
  implicit none
  real :: xMin, xMax
  integer :: n
  real, external :: integrate
  !---------------------------
  call read_parameters
  write(*,*)'Numerical result=',integrate(xMin, xMax, n)
  write(*,*)'Analytic  result=',exp(xMax)-exp(xMin)
contains
  subroutine read_parameters
    do
       write(*,*)'xMin,xMax='
       read(*,*)xMin,xMax
       if(xMax>xMin) exit
       write(*,*)'Error: xMax <= xMin !!!'
    end do
    do
       write(*,*)'number of steps='
       read(*,*) n
       if(n>0) exit
       write(*,*)'Error: number of steps < 1 !!!'
    end do
  end subroutine read_parameters
end program integral
!=============================================================
real function integrate(xMin,xMax,n)
  implicit none
  real, intent(in)    :: xMin, xMax
  integer, intent(in) :: n
  integer :: i
  real :: Dx
  !-----------------------------------
  Dx = (xMax-xMin)/n
  integrate=0.0
  do i=1,n
     integrate = integrate + Dx*exp(xMin + i*Dx)
  end do
end function integrate
```

Probáljuk ki, hogy a rosszul megadott változókat elfogadja-e a program:

```
pgf90 integral.f90
a.out
 xMin,xMax=
1 0
 Error: xMax <= xMin !!!
 xMin,xMax=
0 1
 number of steps=
-1
 Error: number of steps < 1 !!!
 number of steps=
100
 Numerical result=   1.7268876
 Analytic  result=   1.7182817
```

A külső `integrate` függvényt általánosítsuk úgy, hogy tetszőleges $\Re \to \Re$ függvényt tudjon integrálni. Ehhez használjunk függvény argumentumot:

```
real function integrate(xMin,xMax,n,Func)
  implicit none
  real, intent(in)    :: xMin, xMax
  integer, intent(in) :: n
  interface
    real function Func(x)
      implicit none
      real, intent(in) :: x
    end function Func
  end interface
  integer :: i
  real :: Dx
  !-------------------------------------
  Dx = (xMax-xMin)/n
  integrate=0.0
  do i=1,n
     integrate = integrate + Dx*Func(xMin + i*Dx)
  end do
end function integrate
```

Az uj integráló eljárást nem hívhatjuk meg közvetlenül az `exp` függvénnyel, mert a függvény argumentum nem lehet a nyelvben definiált (intrinsic) függvény. Természetesen amúgy sem az exponenciális függvény integrálása a végső cél. Hogy a működőképességet továbbra is ellenőrizni tudjuk, definiáljunk egy `MyExp` függvényt:

```
real function MyExp(x)
```

```
  implicit none
  real, intent(in) :: x
  MyExp = exp(x)
end function MyExp
```

A főprogramban a `MyExp` és az `integrate` függvényeket vagy `real, external` típusúként kell deklarálni, vagy teljes interfésszel. Az `integrate` függvény meghívása a következőképp módosul:

```
   write(*,*)'Numerical result=',integrate(xMin, xMax, n, MyExp)
```

Ismét ellenőrizzük le, hogy a program működik-e!

Definiáljunk más függvényeket (pl. `MyCos`). Olvassuk be a függvény nevét, és a megfelelőt integráljuk ki. Ehhez vezzessük be a `NameFunc` karakter változót, aminek értéke 'c' (cos) vagy 'e' (exp) lehet. Ezt a `read_parameters` eljárásban olvassuk be, és ellenőrizzük. A 'C' illetve 'E'-t javítsuk ki kisbetűre, egyéb hibáknál kérjük újra az adatot.

A `MyCos` függvényt pontosan úgy lehet megadni, mint a `MyExp`-t. A főprogramban a deklaráljuk a

```
  character :: NameFunc
```

változót, amit a `read_parameters` eljárásba a következő ciklussal olvasunk be:

```
    do
       write(*,*)'Name of function (e=exp, c=cos)='
       read(*,*) NameFunc
       select case(NameFunc)
       case('c','C')
          NameFunc='c'
       case('e','E')
          NameFunc='e'
       case default
          write(*,*)'Error: unknown NameFunc !!!'
          cycle
       end select
       exit
    end do
```

Végül a főprogramban is a `select case` kontrol utasítással választjuk ki, hogy melyik függvényt integráljuk, illetve, hogy mi az analitikus megoldás:

```
  select case(NameFunc)
  case('c')
     write(*,*)'Numerical result=',integrate(xMin, xMax, n, MyCos)
     write(*,*)'Analytic  result=',sin(xMax)-sin(xMin)
  case('e')
     write(*,*)'Numerical result=',integrate(xMin, xMax, n, MyExp)
     write(*,*)'Analytic  result=',exp(xMax)-exp(xMin)
  end select
```

Az eddigi integrálási módszerünk meglehetősen pontatlan, elsőrendű, azaz a hiba `Dx` lépésközzel egyenesen arányosan csökken. A másodrendű trapéz illetve a harmadrendű Simpson módszerrel sokkal pontosabb eredményt kaphatunk. Az integrálási módszert a `NameMethod` változóba olvassuk be, értéke lehet 'l' (lineáris), 't' (trapéz) és 's' (simpson). A változó deklarálása és beolvasása teljesen analóg a `NameFunc` változóéval, arra azonban vigyázni kell, hogy a Simpson módszer csak páros n esetén működik. Az `integrate` függvény argumentumlistáját kiegészítjük a `NameMethod` változóval, és a függvényt átírjuk:

```
real function integrate(xMin,xMax,n,Func,NameMethod)
...
  character, intent(in) :: NameMethod
...
  select case(NameMethod)
  case('l')
     integrate = 0.0
     do i=1,n
        integrate = integrate + Func(xMin + i*Dx)
     end do
     integrate = Dx * integrate
  case('t')
     integrate=0.5*(Func(xMin)+Func(xMax))
     do i=1,n-1
        integrate = integrate + Func(xMin + i*Dx)
     end do
     integrate = Dx * integrate
  case('s')
     if(mod(n,2)/=0)stop 'Simpson rule needs an even number of intervalls!'
     integrate=Func(xMin)+Func(xMax)
     do i=1,n-1,2
        integrate=integrate+4*func(xMin + i*Dx)
     end do
     do i=2,n-2,2
        integrate=integrate+2*func(xMin + i*Dx)
     end do
     integrate=integrate*Dx/3.
```

```
  end select
end function integrate
```

Végül helyezzük az `integrate` függvényt az `xMin, xMax, n, NameMethod` integrálási paraméterekkel együtt egy `ModIntegrate` modulba, a `MyExp, MyCos` függvényeket pedig egy `ModFunc` modulba. Ezzel az `integrate` függvény meghívása leegyszerűsödik, hiszen csak a az integrálandó függvényt kell átadni. A főprogramban nincs szükség `external` deklarációkra, hiszen a modulfüggvények ismertek. Így a főprogram a következő lesz:

```
program integral
  use ModIntegrate
  use ModFunc
  implicit none
  real :: Result, Analytic
  character :: NameFunc
  !---------------------------
  call read_parameters
  select case(NameFunc)
  case('c')
     Result = integrate(MyCos)
     Analytic = sin(xMax)-sin(xMin)
  case('e')
     Result = integrate(MyExp)
     Analytic = exp(xMax)-exp(xMin)
  end select
  write(*,*)'Numerical result=',Result
  write(*,*)'Analytic  result=',Analytic
  write(*,*)'Error           =',abs(Result-Analytic)

contains
  subroutine read_parameters
    ...
end program integral
```

Az integrálandó függvényeket tartalmazó modul

```
module ModFunc
contains
  real function MyExp(x)
    implicit none
    real, intent(in) :: x
    MyExp = exp(x)
  end function MyExp
```

```
  real function MyCos(x)
    implicit none
    real, intent(in) :: x
    MyCos = cos(x)
  end function MyCos
end module ModFunc
```

végül az integrálási paramétereket és az integrálási eljárást tartalmazó modul:

```
module ModIntegrate
  implicit none
  real :: xMin, xMax
  integer :: n
  character :: NameMethod
contains
  real function integrate(Func)
    implicit none
    ...
end module ModIntegrate
```

Ezzel az egydimenziós integráló modul fejlesztését befejeztük.

6.2. Több állomány fordítása: `Makefile, make`

Ahogy a program egyre hosszabb lesz, egyre nehezebb áttekinteni a programszöveget tartalmazó fájlokat. A fordító programnak is egyre tovább tart a program fordítása. Ha az egész program egyetlen fájlban van, akkor minden változtatás után le kell fordítani az egész programot.

Célszerű tehát egy programot több fájlban tárolni. Egy-egy fájlban egy vagy több programegység (főprogram, eljárás, függvény vagy modul) helyezehető el. Az egyes fájlok egyszerre is fordíthatók:

```
pgf90 -o prog.exe file1.f90 file2.f90 ...
```

Ügyelni kell arra, hogy a modulok az őket használó programegységek előtt kerüljenek sorra.

Az egyes fájlok külön-külön is fordíthatók úgynevezett tárgykóddá (object files) melyek összefűzhetők (link) a végrehajtható (executable) programba:

```
pgf90 -c file1.f90    # --> file1.o
```

```
pgf90 -c file2.f90    # --> file2.o
...
pgf90 file1.o file2.o ...
```

A `-c` azt jelenti, hogy csak fordítani (compile) kell. A modulokat most is előbb kell fordítani, mint az őket használó programegységeket, ugyanis a modulban lévő információkra már a tárgykód létrehozásához is szükséges.

Ha a programban változtatást hajtunk végre, akkor csak azokat a fájlokat kell újrafordítáni, melyek a változtatástól függenek. A `make` program használáta automatizálja az egyes fordítási lépések végrehajtását, és a függési relációkat is figyelembe tudja venni. A `make` program általában a `Makefile` vagy `makefile` nevű fájlban lévő információkat használja, de megadható más fájlnév is a `-f` kapcsoló segítségével:

```
make -f Makefile.mine target1 target2 ...
```

A `target1, target2, ...` a `make` program által létrehozandó fájlokat, illetve absztrakt célokat (target) jelöli. Ha nem adunk meg célt, akkor a `Makefile`-ban szereplő első célt hozza létre a `make`. Hogy az egyes célok eléréséhez mit kell tenni, azt a `Makefile` tartalmazza:

```
target1 target2 ... targetN : depend1 depend2 ... dependK
        action1
        action2
        ...
        actionM
```

A cél(ok) után mindig kell kettőspontot tenni, amit azon egyéb célok listája követ, melyektől függ az adott cél. Ez a lista lehet üres is. A következő sorokban szerepelnek azok az utasítások, amiket a cél megvalósítása érdekében végre kell hajtani. Ezek a sorok a TAB karakterrel kell, hogy kezdődjenek, így a `Makefile` létrehozására csak olyan szövegszerkesztő alkalmas, amelyik a TAB karaktert nem helyettesíti szóközökkel! Az is elkőfordul, hogy egyetlen utasítás sem követi a célt, ilyenkor a függések ellenőrzése és létrehozása a cél. Példák:

```
test: integral.exe
        integral.exe

integral.exe: integrate_mod.o integral.o
        pgf90 -o integral.exe integrate_mod.o integral.o

integrate_mod.o: integrate_mod.f90
```

```
        pfg90 -c integrate_mod.f90

integral.o: integral.f90 integrate_mod.o
        pfg90 -c integral.f90

clean:
        rm -f *.o

cleanall: clean
        rm *.exe
        rm *~
```

Ha a `Makefile` a fenti szabályokat tartalmazza, akkor például a

```
make integral.exe
```

utasítás hatására először az első függést, azaz a `integrate_mod.o` kell elkészíteni, ami függ a `integrate_mod.f90` fájltól, és a `pgf90 -c integrate_mod.f90` utasítással lehet létrehozni. Ezután a második függésre kerül a sor, azaz a `integral.o` tárgykódot hozza létre a `pgf90 -c integral.f90` utasítással, végül a két tárgykódot összefűző `pgf90 -o integral.exe integrate_mod.o integral.o` utasítás hajtódik végre, ami létrehozza a célt, azaz az `integral.exe` végrehajtható programot. Ha mégegyszer kiadjuk a `make integral.exe` utasítást, akkor a

```
make: 'integral.exe' is up to date.
```

üzenetet kapjuk. Ha megváltoztatjuk az `integral.f90` fájlban a forráskódot, akkor a `make integral.exe` utasítás csak az `integral.f90` fájlt fordítja újra, majd az új `integral.o` tárgykódot összefűzi a már meglevő `integrate_mod.o` tárgykóddal.

A `make clean` letörli a tárgykódokat a UNIX `rm` (remove) utasításának a segítségével. A `make cleanall` először elvégzi a `clean` célhoz tartozó utasításokat, majd letörli a végrehajtható programokat és az `emacs` szövegszerkesztő által létrehozott biztonsági másolatokat.

Általában a `make` visszaírja a képernyőre az elvégzendő utasításokat. Ez néha zavaró, például ha szöveget írunk ki a képernyőre az `echo` utasítással. Ilyenkor a karaktert kell az utasítás elé írni:

```
help:
        @echo "Kiirom ezt a szoveget,"
        @echo "de nem az echo utasitast!"
```

Folytatósorokat a sorvégére helyezett \ karakterrel lehet írni, míg megjegyzéseket a # karakter után

```
# Ez egy megjegyzes
target1: \
        object1 \
        object2
action1; action2 \
        action3 # ez is egy megjegyzes
```

A sorok végére lehet megjegyzést írni, da a folytatósorok végére nem.

A Makefile használátát változókkal lehet megkönnyíteni. Például a tárgykódok listáját elég egyszer leírni, ha bevezetjük az `OBJ` változót:

```
OBJECTS = integrate_mod.o integral.o
```

A változó neve állhat kis és nagybetűkből, számokból, és aláhúzás karakterekből. Általában nagy betűket szokás használni.

A változók értékét a $ karakter mögé gömbölyű vagy kapcsos zárójelek közé írt változónévvel kapjuk meg. Például:

```
integral.exe: ${OBJECTS}
        pgf90 -o integral.exe ${OBJECTS}
```

Ha ugyanezt a `Makefile`-t egy másik gépen is akarnánk használni, ahol a Fortran fordítót másképp hívják, akkor mindenhol ki kellene javítani a `pgf90`-t egy másik névre. Ezt megelőzendő érdemes a fordítót és a hozzátartozó kapcsolókat is változókba írni:

```
FTN   = pgf90
FLAGS = -O2

integral.exe: $(OBJECTS)
        $(FTN) $(FLAGS) -o integral.exe $(OBJECTS)
```

A változóknak a `Makefile`-ban megadott értéke felülírható a `make` meghívásakor, például

```
make FTN=f90 FLAGS=-O3 integral.exe
```

Ha nagyon sok fájlból áll össze a program, akkor érdemes bevezetni általános szabályt, ami megmondja, hogy egy adott végződésű fájlból hogyan állítható elő egy másik. Először egy speciális `.SUFFIXES` változóban fel kell sorolni azokat a végződéseket, amire vonatkozóan általános szabályokat kívánunk megadni. Mivel a `make` eleve használ bizonyos általános szabályokat, a legegyszerűbb először törölni a változót, majd felsorolni az általunk használt végződéseket. Például:

```
.SUFFIXES:
.SUFFIXES: .o .f90
```

Ezután a következő formában írhatjuk fel az általános szabályt:

```
.f90.o:
        $(FTN) $(FLAGS) -c $*.f90
```

A cél (target) helyére tehát a kiinduló fájl és a célfájl végződései kerülnek közvetlenül egymás mellé. A kettőspont mögött nem adható meg függés, ugyanis a függés mindig az, hogy a célfájl függ a forrásfájltól. A speciális `$*` változó tartalmazza a fájlnevek kiegészítés nélküli törzsét, ez használható fel a szabály felírására, azaz jelen példában a `$*.f90` forrásfájlból hozzuk létre a `$*.o` célfájlt.

Ha egy célra az általános szabályon kívül egy konkrét, utasításokat is tartalmazó szabály is vonatkozik, akkor az utóbbi hajtódik végre:

```
.f90.o:
        $(FTN) $(FLAGS) -c $*.f90

special.o: special.f90
        $(FTN) -O0 -c $*.f90
```

A `make akarmi.o` az általános szabály szerint fog eljárni, és a `$FLAGS` változóban használt kapcsolókat fogja használni a fordításhoz, viszont a `make special.o` a speciális szabály alapján a `-O0` kapcsolóval, azaz optimalizalizáció nélkül fordít.

Egy célhoz több olyan szabály is rendelhető, amelyik függéseket sorol fel, de csak egy szabály tartalmazhat utasításokat. Így például az általános szabályt kiegészíthetjük egy

```
.f90.o:
        $(FTN) $(FLAGS) -c $*.f90

${OBJECTS}: Makefile
```

szabállyal, ami azt jelenti, hogy ha a `Makefile`-t megváltoztattuk, akkor az összes az `OBJECTS` változóban felsorolt tárgykódot újra kell csinálni. Ebben az esetben a fordításhoz az általános szabályt fogja a `make` alkalmazni.

6.3. Programozási feladat: make és Makefile használata

Vágja három részre az `integral.f90` fájlt. Az `integral_int.f90` fájlba tegye a `ModIntegrate` modult, a `integral_func.f90` fájlba a `ModFunc` modult, és csak a főprogramot hagyja benn az `integral.f90` fájlban. Készítse el a `Makefile`-t!

Megoldás:

```
FTN = pgf90  # Fortran fordito program
FLAGS = -O2  # kapcsolok (optimalizacio)

# targykodok listaja
OBJ = integral.o integral_int.o integral_func.o

# vegrehajthato program
integral.exe: $(OBJ)
        $(FTN) $(FLAGS) -o integral.exe $(OBJ)

# vegzodesek listaja a targykodokra vonatkozo altalanos szabalyhoz
.SUFFIXES:
.SUFFIXES: .f90 .o

# altalanos szabaly targykodokra
.f90.o:
        $(FTN) $(FLAGS) -c $*.f90

# specialis fugges a moduloktol
integral.o: integral_int.o integral_func.o

# targykodok torlese
clean:
        rm $(OBJ)
```

7. Konstansok és változók kezdeti értékének megadása

7.1. Konstansok deklarálása: `parameter`

Konstansok használatával elkerülhető számos tipikus programozási hiba. Például ha ugyanazt a konstanst használjuk a tömbök méretének megadására és a ciklusváltozó maximális értékére, akkor biztos, hogy a két érték ugyanaz lesz. Ha számokat használunk, amiket időnként megváltoztatunk, akkor könnyen el lehet feledkezni az összes összetartozó érték szinkronban tartásáról, arról nem is beszélve, hogy ez sok felesleges munkával is jár.

Konstansok használhatók matematikai és fizikai konstansok értékének tárolására is, pl.

```
Pi            = 3.1415926535
SpeedOfLight  = 2.9979e8
```

A deklarációban szerepelhetnek korábban definiált konstansok és bizonyos műveletek, függvények is. A műveletekre az a megkötés, hogy csak egész kitevő használható, míg a függvények argumentuma és értéke csak egész vagy karakter típusú lehet. Néhány transzformációs függvény is használható, ezek közül a már említett `reshape` a leghasznosabb. Néhány példa:

```
! Konstansok
character (len=*), parameter :: quote='To be, or not to be'
integer, parameter           :: n=10, n2=n**2, m2=67, MaxNM2=max(n2,m2)
real, parameter              :: Pi=3.1415926535, TwoPi=2*Pi
integer, dimension(3,3), parameter :: &
    Kronecker=reshape( (/1,0,0,       &
                         0,1,0,       &
                         0,0,1 /), (/3,3/) )
! Valtozok
real                  :: a(n,n), b(n2)
character(len=MaxNM2) :: text
```

Az első karakterkonstans deklarációnál a karakterlánc hosszát a `(len=*)` utasítással adhattuk meg, ugyanis a konstans értékéből úgyis kiderül a hossz. Ha konkrét hosszat adunk meg, akkor ha az rövidebb a konstansnál, akkor a konstans vége le lesz vágva, ha hosszabb, akkor szóközökkel töltődik fel.

7.2. Változók kezdeti értékének megadása

Változók kezdeti értéke ugyanúgy adható meg, mint a konstansoké, csak a `parameter` tulajdonságot kell elhagyni. A változó kezdőértéke természetesen a program végrehajtása során megváltoztatható. Tömbök esetén van néhány különbség a konstansokhoz képest. Egy egész tömb kezdő értéke megadható egy skalárral (ennek konstans tömb esetén nem sok értelme lenne, hiszen egy csupa egyforma elemből álló tömb, aminek az értéke nem változhat meg nem túl hasznos):

```
real, dimension(10,10) :: a=0.0
```

A tömb egy részét a `data` utasítással lehet feltölteni. Ez az utasítás a deklaráció után következik, és a feltöltendő változóból, valamint két törtvonal közé írt adatokból áll:

```
real :: a(10,10)
data a(:,1) /1.0, 2.0, 5*7.0, 3*-2.0/
integer, parameter :: N=2*2
data a(3:4,3:4) /N*1.23e2/
```

Az ismétlődő adatokat egy konstanssal való szorzás formájában lehet tömören jelölni. A `data` utasításnál nincs jelentősége a tömb alakjának, csak az adatok mennyiségének kell megegyeznie a feltöltendő tömbelemek számával.

7.3. Változók megőrzése: `save`

A főprogramban deklarált változók értéke nem veszhet el, azonban az eljárásokban és függvényekben deklarált lokális változók értéke általában nem őrződik meg a vezérlés visszatérte után. Ha a deklarációban szerepel a `save` tulajdonság, akkor a változó értéke megmarad. Ugyanez történik akkor is, ha a változó kezdeti értéket kap. Például írjuk ki, hogy egy eljárást összesen eddig hányszor hívtak meg:

```
subroutine sub
  implicit none
  integer :: counter=0
  counter=counter+1
  write(*,*)'Total number of calls:', counter
end subroutine sub
```

Fontos megjegyezni, hogy a kezdeti értékadás csak az első meghívásra vonatkozik. Ha mindig egy adott változóértékkel akarjuk kezdeni az eljárást, akkor

nem a deklarációban, hanem a végrehajtandó utasítások között kell az értéket megadni. Ha ezeknek az értékeknek a kiszámítása nagyon számításigényes, vagy egy kezdeti értékadás olyan műveletet igényel, ami nem végezhető el a deklarációs részben, akkor ezt a következőképpen oldható meg:

```
subroutine sub(n)
  implicit none
  integer, intent(in) :: n
  logical :: initialized=.false.  ! else hivaskor meg hamis
  real, save :: a,b               ! elmentendo valtozok
  real, allocatable, save :: c(:) ! elmentendo foglalhato tomb
  real, pointer, save :: d(:)     ! elmentendo mutato
  if(.not.initialized)then        ! ha meg nem volt meg az ertekadas
     a=sin(1.0)                      ! ezt nem lehet deklaracioba
     b=cos(1.0)                      ! irni, mert valos fuggveny
     allocate(c(n),d(n))             ! tarfoglalas
     c=0.0; d=1.0                    ! ertekadas
     initialized=.true.              ! tovabbi hivasokkor mar igaz
  endif
  ...
```

Mint látható a `save` tulajdonság foglalható `allocatable` tömbök illetve mutatók esetén is szükséges, ha a változókat a következő eljáráshívásnál újra akarjuk használni.

Rekurzív eljárásokban és függvényekben a `save` tulajdonsággal deklarált változók, valamint a felhasznált modulokban deklarált változók minden rekurziós szintről látszanak, és értéküket megőrzik ellentétben a `save` tulajdonság nélküli lokális változókkal, melyek minden rekurziós szinten újabb tárterületet kapnak.

Modulokban deklarált változók esetén is szükséges lehet a `save` tulajdonság megadása. Ha a program futása során előfordul, hogy a modult éppen egyetlen aktív eljárás vagy függvény sem használja, akkor a modulban lévő `save` tulajdonság nélkül deklarált változók értéke elveszhet. Ha egy modult maga a főprogram is használ, akkor ez nem fordulhat elő.

8. Dinamikus változók, általánosított alprogramok

8.1. Dinamikus tömbök és karakterláncok eljárásokban és függvényekben

Egy eljárás vagy függvény argumentum listájában szereplő belső változó vagy a külső változóhoz asszociálódik (`intent(out)` és `intent(inout)`) vagy dinamikusan foglalt tárterületet kap (`intent(in)` és a külső argumentum egy kifejezés). Ez lehetővé teszi, hogy az argumentumban szereplő tömb és/vagy karakterlánc méretét dinamikusan határozzuk meg.

Az egyik lehetőség, hogy a változó méretét egy másik argumentumban adjuk át:

```
subroutine sub1(n,a,b,c)
  implicit none
  integer, intent(in)          :: n
  real, intent(in)             :: a(n)
  real, intent(out)            :: b(-n:n,0:n+1)
  character (len=n),intent(in) :: c
...
```

Figyelni kell arra, hogy a méretet előbb kell deklarálni, mint az azt használó egyéb változókat. Természetesen ez nem jelenti azt, hogy az argumentum listában is előrébb kellene szerepelni a méretnek. Ha az eljárást a belül deklaráltnál nagyobb méretű tömbbel, vagy karakterlánccal hívjuk meg, akkor csak a változó egy részéhez lesz hozzáférésünk. Ezt gyakran használják is főleg Fortran 77-hez szokott programozók. A fordított eset, amikor külső változó kisebb mint a belső, azonban már súlyos programhiba, ami általában memória hozzáférési hibához (segmentation fault) vezet.

Előfordulhat, hogy az eljárában szükségünk van egy másik tömbre, aminek ugyancsak az argumentumban adott a mérete. Ezt is egyszerűen deklarálhatjuk, és automatikus (automatic) tömbnek vagy karakter láncnak nevezzük. Az elnevezés arra utal, hogy az eljárás automatikusan lefoglalja a változóhoz tartozó tárterületet a végrehajtás kezdetén, a végén pedig felszabadítja. Egy automatikus tömb vagy karakter lánc ennek megfelelően nem deklarálható a `save` tulajdonsággal. Példa:

```
subroutine sub1(n,a,c)
  implicit none
  integer, intent(in)          :: n
```

```
  real, intent(in)             :: a(n)
  character (len=n),intent(in) :: c
  real                         :: b(-n:n) ! automatikus tomb
  character (len=n+2)          :: c2      ! automatikus karakterlanc
...
```

Egy függvény változó mérete is megadható az argumentumlistában, például a következő függvény megdupláz egy n hosszúságú karakterláncot:

```
function duplaz_n(n,c)
  implicit none
  integer, intent(in) :: n
  character (len=n), intent(in) :: c
  character (2*n) :: duplaz_n
  !-----------------------------------
  duplaz_n = c//c
end function duplaz_n
```

Dinamikus méretű változók a méret explicit átadása nélkül is létrehozhatók, ugyanis a külső argumentummal is meg lehet határozni a változó méretét. Ennek megfelelően a belső változót átvett alakú tömbnek (assumed shape array), illetve átvett hosszúságú karakterláncnak (assumed length string) nevezzük. Az átvett alakú tömb esetében megadható az alsó index értéke (alapértelmezésben 1). Például:

```
subroutine sub2(a,b,c)
  implicit none
  real, intent(in)                 :: a(:)     ! atvett alaku tomb
  real, intent(out)                :: b(0:,0:) ! atvett alaku tomb
  character (len=*),intent(in)     :: c        ! atvett hosszu karakterlanc
  real, dimension(-size(a),size(a) :: a2       ! automatikus tomb
  character (len=len(c)+2)         :: c2       ! automatikus karakterlanc
...
```

Mint látható az automatikus tömbök deklarációjában a tömb méretét megadó size függvényt, míg az automatikus karakterlánc deklarációjában a karakterlánc hosszát megadó len függvényt használtuk fel. Ha a sub2 eljárást meghívjuk

```
call sub2((/ 1.0, 2.0, 3.0 /), z(3:4,5:10), 'szoveg')
```

külső argumentumokkal, akkor a belső argumentumok mérete olyan lesz, mintha

```
   real, intent(in)                 :: a(3)
```

```
   real, intent(out)                :: b(0:1,0:4)
   character (len=6), intent(in)    :: c
```

módon lettek volna deklarálva. Különösen hasznos az átvett hosszúságú karakter változók használáta, ugyanis ez lehetővé teszi a karakter hosszától független eljárások és függvények írását. Például a fenti `duplaz_n` függvény leegyszerűsíthető úgy, hogy a karakterlánc hosszát nem kell átadni:

```
function duplaz_c(c)
  implicit none
  character (*), intent(in) :: c
  character (2*len(c)) :: duplaz_c
  !---------------------------------
  duplaz_c = c//c
end function duplaz_c
```

Ugyanez a függvény valós tömbökre

```
function duplaz_r(a)
  implicit none
  real, intent(in) :: a(:)
  real (2*size(a)) :: duplaz_a
  !---------------------------------
  duplaz_r( 1 : size(a) ) = a     ! az eredmeny tomb elso fele
  duplaz_r( size(a)+1 : ) = a     ! az eredmeny tomb masodik fele
end function duplaz_r
```

Átvett alakú tömböt használó eljárás és függvény, valamint automatikus tömb vagy automatikus karakterlánc értékű függvények esetén kötelező az interfész explicit megadása.

8.2. Általános interfész

Lehetőség van arra, hogy kis mértékben különböző alprogramokat ugyanazon a néven hívjunk meg. Például a fent definiált `duplaz_n`, `duplaz_c`, `duplaz_r` függvények valamennyien megduplázzák az argumentum hosszát. Az általános interfész (generic interface) az `interface GeneralName` utasítással kezdődik, tartalmazza a speciális esetek interfészét, és az `end interface` utasítással végződik. Például:

```
interface duplaz
  function duplaz_n(n,c)
```

```
    implicit none
    integer, intent(in) :: n
    character (len=n), intent(in) :: c
    character (2*n) :: duplaz_n
  end function duplaz_n

  function duplaz_c(c)
    implicit none
    character (*), intent(in) :: c
    character (2*len(c)) :: duplaz_c
  end function duplaz_c

  function duplaz_r(a)
    implicit none
    real, intent(in) :: a(:)
    real (2*size(a)) :: duplaz_a
  end function duplaz_r
end interface
...
write(*,*)duplaz(4,'cica'),duplaz('eger')  ! cicacicaegereger
write(*,*)duplaz( (/1,2,3/) )              ! 1 2 3 1 2 3
```

Az általános interfészben felsorolt alprogramoknak különbözniük kell egymástól az argumentumok számában, vagy típusában, vagy a tömbök dimenziószámában. Így a meghívásból kiderül, hogy melyik konkrét alprogromról van szó.

A Fortran nyelv saját (intrinsic) eljárásai és függvényei is kiterjeszthetők új típusokra, például a gyökvonást a Fortran csak valós és komplex argumentumra értelmezi. Íme egy egyszerű teszt program, ami kiterjeszti az `sqrt` fi2ggvényt egész argumentumra:

```
program tst
  implicit none
  interface sqrt
     real function sqrt_int(i)
       implicit none
       integer :: i
     end function sqrt_int
  end interface
  !eredmeny:1.4142135   1.4142135  (1.4142135,0.0000000E+00)
  write(*,*) sqrt(2),   sqrt(2.0), sqrt((2.0,0.0))
end program tst
real function sqrt_int(i)
  implicit none
  integer :: i
```

```
  sqrt_int = sqrt(real(i))
end function sqrt_int
```

8.3. Műveletek általánosítása: `operator`

Nemcsak a Fortran 90 függvényei és eljárásai, de az alapműveletek is kiterjeszthetőek. Ehhez az interfész egy speciális formáját, az `interface operator(OP)` utasítást kell használni, ahol `OP` vagy egyike a műveleti jeleknek (`+, -, *, /, **, //, ==, /=, <=, >=, <, >`) vagy pedig `.NEV.` alakú, ahol `NEV` egy legfeljebb 31 betűből álló tetszőleges szó, kivéve a `TRUE` és `FALSE` szavakat. Az `.eq.` logikai operátor általánosítása automatikusan vonatkozik az `==` operátorra is, és fordítva.

Az `interface operator(OP)` utasítást egy vagy több függvény definíció követheti, melyeknek két `intent(in)` argumentuma van. Például polinomok összeadásához a `+` műveleti jelet általánosíthatjuk:

```
interface operator(+)
  function add_polynom(p,q)
     implicit none
     type(PolynomType), intent(in) :: p,q
     type(PolynomType) :: add_polynom
  end function add_polynom
end interface
```

Az *unáris*, azaz egy operandusra ható műveletek is álalánosíthatóak, ilyenkor egyszerűen egyváltozós függvényt kell megadni. A Fortranban már definiált műveletek közül csak a `+` és `-` értelmezett egy operandusra, így csak ezek általánosíthatóak. Például a polinom előjelét megfordító unáris `-` és a polinomok kivonását megadó két operandusra ható kivonás interfésze a következő lehet:

```
interface operator(-)
  function sub_polynom(p,q)
     implicit none
     type(PolynomType), intent(in) :: p,q
     type(PolynomType) :: sub_polynom
  end function sub_polynom

  function neg_polynom(p)
     implicit none
     type(PolynomType), intent(in) :: p
     type(PolynomType) :: neg_polynom
  end function neg_polynom
```

```
end interface
```

Természetesen a Fortranban nem szereplő új `.OP.` művelet lehet unáris, például a mátrix invertáláshoz bevezethetünk egy interfészt:

```
interface operator(.inv.)
  function invert_matrix(m)
    use ModMatrix
    implicit none
    real, dimension(n,n)  intent(in) :: m
    real, dimension(n,n)             :: invert_matrix
  end function invert_matrix
end interface
```

8.4. Értékadás általánosítása: `assignment(=)`

A műveleteken kívül az értékadás operátort is általánosíthatjuk egy speciális interfésszel, amely az `interface assignment(=)` utasítással kezdődik. Ezt követi egy vagy több eljárás, melyeknek két argumentuma van. Az első az értékadás jobboldala, és `intent(out)` vagy `intent(inout)` szándékú, míg a második a bal oldal, és `intent(in)` szándékú.

Például polinomoknak adhatunk értéket egy másik polinommal, egy konstanssal, vagy együtthatók tömbjével:

```
  interface assignment(=)
     subroutine set_polynom(p,q)
        implicit none
        type(PolynomType),intent(inout) :: p
        type(PolynomType),intent(in)    :: q
     end subroutine set_polynom

     set_polynom_const(p,c)
        implicit none
        type(PolynomType),intent(inout) :: p
        real,intent(in)                 :: c
     end subroutine set_polynom_const

     set_polynom_coeff(p,a)
        implicit none
        type(PolynomType),intent(inout) :: p
        real,intent(in)                 :: a(0:)
     end set_polynom_coeff
```

```
  end interface
```

Az utolsó `set_polynom_coeff` második argumentuma egy átvett alakú valós tömb, aminek alsó indexét célszerű 0-nak venni, hiszen a polinom együtthatói az $x^0 ... x^n$ hatványokhoz tartoznak.

8.5. Opcionális argumentumok: `optional`

Egy alprogramnak nagyon sok argumentuma lehet. Gyakran ezeknek egy része mindig ugyanolyan értékű, vagy a hívások nagy részében nincsenek befolyással az eredményre. Az ilyen argumentumokat az alprogramon belül célszerű `optional` tulajdonsággal deklarálni. Például:

```
subroutine write_array(a,backwards)
  implicit none
  real,    intent(in)           :: a(:)
  logical, intent(in), optional :: backwards
  ...
end subroutine
```

Ezt az eljárás meghívhatjuk egy vagy két argumentummal:

```
  call write_array(a)
  call write_array(a,.true.)
```

Az eljáráson belül a `present` függvény segítségével dönthető el, hogy egy opcionális argumentum szerepelt-e a külső argumentum listában vagy sem. Például:

```
subroutine write_array(a,backwards)
  implicit none
  real,    intent(in)           :: a(:)
  logical, intent(in), optional :: backwards
  !--------------------------------------
  if(present(backwards))then
     write(*,*)a(size(a):1:-1)
  else
     write(*,*)a
  endif
end subroutine
```

Ez a megoldás nem teljesen jó, hiszen akkor is hátulról kezdve írja ki a tömb elemeket, ha `call write_array(a,.false.)` a meghívás. Mivel az opcionális

változó csak akkor használható, ha jelen van, legcélszerűbb bevezetni egy belső változót, mely vagy az opcionális változótól kap értéket, vagy ha az nincs jelen, akkor az alapértelmezést használjuk:

```
subroutine write_array(a,BackwardsOpt)
  implicit none
  real,    intent(in)           :: a(:)
  logical, intent(in), optional :: BackwardsOpt    ! opcionalis valtozo
  logical:: Backwards                              ! belso valtozo
  !-----------------------------------------------------------------
  backwards=.false.                                ! alapertelmezes
  if(present(BackwardsOpt)) Backwards=BackwardsOpt ! felulir ha lehet
  if(Backwards)then                                ! belso valtozot hasznal
     write(*,*)a(size(a):1:-1)
  else
     write(*,*)a
  endif
end subroutine
```

Ha egy alprogramnak van opcionális argumentuma, akkor az interfész megadása kötelező.

8.6. Név szerinti argumentumok

Ha opcionális változók vannak az argumentumlistában, akkor a külső argumentumok sorrendje nem feltétlenül adja meg, hogy melyik belső argumentumhoz tartoznak. Ilyenkor – vagy, amikor az argumentumok jelentését érthetőbbé kívánjuk tenni – név szerint lehet hivatkozni az argumentumokra a `BELSO=KULSO` formában, ahol `BELSO` a kötelező interfészben definiált belső argumentum név, míg `KULSO` a külső argumentum, ami lehet változó, vagy `intent(in)` szándék esetén tetszőleges kifejezés is. Például a fenti `write_array` meghívható

```
  call write_array(a=x,BackwardsOpt=.true.)
```

módon is. A külső argumentum listában szerepelhetnek pozícionális és név szerint megadott argumentumok is, de csak ilyen sorrendben. A név szerint hivatkozott argumentumok sorrendje azonban – egymás között – tetszőlegesen felcserélhető.

Hiba, ha az argumentumok sorrendjének cseréjével egy általánosított interfészben felsorolt két alprogram argumentumlistája azonossá tehető, mert ilyenkor név szerinti argumentum hívás esetén a két alprogram nem különbötethető meg. Például:

```
interface multiply
  function multiply_v_m(vector,matrix)
    real, intent(in) :: vector(3), matrix(3,3)
    real             :: multiply_v_m(3)
  end subroutine multiply_v_m
  function multiply_m_v(matrix,vector)
    real, intent(in) :: vector(3), matrix(3,3)
    real             :: multiply_m_v(3)
  end subroutine multiply_m_v
end interface
```

fordításnál hibát ad, ugyanis a `multiply(vector=v,matrix=m)` függvény hívásról nem lehetne eldönteni, hogy melyik alprogramra vonatkozik. A belső változónevek megfelelő megváltoztatásával ez a probléma elkerülhető:

```
interface multiply
  function multiply_v_m(vector1,matrix2)
    real, intent(in) :: vector1(3), matrix2(3,3)
    real             :: multiply_v_m(3)
  end subroutine multiply_v_m
  function multiply_m_v(matrix1,vector2)
    real, intent(in) :: vector2(3), matrix1(3,3)
    real             :: multiply_m_v(3)
  end subroutine multiply_m_v
end interface
```

Az argumentumok sorrendjén túl a nevek is különböznek, így a `multiply(v,m)`, `multiply(vector1=v,matrix2=m)` az első `multiply_v_m` függvényt, míg a `multiply(m,v)` és `multiply(vector2=v,matrix1=m)` a második `multiply_m_v` függvényt adja meg.

8.7. Programozási feladat: polinomok összeadása

Írjon egy programot, ami polinomokat tud összeadni. A polinomokat egy összetett típussal reprezentálja, amelyik tartalmazza a polinom fokát és az együtthatókat. Az együtthatók indexe 0-tól (konstans tag) a fok értékéig menjen. A típusdefiníciók, az interfészek és a modul eljárásokat egy `ModPolynom` modulba helyezze el. Az összeadást és az értékadást definiálja `operator(+)` és `assignment(=)` interfészekkel. Ügyeljen arra, hogy egy müvelet vagy értékadás során a polinom foka csökken, ha a legnagyobb indexű együttható 0-vá válik.

Megoldás:

```
module ModPolynom
```

```
  implicit none

  type PolynomType          ! polinom osszetett tipus
     integer :: n           ! polinom foka
     real, pointer :: a(:)  ! egyutthatok (allocatable nem hasznalhato
  end type PolynomType      !              osszetett tipusban)

  interface operator(+)
     module procedure add_polynom
  end interface

  interface assignment(=)
     module procedure set_polynom, set_polynom_array
  end interface
contains
...
end module ModPolynom
```

Két polinom közti értékadás:

```
  subroutine set_polynom(p,q)
    type(PolynomType),intent(inout) :: p   ! ertekadas eredmenye
    type(PolynomType),intent(in)    :: q   ! ertek
    integer :: stat                        ! deallocate statusza
    !-------------------------------------------------------------
    p%n=q%n                              ! polinom foka
    if(associated(p%a))deallocate(p%a)  ! memoria felszabaditasa, ha kell
    allocate(p%a(0:q%n))                ! memoria foglalas
    p%a=q%a                              ! egyutthatok atadasa
  end subroutine set_polynom
```

Az értékadás bal oldalán álló `p` polinomot azért deklaráltuk `intent(inout)` tulajdonsággal, hogy a `deallocate(p%a)` felszabadíthassa az esetlegesen foglalt memóriát.

Definiáljuk a valós tömbbel történő értékadást is! Az eljáráson belül a tömb elemeinek számát a `size` függvénnyel olvashatjuk ki.

```
  subroutine set_polynom_array(p,a)
    type(PolynomType),intent(inout) :: p ! ertekadas eredmenye
    real, intent(in) :: a(0:)            ! atvett alaku tomb
    integer :: n                         ! a polinom foka
    !-------------------------------------------------------------------
    do n=size(a)-1,1,-1                  ! a 0-dik elemet mar nem ellenorizzuk!
```

```
      if(a(n)/=0.0)exit                  ! kilepes: nem nulla egyutthato
    end do
    p%n=n                                ! ertekadas a polinom fokanak
    if(associated(p%a))deallocate(p%a)   ! memoria felszabaditas, ha kell
    allocate(p%a(0:n))                   ! tar foglalas az egyutthatoknak
    p%a = a(0:n)                         ! ertekadas a polinom egyutthatoinak
  end subroutine set_polynom_array
```

Látható, hogy ellenőrizzük, hogy a legmagasabb rendű együttható ne legyen nulla, kivéve a 0-d fokú 0 értékű konstans polinomot.

Ezután a polinomok összedása a következő függvénnyel adható meg:

```
  function add_polynom(p,q)
    type(PolynomType), intent(in) :: p,q ! operandusok
    type(PolynomType) :: add_polynom     ! az osszeg
    real, allocatable :: a(:)            ! foglalhato tomb az egyutthatoknak
    !-------------------------------------------------------------------------
    allocate(a(0:max(p%n,q%n)))          ! memoria foglalas
    a=0.0                                ! kezdoertek 0.0
    a(0:p%n) = a(0:p%n) + p%a            ! a p egyutthatoinak hozzadasa
    a(0:q%n) = a(0:q%n) + q%a            ! a q egyutthatoinak hozzadasa
    nullify(add_polynom%a)               ! az eredmeny polinom inicializalasa
    add_polynom = a                      ! ertekadas polinom=tomb modon
    deallocate(a)                        ! egyutthtok torlese
  end function add_polynom
```

Az `nullify(add_polynom%a)` utasításra azét van szükség, mert az `add_polynom%a` mutató eredetileg definiálatlan értékű, ami program leálláshoz vezető hibát okoz a `deallocate` utasításban (akkor is, ha a `stat` státusz változóval hívjuk meg), illetve az `associated` függvény sem ad definiált értéket (lehet `.true.` is).

A főprogram a következő lehet:

```
program polynom

  use ModPolynom
  implicit none
  type(PolynomType) :: p, q, r
  !--------------------------
  nullify(p%a,q%a,r%a)

  p = (/ 1.0, 2.0/)       ! p = 1 + 2*X
  q = (/-1.0, 2.0, 1.0/)  ! q = -1+ 2*X + X^2
```

```
  r = p + q
  write(*,*)r%a           ! 0.0000000E+00   4.0000000   1.0000000
  p = (/0.0, 0.0, -1.0/)  ! p = -X^2
  q = p + r
  write(*,*)q%a           ! 0.0000000E+00   4.0000000
end program polynom
```

9. A Fortran 90 függvényei

Ebben a fejezetben ismertetjük a Fortran 90 függvényeinek nagy részét. Bizonyos függvények, amelyeknek fizikusok számára nincs túl nagy jelentősége, illetve melyek használata másképp és egyszerűbben is megoldható, kimaradtak.

9.1. Numerikus konverzió

Az alábbi függvények skalár és tömb argumentummal is használhatók. Definiáljunk egy valós tömböt:

```
real, dimension(3) :: rv=(/-2.6,3.5,1.0/)
```

Erre a tömbre alkalmazzuk a függvényeket

`floor(rv) = -3 3 1`	egész rész
`ceiling(rv)= -2 4 1`	felkerekítés egész számra
`int(rv) = -2 3 1`	egész rész, de negatívokra felfelé kerekít
`nint(rv) = -3 4 1`	legközelebbi egész
`aint(rv) = -2.0 3.0 1.0`	ua. mint az tt int csak valós értékű
`anint(rv) = -3.0 4.0 1.0`	ua. mint az tt nint csak valós értékű

Egész számokat is lehet valósra vagy komplexre konvertálni:

`integer, dimension(3) :: iv=(/-2,1,4/)`	3 elemű egész tömb
`real(iv) = -2.0 1.0 4.0`	egészet valósra konvertálva
`cmplx(iv) = (-2.0,0.0) (1.0,0.0) (4.0,0.0)`	egészet vagy valósat komplexre konvertál

Komplex számok konverziója:

`cv=(/(-2,1), (1.1,0), (0,-3)/)`	3 elemű komplex tömb
`real(cv) = -2.0 1.1 0.0`	valós rész
`aimag(cv)= 1.0 0.0 -3.0`	képzetes rész
`conjg(cv)= (-2.0,-1.0) (1.1,-0.0) (0.0,3.0)`	komplex konjugált

9.2. Matematikai függvények

Abszolút érték egész, valós és komplex argumentumokkal:

```
abs(iv)          = 2 1 4
abs(rv)          =   2.5999999   3.5000000   1.0000000
abs(cv)          =   2.2360680   1.1000000   3.0000000
```

Maximum és minimum tetszőleges számú skalár vagy tömb argumentummal. Az argumentumok valósak vagy egészek lehetnek, és típusuknak, illetve tömbök esetén alakjuknak is, egyezniük kell:

```
max(-1., 2., 0.)      =   2.0000000
min(-1., 2., 0.)      =  -1.0000000
max((/1,3/),(/-1,4/))= 1 4
```

Az előjel függvény az első argumentumot ellátja a második argumentum előjelével, azaz `sign(a,b)`=$|a|\mathrm{sgn}b$, ahol "sgn" a matematikai előjel függvény. Ha $b = 0$, akkor pozitív előjelet veszünk. A két argumentum típusának, tömbök esetén alakjának, egyeznie kell, csak valós vagy egész argumentum megengedett:

```
sign(2.0,-1.0)     =  -2.0000000
sign(-2,-8)        = -2
sign(-2,0)         = 2
```

A maradék fűggvény is egész illetve valós skalár vagy tömb változókra értelmezett. Negatív számok esetén a `modulo` függvény adja a matematikai értelemben helyes eredményt:

```
mod(-23,10)        = -3             ! == a-(a/b)*b
mod(-23.0,10.5)    =  -2.0000000    ! == a-aint(a/b)*b
modulo(-23,10)     = 7              ! == a - floor(real(a)/real(b))*b
modulo(-23.0,10.5)=   8.5000000    ! == a - floor(a/b)*b
```

9.3. Trigonometrikus függvények

A trigonometrikus függvények mind radiánban számolnak. A sinus és cosinus függvények komplex argumentumra komplex értéket adnak vissza, a többi függvény csak valós argumentummal hívható meg, és értéke is valós. Skalár és tömb argumentumokkal is használhatók:

```
cos(5.0) = 0.2836622
sin(5.0) = -0.9589243
cos((5.0,1.0)) = (0.4377136,1.1269289)
sin((5.0,1.0)) = (-1.4796976,0.3333602)
tan(5.0) = -3.3805151
acos(-1.0) = 3.1415927                    ∈ [0, π]
asin(-1.0) = -1.5707964                   ∈ [−π/2, π/2]
atan(-2.0) = -1.1071488                   ∈ [−π/2, π/2]
```

Ha x, y Descartes koordinátákat transzformálunk r, φ polár koordinátákra, akkor a φ =`atan(y/x)` nem ad helyes eredményt a negatív félsíkra. Ezen segít az φ =`atan2(y,x)` függvény, amelyik az egész síkra helyes eredményt ad:

```
atan2(-2.0, 1.0)= -1.1071488   ∈ [−π, π]
atan2( 2.0,-1.0)= 2.0344439    ∈ [−π, π]
```

A hiperbolikus függvények csak valós skalár vagy tömb argumentumokkal hívhatók meg:

```
sinh(1.0)  =   1.1752012
cosh(1.0)  =   1.5430807
tanh(1.0)  =   0.7615942
```

9.4. Gyökvonás, exponenciális és logaritmus

Ezek a függvények is használhatók skalár vagy tömb argumentumokkal.

A gyökvonás függvény nem negatív valós vagy tetszőleges komplex argumentummal hívható meg, az eredmény ezzel egyező típusú. Komplex esetben a pozitív valós részű gyököt, ha a valós rész nulla, akkor a pozitív képzetes részű gyököt kapjuk vissza:

```
sqrt(2.0)   =   1.4142135
sqrt((1,1)) = (1.0986841,0.4550899)
sqrt((-1,0))= (0.0      ,1.0)
```

A természetes logaritmus függvény pozitív valós vagy tetszőleges komplex argumentummal. Ha komplex az argumentum, akkor az eredmény képzetes része a $[-\pi, \pi]$ intervallumba esik. A 10-es alapú logaritmus függvény csak pozit1iv valós argumentumra használható. Az exponenciális függvény viszont tetszőleges valós vagy komplex argumentummal hívható meg:

```
exp(1.0)    =   2.7182817
```

```
log(10.0)   =   2.3025851
exp((1,1))  = (1.4686939,2.2873552)
log((1,1))  = (0.3465736,0.7853982)
log10(10.0) =   1.0000000
```

9.5. Karakter manipuláció

A számítógép a karaktereket egész értékű kódokkal ábrázolja. Előfordulhat, hogy olyan karaktert akarunk kiírni, ami a billentyűzetről nem gépelhető be. Ilyenkor a karaktert a kóddal adjuk meg. Az ASCII sztenderd megadja az $0-127$ kódú karaktereket. A legtöbb modern gép az ASCII karakterkészletet használja. A magasabb kódú karakterek azonban már gép függőek. A következő függvények az egész típusú kódok és a karakter típusú értékek közötti konverziót teszik lehetővé. Az argumentumok lehetnek skalárok vagy tömbök:

`char(88) =X`	a 88-dik kódú karakter
`achar(88) =X`	a 88-dik ASCII kódú karakter
`ichar('X') = 88`	az X karakter kódja
`iachar('X')= 88`	az X karakter ASCII kódja

9.6. Karakterlánc manipuláció

Az `adjustl` és `adjustr` függvények a karakterlánc elején illetve végén lévő szóközöktől szabadít meg a karakterlánc balra illetve jobbra való eltolásával. Ettől a karakterlánc hossza nem változik, igy ezek a függvények tömbre is használhatóak:

```
adjustl("  baba  ") =baba    <<<
adjustr("  baba  ") =    baba<<<
```

A `trim` függvény a karakterlánc végén lévő szóközöket levágja és ezzel a karakterláncs hosszát is megváltoztatja, így csak skalár argumentummal használható. Ha a kezdő szóközöktől is meg akarunk szabadulni, akkor az `adjustl` függvénnyel együtt használható:

```
trim("  baba  ")             =  baba<<<
trim(adjustl("  baba  "))    =baba<<<
```

Karakterláncok ismétlése szintén változtatja a hosszat, így csak skalár argumentummal használható:

```
repeat("  baba  ",2)=  baba    baba  <<<
```

A karakterlánc hosszát a `len` függvény adja meg. Mivel a karakterlánc tömbökben minden elem ugyanolyan hosszú, erre is skalár értéket ad vissza, azaz egy elem hosszát:

```
len("  baba  ")     = 8
```

Ha a végző szóközök nélküli hosszra vagyunk kíváncsiak, akkor a `len_trim` függvényt kell használni. Ez tömb karakterlánc argumentum esetén egy hasonló alakú egész tömböt ad vissza az egyes elemek végző szóköz nélküli hosszával:

```
len_trim("  baba  ")= 6
```

9.7. Keresés karakterláncokban

Ezek a függvények karakterlánc skalár vagy tömb argumentummal hívhatók meg, és egész skalár vagy tömb értéket adnak.

```
index("  baba  ","ba")            = 3
index("  baba  ","ba",BACK=.true.)= 5
index("  baba  ","xx")            = 0
index("  baba  ","xx",BACK=.true.)= 0
scan("baba","aeiou")= 2
scan("brrr","aeiou")= 0
verify("baba" ,"abcdef") = 0
verify("baXba","abdcdef")= 3
```

9.8. Vektor és mátrix műveletek

```
dot_product((/1.0,2.0/),(/-1.0,-0.5/))          =  -2.0
dot_product((/(0,1),(0,2)/),(/(0,-1),(0,-0.5)/))= (-2,0)
dot_product((/.true.,.true./),(/.false.,.true./)= T
A=reshape((/1,0,1,0/),(/2,2/))
B=(/3,-1/)
matmul( A, B )= 2 0
matmul( B, A )= 3 3
matmul( A, A )= 1 0 1 0
transpose( A )= 1 1 0 0
```

9.9. Tömb redukciós függvények

```
all( (/.true.,.false./) ) = F
any( (/.true.,.false./) ) = T
count((/.true.,.false./)) = 1
maxval((/2,3,1,3,1/))     = 3
minval((/2,3,1,3,1/))     = 1
maxloc((/2,3,1,3,1/))     = 2
minloc((/2,3,1,3,1/))     = 3
sum((/2,3,1,3,1/))        = 10
product((/2,3,1,3,1/))    = 18
product((/2,3,3/),MASK=(/.true.,.true.,.false./))= 6
C=reshape((/11,21,31,12,22,32/),(/3,2/))
maxval(C,DIM=1)= 31 32
maxval(C,DIM=2)= 12 22 32
```

9.10. Tömbök mérete és alakja

```
real :: x(-2:2,4)
shape(x) = 5 4
lbound(x)= -2 1
ubound(x)= 2 4
size(x)  = 20
lbound(x,dim=1)= -2
ubound(x,dim=2)= 4
size(x,dim=1)  = 5
```

9.11. Dátum és idő

```
call date_and_time(date,time,zone)
date=20020410 time=093317.179 zone=+0100
call system_clock(count,count_rate,count_max)
counter= 27197179  count_rate/s= 1000  count_max= 86399999
```

9.12. Véletlen számok

```
call random_number(x)
x=   0.8949342   0.1597048   0.1592439   0.4130480   9.8006591E-02   0.1967321
 0.4767601   0.9069280   0.7385046   4.7338646E-02   0.6206299   0.9271747   2.5259895E-02
x=   0.8069772   0.8661487   0.3605112   0.1123758   0.6996807   0.5335533   0.4302775   0.
 0.7740043   0.6910374   0.2658404   0.9786111   0.5168514   0.7221339   0.9037358
```

9.13. Egyéb függvények

Kimaradtak

- `DIM(x,y)`=max$(0, x - y)$ függvény, mivel leírható másképp is
- az ASCII karakter sorrenden alapuló string összehasonlító `LGE, LGT, LLE, LLT` függvények, mivel ASCII karakterkészletet használtó gépeken ezekkel ekvivalensek a `>=, >, <=, <` operátorok.
- a logikai típusok között konvertáló `LOGICAL` függvény, mivel többnyire egy logikai típus elegendő
- az egész és valós számábrázolás határaira és pontosságára vonatkozó függvények: `DIGITS, EPSILON, HUGE, MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE, TINY, EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING, SELECTED_INT_KIND, SELECTED_REAL_KIND`, mivel ezek csak nagyon ritkán szükségesek
- a bit manipulációs függvények: `BIT_SIZE, BTEST, IAND, IBCLR, IBITS, IBSET, IEOR, IOR, ISHIFT, ISHIFTC, NOT, MVBITS`, mivel ezek numerikus modellezésben ritkán kerülnek elő
- egyes tömböket transzformáló függvények: `TRANSFER, MERGE, PACK, UNPACK, SPREAD, CSHIFT, EOSHIFT, TRANSPOSE`, mivel ezek gyakran egyszerűen helyettesíthetők ciklusokkal.
- A `RANDOM_SEED` függvény a véletlenszám generálást teszi prediktálhatóvá.

10. Írás és olvasás formázása

```
 Format specifications
 ---------------------
 write(*,*)       : 1
write(*,"(a,i4)"):   1
 character (*), parameter :: form="(a,i4)"
write(*,form)    :   1
write(*,100)     :   1
 100 format(a,i4)
 width=4; write(formvar,*)"(a,i",width,")"
write(*,formvar) :   1

 Edit descriptors
 ----------------
```

```
 Integers
 --------
decimal    3i10   :         1        11        -1
decimal    3i10.4 :      0001      0011     -0001
binary     3b10   :         1      1011**********
binary     3b6.4  :      0001      1011**********
octal      3o10   :         1        13**********
octal      3o10.4 :      0001      0013**********
hexadec    3z10   :         1         B  FFFFFFFF
hexadec    3z10.4 :      0001      000B  FFFFFFFF
genaral    3g10.4 :         1        11        -1

 Reals
 ----
fixed      3f13.4  :       1.0000    1234.5000      -0.0001
exponent   3e13.4  :   0.1000E+01   0.1234E+04  -0.1000E-03
engineer   3en13.4 :   1.0000E+00   1.2345E+03-100.0000E-06
scientific 3es13.4 :   1.0000E+00   1.2345E+03  -1.0000E-04
general    3g13.4  :    1.000        1234.      -0.1000E-03

exponent width 3e13.4e3 :  0.1000E+001  0.1234E+004 -0.1000E-003
position       1p,3e13.4:   1.0000E+00   1.2345E+03  -1.0000E-04
position       2p,3e13.4:   10.000E-01   12.345E+02  -10.000E-05
sign supress   ss,3e13.4:   0.1000E+01   0.1234E+04  -0.1000E-03
sign print     sp,3e13.4:  +0.1000E+01  +0.1234E+04  -0.1000E-03

 Complex = 2 reals
 -----------------
scientific 2es13.4         :   1.0000E+00   1.2345E+03
string     f5.2,"+i*",f8.2 : 1.00+i* 1234.50
general     2g13.4          :    1.000        1234.

 Logical
 -------
logical 2l1  :TF
logical 2l4  :   T   F
logical 2g4.4:   T   F

 Character
 ---------
2a:   shorttoo-long-to-fit
2a10  :     shorttoo-long-t
2a10  :     shorttoo-long-t
2g10.4:     shorttoo-long-t

 Tabulating
```

```
 ----------
tabs         t40,a,t50,a:              abcd      efghij
tab left     tl2,a,tl2,abefghij
tab right    tr2,a,tr2,a:  abcd  efghij
blank insert 2x,a,2x,a  :  abcd  efghij

 New line
 --------
/,a,2/,a:
first_line

third_line

 Colon editing
 -------------
1 item without colon "x(1)=",i2," x(2)=",i2   -->x(1)= 1 x(2)=
1 item with    colon "x(1)=",i2,:," x(2)=",i2 -->x(1)= 1
```

10.1. Programozási feladat: dátum kiírása

Tegye a dátumot és az időt egész illetve a valós változókba és ezeket írja ki!

```
program datum
  implicit none
  character (len=8)  :: date
  character (len=10) :: time
  character (len=5)  :: zone
  integer :: ev,honap,nap,ora,perc
  real    :: masodperc

  call date_and_time(date,time,zone)

  read(date,'(i4,i2,i2)')ev, honap, nap
  write(*,'(a,i4)')'ev=',ev
  write(*,'(a,i2)')'honap=',honap
  write(*,'(a,i2)')'nap=',nap

  read(time,'(i2,i2,f5.3)')ora, perc, masodperc
  write(*,'(a,i2)')'ora=',ora
  write(*,'(a,i2)')'perc=',perc
  write(*,'(a,f5.3)')'masodperc=',masodperc

end program datum
```

10.2. Programozási feladat: polinomok kiírása

Írjon ki egy polinomot minél egyszerűbb formában, pl. (0, 1, x+1, -x, -x^2+3, ...).

11. Névlisták és külső fájlok használata

11.1. névlista: `namelist`

A névlista lehetőséget ad arra, hogy a változókra a programban használt nevükön hivatkozzunk, hogy tömb elemeknek adjunk értéket, valamint, hogy csak annak a változónak adjunk értéket, amelyik eltér az alapértelmezésben vétt értéktől.

A névlistát a `namelist /NEV/ VALTOZOK` utasítással kell megadni. Irásnál és olvasásnál a névlista neve a formátum helyére kerül:

```
program namelisttst
  implicit none
  character (len=10) :: nev='ismeretlen'
  real :: a=1.0, b(10)=0.0
  integer :: n=10
  namelist /parameters/ a, b, n, nev
  read(*,parameters)
  write(*,parameters)
end program namelisttst
```

A program számára a névlistát a `&NEV valtozo=ertek /` formában kell megadni, például:

```
  &parameters
    nev='Kiss'
    n=2
    b(3:4)=2*3.1
    b(7)  =-1.
  /
```

Mint látható az értékadások sorrendje tetszőleges. Az `a` változónak például nem is adunk értéket, így megőrzi a kezdő értékét. A `b` tömb két elemének adunk értéket, itt használható `DARAB*ERTEK` formátum több egyforma érték megadására. A karakterláncok értékét idézőjelek közé kell zárni. Maguk az értékek csak konstansokat tartalmazhatnak.

Kiíráskor az összes érték megjelenik. A változó nevek csupa nagy betűvel. A konkrét formátum függ a fordító programtól is. A fenti példaprogram eredménye a fenti névlista beolvasása esetén például:

```
pgf90 -o namelist.exe namelist.f90
namelist.exe
  &parameters
    nev='Kiss'
    n=2
    b(3:4)=2*3.1
    b(7)  =-1.
  /
 &PARAMETERS
 A =    1.000000    ,
 B =    0.000000    ,
        0.000000    ,
        3.100000    ,
        3.100000    ,
        0.000000    ,
        0.000000    ,
       -1.000000    ,
        0.000000    ,
        0.000000    ,
        0.000000    ,
 N =            2,
 NEV = Kiss
 /
```

11.2. Fájl megnyitása: `open`

Külső fájlokat (azaz se nem karakterlánc, se nem a sztenderd input vagy output) írás vagy olvasás előtt meg kell nyitni, illetve az írás és olvasás végeztével érdemes bezárni (bár a program normális futásának végén ez mindenképp bekövetkezik). A megnyitást az `open(UNIT, ARGUMENTLIST)` utasítással végezzük el, ahol a `UNIT` egy pozitív egész szám értékű konstans, változó, vagy kifejezés, általában a 10...99 tartományban. Az `ARGUMENTLIST` számos opcionális argumentumot tartalmazhat vesszővel elválasztva. Ezek fontossági sorrendben

- `FILE=fl` ahol `fl` egy karakterlánc, ami a megnyitandó fájl nevét tartalmazza. Ha a FILE argumentum hiányzik, akkor a fájl neve 'fort.UNIT' lesz, ahol UNIT egy pozitív egész.
- `STATUS=st` ahol `st` lehet 'OLD', 'NEW', 'REPLACE', 'SCRATCH', vagy 'UNKNOWN'. Ha 'OLD', akkor a fájlnak már léteznie kell, ha 'NEW' ak-

kor viszont nem. Ha 'REPLACE' akkor a fájl – ha már létezett – törlődik. Ha 'SCRATCH' akkor a fájl nevét nem szabad megadni, és automatikusan törlődik a program végen, vagy bezárás után. Az 'UNKNOWN' értelmezése fordító függő, és sajátos módon ez az alapértelmezés.

- `ACCESS=acc` ahol `acc` 'SEQUENTIAL', azaz soros, vagy 'DIRECT', azaz közvetlen elérés. Az alapértelmezés soros elérés.
- `FORM=fm` ahol `fm` lehet 'FORMATTED' vagy 'UNFORMATTED'. Az alapértelmezés 'FORMATTED' soros elérésnél és 'UNFORMATTED' közvetlen elérésnél.
- `RECL=rl` ahol `rl` egy egész értékű kifejezés ami megadja a rekordok hosszát. Közvetlen elérésű faj1lra ezt kötelező megadni. Soror elérésű fájlra a maximális méretű rekordot adja meg.
- `POSITION=pos` ahol `pos` lehet 'ASIS' (ahogy van), 'REWIND' (elejéről), vagy 'APPEND' (végére). Az 'ASIS' csak akkor működik determinisztikusan, ha a faj1l már meg van nyitva.
- `ACTION=act` ahol `act` lehet 'READ', 'WRITE', vagy 'READWRITE' Az alapértelmezésre nincs sztenderd, de általában 'READWRITE'.
- `ERR=label` ahol `label` egy numerikus cimke, ahova a vezérlés kerül hiba esetén.
- `IOSTAT=ios` ahol `ios` egy egész változó, ami 0 értéket kap, ha nincs hiba, és pozitívat ha van.
- `BLANK=bl` ahol `bl` lehet 'NULL' vagy 'ZERO'. Formattált olvasásnál számít. Ha 'ZERO' akkor a szóközöket 0-nak tekinti, ha 'NUL', akkor a szóközöket nem veszi figyelembe. Az alapértelmezés a 'NUL'.
- `DELIM=dl` ahol `dl` lehet 'APOSTROPHE' (normál idézőjel), 'QUOTE' (szimpla idézőjel), vagy 'NONE', ami megadja, hogy karakterstringek hogyan legyenek határolva *-gal formattált illetve névlistás kiírásnál. Az alapértelmezés 'NONE'.
- `PAD=pd` ahol `pd` lehet 'YES' vagy 'NO', ami eldönti, hogy formattált beolvasásnál a rekord vége szóközökkel egészüljön-e ki, ha a rekord rövidebb mint a beolvasandó adat. Az alapértelmezés 'YES'. Ha 'NO', akkor hibaüzenetet kapunk amikor a rekord rövidebb.

11.3. Fájl bezárása: `close`

A bezárást a `close(UNIT, ARGUMENTLIST)` utasítással végezzük el, ahol UNIT egy pozitív egész értékű kifejezés, míg az ARGUMENTLIST a következő opcionális argumentumokból áll:

- `IOSTAT=ios` Ugyanaz mint az `open` utasításnál.
- `ERR=label` Ugyanaz mint az `open` utasításnál.
- `STATUS=st` ahol `st` lehet 'KEEP' (megtart) vagy 'DELETE' (töröl), ami eldönti, hogy a bezárás után a fájl megmarad, vagy megszűnik. Az alapértelmezés 'KEEP' kivéve ha a megnyitás `STATUS='SCRATCH'` paraméterrel történt, amikor a 'KEEP' nem is adható meg a bezárásnál.

11.4. Fájl állapotának vizsgálata: `inquire`

Az `inquire` utasítással szinte minden megtudható egy fájlról, illetve a megnyitás paramétereiről. Háromféleképpen is meg lehet hívni, ebből az első kettő `inquire(UNIT, ARGUMENTLIST)` illetve `inquire(FILE=filename, ARGUMENTLIST)`. Az opcionális argumentumok közül nyilvánvaló az IOSTAT, ACCESS, SEQUENTIAL, FORM, RECL, BLANK, POSITION, ACTION, DELIM és PAD változók jelentése, amik visszaadják az `open` utasításban használt argumentumok értékét. További opcionális argumentumok:

- `EXIST=ex` ahol `ex` egy logikai változó, ami `.true.` értéket kap, ha a fájl létezik.
- `OPENED=op` ahol `op` egy logikai változó, ami `.true.` értéket kap, ha a fájl meg van nyitva.
- `NAME=name` ahol `name` karakterlánc változó, ami visszaadja a fájl nevét.
- `NUMBER=num` ahol `num` egy egész szám, ami a UNIT értéke ami a fájlhoz tartozik, és -1, ha a fájl nincs nyitva.
- `NEXTREC=nr` ahol `nr` egy egész szám, ami a következő rekord indexe.

Az `inquire` utasítás harmadik formája az

```
inquire(IOLENGTH=len) IOLIST
```

arra használható, hogy megállapítsuk, hogy milyen hosszú lenne az a rekord, amibe az IOLIST-ben szereplő értékeket kiírnánk. A hosszat általában byte-ban kapjuk vissza, bár ez elvileg fordító függő. Ilyen módon megállapítható, hogy például a valós számok 4 vagy 8 byte-ban vannak-e ábrázolva.

11.5. Bináris írás és olvasás: FORM='UNFORMATTED'

Külső fájlokba a legegyszerűbben, leggyorsabban, és a pontosság teljes megőrzésével írhatunk, illetve onnan olvashatunk adatokat. Ehhez a fájl-t

```
open(unit,FILE=nev,FORM='UNFORMATTED')
```

módon kell megnyitni, és írásnál illetve olvasásnál a formátumot egyszerűen el kell hagyni:

```
write(unit)x(1:10),y
read(unit)i,n
```

A fájl továbbra is rekordokból áll, de ezek végét nem a sorvége karakter jelzi, hanem a rekord hosszát egy általában négy byte-os egész szám adja meg a rekord elején és végén is.

11.6. Soros és közvetlen elérésű fájlok

Soros elérésű fájlokat általában elejétől végéig olvasunk vagy írunk. Ettől el lehet térni a következő utasításokkal:

`BACKSPACE unit`	visszaugrás 1 rekorddal
`REWIND unit`	vissza a fájl elejére
`ENDFILE unit`	ugrás a fájl végére

Közvetlen elérésű fájlokat az

```
open(unit,FILE=filename,ACCESS='DIRECT',RECL=100)
```

utasítással lehet megnyitni, és ezután az írásnál és olvasásnál meg lehet adni a rekord indexét:

```
write(unit,REC=1)'valami'
write(unit,REC=4)'mas'
read(unit,REC=6) name
```

Ha a közvetlen elérésű fájlt `FORM='FORMATTED'` argumentummal nyitjuk meg, akkor a formátumot expliciten meg kell adni, azaz a `*` formátum nem használható.

11.7. Programozási feladat: diffúzió 2 dimenzióban

Oldja meg az FTCS módszerrel a

$$\frac{\partial \rho}{\partial t} = \alpha_x \frac{\partial^2 \rho}{\partial x^2} + \alpha_y \frac{\partial^2 \rho}{\partial y^2} \tag{1}$$

diffúziós egyenletet 2 dimenzióban. Használjon névlistát a paraméterek beolvasására. A rács mérete is legyen paraméter. Az eredményt írja ki egy fájlba paraméterrel megadott lépésszámonként. Érdemes az egyes időponthoz tartozó eredményeket külön-külon2n fájlba írni. A fájl neve tartalmazza a lépésszámot. Vigyázat! A fájl neve nem tartalmazhat szóközt, azaz a lépésszám kiírásánál olyan formátumot kell választani, ami 0-t ír a szám elé.

A számítás eredményét az ingyenes GNUPLOT program segítségével fogjuk vizualizálni. Két dimenziós rács esetén a GNUPLOT az adatokat a következő formátumban várja: az egyes változókat szóközzel elválasztva egy sorba kell írni. A rács egy sorához tartozó pontok adatait egymást követő sorokba írjuk. Az egyes rács sorok közé üres sort írunk. Tehát ha egy $M \times N$ méretű rácson K változónk van, akkor a fájl a következő formátumú lesz:

```
Var1_1,1 Var2_1,1 ... VarK_1,1
Var1_1,2 Var2_1,2 ... VarK_1,2
...
Var1_1,N Var2_1,N ... VarK_1,N

Var1_2,1 Var2_2,1 ... VarK_2,1
Var1_2,2 Var2_2,2 ... VarK_2,2
...
Var1_2,N Var2_2,N ... VarK_2,N

.
.
.

Var1_M,1 Var2_M,1 ... VarK_M,1
Var1_M,2 Var2_M,2 ... VarK_M,2
...
Var1_M,N Var2_M,N ... VarK_M,N
```

A vizualizációhoz indítsuk el a GNUPLOT-t:

```
gnuplot
```

majd írjuk be a következő parancsokat

```
set hidden3d
set data style lines
```

Az első utasít1s azt jelenti, hogy felületeknél a takarásban levő része a felületnek ne látsszon, a második pedig azt, hogy az adatpontokat vonalakkal kössük össze. Ezután egy adott fájban levő adatokat az

```
splot 'file.nev' using NCOL
```

utasítással ábrázolhatjuk. A `using NCOL` választja ki, hogy melyik oszlopot (column) azaz melyik változót akarjuk ábrázolni. Alapértelmezésben az első oszlopot ábrázoljuk. Ha csak egy változó van a fájlban, akkor nincs szükség a `using` paraméterre. Lehet egyszerre több fájlt illetve több változót is ábrázolni, de erre itt nem térünk ki. A GNUPLOT jól használható `help` utasítással rendelkezik, amin keresztül minden információ elérhető.

12. Egész és valós számok pontossága

12.1. Egész és valós számábrázolások

Az egész és valós számok többféle módon is ábrázolhatók. Például egy egész szám ábrázolható 1, 2, 4 vagy 8 byte-tal is, míg egy valós szám 4, 8, vagy ritkán 16 byte-tal. Egészek esetében a különbség az, hogy mekkora számok írhatók le. Például az egy byte-os egész tipikusan -127 és 127 közötti értékeket vehet fel, míg a két byte-os -32767-tól 32767-ig.

A valós számok esetében az értékes jegyek száma és a maximális exponens függ a szám ábrázolásától. Általában a 4 és 8 byte-os számábrázolások állnak rendelkezésünkre, ezeket hagyományosan szimpla és dupla pontosságú valós számoknak nevezzük. Egyes gépeken (pl. Cray) azonban a 8 és 16 byte-os valós típusokat használnak. A 4 byte-s valós számokkal végzett műveletek eredménye általában 6, a 8 byte-ossal legalább 12 tizedes jegyre pontos. A 4 byte-s számokkal 10^{-38} és 10^{+38} közti exponensű számokat lehet ábrázolni, a míg a 8 byte-sak esetében 10^{-300} és 10^{+300} között.

12.2. Valós típus választása fordításkor

Szinte az összes Fortran fordító program lehetővé teszi, hogy az egyszerű `real :: ...` utasítással deklarált változókat az alapértelmezés szerinti 4 helyett 8

byte-n ábrázoljuk. Például a Portland Group `pgf90` fordítójánál a `-r8` kapcsolóval lehet ezt megtenni. Ez az egyik legegyszerűbb és legkényelmesebb módja, hogy ugyanazt a programot szimpla és dupla pontossággal is tudjuk használni. Természetesen ez a kapcsoló nem változtatja meg azokat a deklarációkat, melyek expliciten megadják a változó pontosságát.

12.3. Valós típusok Fortran 77-ben: `double precision, real*4, real*8`

A hagyományos Fortran 77-ben a dupla pontosságú valós számokat `real` helyett

```
double precision ...
```

módon deklarálták. Konstansok esetén pedig az `E` exponens helyett a `D`-t használták, pl. `1.0D0` egy dupla pontosságú egyes. A dupla pontosság általában a 8 bájtos valós számokat jelenti, de nem minden gépen. Emiatt használatos volt a

```
real*4 ...
real*8 ...
```

deklaráció is, ami fordítótól függetlenül 4 illetve 8 byte-s valós számot jelent. Konstansok esetében azonban már nincs olyan jelölés, ami garantáltan 4 vagy 8 byte-t jelentene.

12.4. Valós típusok Fortran 90-ben: `kind`

Bár a Fortran 90-ben is használhatóak, és használatosak is az előző részben ismertetett megoldások, ezeket inkább csak a kompatibilitás kedvéért tartották meg a Fortran 90 tervezői. Fortran 90-ben a valós és egész típusok közül a `kind` tulajdonság megadásával választhatunk:

```
integer (kind=2)  :: i
integer (kind=i1) :: j
integer (i1)      :: k
real (kind=4)     :: r
real (r8)         :: q
```

Mint látható, a típust közvetlenül az `integer` illetve tt real utasítások mögé kell írni zárójelben. A `kind=` kiírása nem kötelező. A típus maga egy egész

szám, mely a legtöbb Fortran fordító esetében a byte-k számát jelenti. Más fordítóknál viszont a típusokat egyszerűen az 1,2,3... számokkal jelölik. Ha olyan programot akarunk írni, ami különböző fordítókkal egyaránt jól működik, akkor a típust egy konstans paraméterrel kell megadni, például:

```
integer, parameter :: i1=selected_int_kind(2)
integer, parameter :: r8=selected_real_kind(12,100)
```

A `selected_int_kind` függvénynek egyetlen paramétere azt mondja meg, hogy az adott típusú egész számok közül a legnagyobb 10 hanyadik hatványánál nagyobb. Például `selected_int_kind(2)` azt az egész típust választja, amelyik már lefedi a -100...100 intervallumot. Ez általában a 1 byte-s egész típus lesz, ami -127..127 közötti értékeket vehet fel, de előfordulhat, hogy a fordító nem támogatja az 1 byte-s egész típust, és ezért a 2 vagy 4 bájtos típust választja a függvény. A `selected_real_kind` függvény első argumentuma azt írja elő, hogy a választandó valós típusban legalább mennyi legyen az értékes tizedesek száma, míg a második paraméter azt adja meg, hogy 10 hanyadik hatványáig kell kiterjednie a lefedett intervallumnak. A példában szereplő `selected_real_kind(12,100)` általában a 8 bájtos valós számot fogja választani.

Konstansok esetén az aláhúzás karakterrel lehet kiválasztani a megfelelő típust, például:

```
i = 354_4
j = 1_i1
r = 1.0_r8
```

Ezekben az értékadásokban az `i` változó a 354-nek a 4 byte-s egész, a j az 1-nek az 1 byte-s egész, míg az r az 1-nek a 8 byte-s valós ábrázolását kapta.

12.5. Típusok lekérdezése

Számos függvény áll rendelkezésre, mellyel információkat kaphatunk egy egész vagy valós szám típusáról. A `kind(...)` függvény például visszaadja egy szám típusát. Példaúl

```
write(*,*)'kind(1)  =',kind(1)
write(*,*)'kind(1.0)=',kind(1.0)
```

kiírja az egész és valós számok alapértelmezés szerinti típusát. Természetesen az argumentum lehet rögzített típusú konstans vagy változó is.

A `huge()` függvény megadja, hogy egy adott típusú változónak mi a legnagyobb lehetséges értéke, pl. `huge(1)` értéke 2147483647 míg `huge(1.0)` értéke `3.4028235E+38` ha az egészeket és a valósakat 4 byte-n ábrázoljuk. A `tiny()` függvénynek csak valós argumenuma lehet, és az adott típusú valós számokkal ábrázolható legkisebb pozitív számot adja vissza, pl. `tiny(1.0)` értéke lehet `1.1754944E-38`.

Ha konkrétan a bájtok számara vagyunk kíváncsiak, akkor azt az `inquire()` függvénnyel olvashatjuk ki, mint az a 11.4 részben olvasható.

12.6. Bináris fájlok

Fontos megjegyezni, hogy a valós és egész számok különböző ábrázolása nemcsak a memóriára, hanem a valós illetve egész számokat tartalmazó bináris fájlokra is vonatkozik. Egy szimpla pontossággal bináris fájlba kiírt valós számot csak szimpla pontosságú valósként lehet beolvasni.

13. Párhuzamos programozás High Performance Fortranban

13.1. Párhuzamos programozási nyelvek

A High Performance Fortran a Fortran (HPF) egy kiterjesztése, mely a vektor és parallel szuperszámítógépeken való hatékony futtatást teszi lehetővé. Ma a HPF leginkább a szétosztott (distributed) memóriájú – ilyen például a Virgo klaszter, amelyben minden gép csak a saját memóriájához fér hozzá hatékonyan – párhuzamosításra használatos, ugyanis a vektor számítógépeken az F90 fordítókhoz képest a HPF nem tud többet, míg a közös (shared) memóriájú párhuzamosításra egyre jobban terjed az OpenMP kiterjesztés (ez utóbbi a C illetve Fortran programokba írt direktívák segítségével thread típusú párhuzamosítást tesz lehetővé).

A HPF legnagyobb vetélytársa a Message Passing Interface (MPI) könyvtár, mely a C és Fortran nyelvekhez készült. Míg a HPF egy magas szintű, úgynevezett adatparallel nyelv, addig az MPI egy alacsonyszintű, végrehajtás parellel nyelv, melyben az egyes processzorok explicit üzenetekkel cserélnek információkat. A legtöbb HPF fordító az MPI-t használja a kommunikációhoz, de a szükséges üzenetváltásokat fordító (és nem a programozó) írja bele a programba. Míg az MPI könyvtár segítségével tetszőleges párhuzamos algoritmus megvalósítható, addig a HPF csak viszonylag egyszerű adatstruktúrák párhuzamosítását teszi lehetővé, viszont ezt nagyon egyszerűen, és általában nagyon hatékonyan.

13.2. Adatparallel megközelítés

Az adatparallel megközelítésben a változókat, tipikusan nagy méretű tömböket osztunk szét a processzorok között. Minden processzor ugyanazokat az utasításokat hajtja végre, de a tömbnek csak azzal a részével foglalkozik, ami hozzá tartozik. A többi processzortól csak akkor kér adatokat, ha szükséges. A HPF-ben a kiírást és a beolvasást egyetlen processzor végzi (általában a 0-s sorszámú). Nagy mennyiségű adat írása vagy olvasása így egy kisebb számításigényű problémában szűk keresztmetszetet jelenthet. Ez egy hátránya a HPF jelenlegi változatának az MPI könyvtárral szemben, amelyben a párhuzamos input output különösebb nehézség nélkül megoldható. A HPF 2.0 változatában már tervezték a párhuzamos input/output lehetővé tételét, de nem biztos, hogy ezt a fordítók meg is fogják valósítani.

13.3. A HPF nyelv alapjai

A HPF a Fortran 90-et direktívákkal, a FORALL utasítással, és az eljárások elé írható PURE tulajdonsággal és néhány függvénnyel egészíti ki. A FORALL utasítás a Fortran 95-nek már része.

A direktívákból csak a leglényegesebbekkel ismerkedünk meg. Egy tömböt a deklaráció után írt `!HPF$ DISTRIBUTE` direktívával lehet a processzorok között elosztani. Például

```
real :: a(100,100), b(90)
!HPF$ DISTRIBUTE a(BLOCK,*)
!HPF$ DISTRIBUTE b(CYCLIC)
```

Az `a` tömb első indexét blokkonként osztottuk el (ezt jelöli a BLOCK), míg a második index replikálva van, azaz minden processzoron az összes index elérhető (ezt jelöli a *). Ha például 10 processzor van, akkor `a(1:10,1:100)` az első, `a(11:20,:)` a második, ..., `a(91:100,:)` a tizedik processzoron lesz. A `b` változót ciklikusan (CYCLIC) osztjuk szét a processzorok között, azaz, `b(1)` az első, `b(2)` a második, ..., `b(10)` a tizedik, `b(11)` pedig ismét az első, `b(12)` a második, ... processzorra kerül.

A BLOCK és a CYCLIC utasítások is kiegészíthetőek egy paraméterrel, ami a blokk méretét adja meg, pl. BLOCK(5) azt jelenty, hogy minden blokk 5 elemből áll, és persze ha kifogytunk a processzorokból, akkor kezdjük elölről. A CYCLIC(5) ugyanezt eredményezi. A BLOCK(1) tulajdonképpen ugyanaz, mint a CYCLIC.

Ha több változót ugyanúgy akarunk szétosztani a processzorok között, akkor használhatjuk a DISTRIBUTE direktíva egy másik formáját:

```
real :: a(100,100), b(100,100)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) :: a, b
```

Az `!HPF$ ALIGN direktíva` segítségével a különböző index határokkal rendelkező tömbök processzorok közti szétosztását tudjuk összehangolni. Például ha egy egy 98*98-as rácsot szellemcellákkal veszünk körül, és az `a` tömb csak a válódi, míg a `b` tömb a szellem cellákat is tartalmazza, akkor érdemes a következő HPF direktívákkal deklarálni ezeket:

```
real :: a(98,98), b(0:99,0:99)
!HPF$ DISTRIBUTE b(BLOCK,BLOCK)
!HPF$ ALIGN a(I,J) WITH B(I,J)
```

Ha a tömböket 10*10 processzoron osztjuk szét, akkor `b(0:9,0:9)` és `a(1:9,1:9)` lesz az első processzoron, `b(10:19,0:9)` és `a(10:19,1:9)` a másodikon, ... `b(10:19,10:19)` és `a(10:19,10:19)` a tizenegyediken, ... és `b(90:99,90:99)` és `a(90:98,90:98)` a századikon.

Az `!HPF$ PROCESSORS` utasítással azt adhatjuk meg, hogy több dimenziós tömbök blokkos szétosztásánál a processzorok hányszor hány blokkra osszák szét a tömböt. Például a 100*100-as tömb 20 processzor között felosztható 5*4 blokkra

```
!HPF$ PROCESSORS PP(5,4)
real :: a(100,100)
!HPF$ DISTRIBUTE a(BLOCK,BLOCK) ONTO PP
```

Így az első processzor az `a(1:20,1:25)` altömböt tartalmazza.

A processzorok számát a `NUMBER_OF_PROCESSORS()` függvénnyel kaphatjuk meg. Ezt például használhatjuk az `!HPF$ PROCESSORS` direktívában és a Fortran programban is, például

```
!HPF$ PROCESSORS PP(5,NUMBER_OF_PROCESSORS()/5)
if(mod(NUMBER_OF_PROCESSORS(),5) /= 0) then
   write(*,*) &
   'Number of processors should be a multiple of 5:', &
   NUMBER_OF_PROCESSORS()
   stop
endif
```

Ha egy tömböt nem látunk el semmilyen HPF direktívával, akkor az minden processzoron teljes egészében megtalálható lesz, ami ugyanaz, mintha a direktívában (*,*) replikálást adtunk volna meg.

A végrehajtás tömb szintaxis esetén automatikusan párhuzamos, pl.

```
real :: a(100,100), b(100,100)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: a,b
a=b**2
a(2:99)=a(1:98)-a(3:100)
```

Az első `a=b**2` végrehajtható utasítás semmilyen kommunikációt nem igényel, hiszen `a(i,j)` tömb elem és a `b(i,j)` tömb elem mindig ugyanazon a processzszoron vannak. A második utasítás már igényel kommunikációt, de ezt a fordító intézi el.

Ha a végrehajtandó utasítás függ a tömb elemek értékétől, akkor a WHERE konstrukciót érdemes használni, mert ez automatikusan párhuzamusan fordítódik, például

```
where(b>0.0)
  a = alog(b)
elsewhere
  a = 0.0
endwhere
```

Ha a végrehajtandó utasítás függ az indexektől, akkor a FORALL ciklus utasításra van szükség, például

```
forall(i=1:100, j=1:100) a(i,j)=100*i+j + b(i-1,j-1)
```

Az `!HPF$ INDEPENDENT` direktívával az egyszerű DO ciklus is párhuzamosítható, ha nem igényel kommunikációt:

```
!HPF$ INDEPENDENT, NEW(s)
do i=1,100
  s = sqrt(a(i))
  b(i) = a(i) + s - 2*s/(s+1)
end do
```

ahol a `NEW(s)` azt jelenti, hogy az `s` skalár változót nem kell a processzorok között replikálni, hanem az minden processzoron más-más értéket vehet fel.

Ha egy ciklusban meghívunk egy eljárást, akkor az általában nem párhuzamosítható. Ha az eljárást a PURE tulajdonsággal definiáljuk, akkor garantáljuk, hogy csak lokális adatokat használ fel, így a ciklus párhuzamosítható:

```
!HPF$ INDEPENDENT
do i=1,100
  a(i) = sq(b(i))
end do

contains
  pure real function sq(x)
    real :: x
    if(x >= 0) then
       sq  = sqrt(x)
    else
       sq = -1.
    endif
  end function sq
```

13.4. HPF program fordítása és futtatása

A Portland Group fordítók közül a `pghpf` fordítja le a HPF direktívákat és utasításokat:

```
pghpf -o tst.exe tst.f90
```

A futtatásnál az `-pghpf` kapcsoló után írhatóak különféle utasítások, például

```
./tst.exe -pghpf -np 4 -host virgo2:2,virgo3,virgo4 -stat
```

A `./` az elején azért kell, mert abban a shellben, amiben a párhuzamosan futó processzek elindulnak, a helyi könyvtár nem szerepel a `path`-ban. A `-np` kapcsoló adja meg a párhuzamosan futó processzek számát, a `-host` felsorolja azokat a gépeket, amin futnia kell a programnak. A `virgö:2` azt jelenti, hogy 2 processzoros a gép. Így a virgö-n 2, a virgő-n és virgo4-n egy-egy processz indul el. Természetesen több processz is futhat egy gépen, mint ahány processzor van rajta, de ez általában nagyon lerontja a hatékonyságot. A -stat kapcsoló a futás után különféle statisztikákat ír ki.

13.5. Programozási feladat: a diffúziós program HPF-ben

Irja át a kétdimenziós diffúziós programot HPF-be! Mérje meg, hogy milyen gyorsan fut a program 1, 2, 4, és 8 processzoron! Vizsgálja meg a skálázást I/O-val és anélkül!

14. Párhuzamos programozás Message Passing Interface könyvtárral

Ingyenes könyvtár, a legnépszerűbb az mpich implementáció. Vannak debuggolást, parallel IO-t stb. segítő verziók is.

Az MPI könyvtárat C-ben írták, de van hozzá C és Fortran 77 interfész is, azaz ebből a két nyelvből használható. A különbség a két interfész között csupán annyi, hogy a C függvények egy hibakóddal térnek vissza, míg a Fortran eljárásoknak van egy extra hiba kód paraméterük (kivéve az időt mérő `MPI_WTIME()` és `MPI_WTICK()` függvényeket).

Minden eljárás, függvény és konstans az `MPI_` karakterekkel kezdődik. Ha egy eljárásban használni akarjuk az MPI könyvtárat, akkor az MPI header (fejléc) fájlt az `include` utasítással be kell olvasni:

```
#include "mpi.h"          -- C header file

include 'mpif.h'          -- Fortran header file
```

A header file konstansokat és függvények deklarációját tartalmazza. Az `include` utasítást az `implicit none` után a deklarációs részben kell elhelyezni.

Egy MPI program elején mindig meg kell hívni az

```
MPI_Init(iError)
```

eljárást. Ezt követően általában szükség van a processzorok számának (nProc), illetve az adott processzor indexének (iProc) megállápítására (ez utóbbi 0-tól (nProc−1)-ig mehet):

```
MPI_COMM_SIZE(iComm, nProc, iError)
MPI_COMM_RANK(iComm, iProc, iError)
```

Az `iComm` paraméter egy kommunikátort definiál, ami a processzorok egy csoportját jelöli ki. A kommunikátor tulajdonképpen egy egész típusú változó illetve konstans, melyet kizárólag az MPI könyvtár eljárásain keresztül használunk. Egyszerű MPI programok esetén az összes processzor résztvesz a kommunikációban, amihez az MPI header fájlban deklarált `MPI_COMM_WORLD` kommunikátor tartozik.

Az MPI programot az

```
MPI_Finalize(iError)
```

eljárás meghívásával kell befejezni, amit minden processzor végrehajt. Ha a végrehajtás során valamilyen hiba fordul elő, akkor az

```
MPI_Abort(iComm,iError)
```

eljárás meghívásával lehet leállítani a program futását. Ezt elég ha egy processzor hívja meg (ahol a hiba előfordult).

A processzorok közti kommunikáció legelemibb formája az úgynevezett pont-pont kommunikáció két processzor között. Ha több üzenetet is cserélnek egyszerre, akkor egy egész számot tartalmazó címke (iTag) segítségével tudják az üzeneteket egymástól megkülönböztetni. Egyszerre egy egész (nVar méretű) tömb küldhető melynek típusát (Type) az MPI header fájlban deklarált

```
MPI_INTEGER, MPI_REAL, MP_DOUBLE_PRECISION
MPI_COMPLEX, MPI_LOGICAL, MPI_CHARACTER
```

konstansok közül lehet kiválasztani. A pont-pont kommunikációhoz az egyik processzoron egy küldő (send), a másikon egy fogadó (receive) eljárást kell meghívni:

```
MPI_Send(Var,nVar,Type,iProcTo,iTag,iComm,iError)
```

```
MPI_Recv(Var,nVar,Type,iProcFrom,iTag,iComm,Status,iError)
```

A fogadó eljárásban szereplő `Status` változót

```
integer :: Status(MPI_STATUS_SIZE)
```

formában kell definiálni. Ez a változó különböző információkat tartalmaz a kapott üzenetről, de ennek kiolvasására egyszerűbb esetekben nincsen szükség.

Bár pont-pont kommunikációval elvileg minden feladat megoldható, az MPI könyvtár sok olyan eljárást tartalmaz, ami a gyakran előforduló kommunikációs feladatok végrehajtását egyszerűbbé és hatékonyabbá teszi. Ilyen feladat, amikor egy processor (Root) küld szét (broadcast) valamilyen információt az összes többinek:

```
MPI_Bcast(Var,nVar,Type,Root,iComm,iError)
```

Egy másik tipikus feladat a processzorok szinkronizációja, azaz a végrehajtás csak akkor mehet tovább, amikor az összes processzor eljut egy adott ponthoz:

```
MPI_Barrier(iComm)
```

Gyakran van szükség valamilyen több processzoron megtalálható numerikus vagy logikai változó összegére, maximumára, és kapcsolatára, stb. Ezt a műveletet redukciónak nevezzük, hiszen sok információból egyre redukálunk. Az eredményt vagy egyetlen processzor (Root) kapja meg, vagy az összesnek (All) is elküldhetjük:

```
MPI_Reduce(SendVar, RecvVar, nVar, Type, Operator, Root, iComm, iError)

MPI_AllReduce(SendVar, RecvVar, nVar, Type, Operator, iComm, iError)
```

Itt az operátor az következő műveletek egyike:

```
MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX, MPI_MINLOC, MPI_MAXLOC   -- sza1mokra
MPI_LOR, MPI_LAND, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR      -- logikaira
```

Végezetül megemlítjük azt a két függvényt, amivel a CPU időt lehet mérni. A fontosabb az `MPI_Wtime()` függvény (nincs paramétere), ami egy tetszőleges kezdőpont óta eltelt időt adja meg másodpercben 8 bájtos valós számként. Az idő mérés pontosságát az `MPI_Wtick()` függvény adja meg szintén másodpercben.

Ebből a rövid összefoglalóból szükségszerűen rengeteg minden kimaradt. A teljesség igénye nélkül a fontosabbak: nem blokkoló kommunikáció, típusok definiálása, csoportok és kommunikátorok definiálása.

14.1. Fordítás és futtatás

Fordító és gép függő. A Portland Group compiler és MPICH könyvtár esetében a fordításnál a könyvtárat meg kell adni:

pgf90 -l mpich ....

Futtatás:

mpirun -np N test.exe

Ilyenkor az mpirun a $PGI/linux86/share/machines.Linux fájlban felsorolt gépeken futtat.

mpirun -np N -machinefile mfile test.exe

ahol az mfile fájl tartalmazza a gépek nevét, soronként egyet, a több processzort kettőspont mögötti számmal jelölhetjük:

virgo4:2 virgo5:2 virgő

mpirun -np N -machinefile mfile -nolocal test.exe

14.2. Programozási feladat: a diffúziós program párhuzamosítása MPI könyvtárral

Párhuzamosítsa a kétdimenziós diffúziós programot az MPI könyvtár felhasználásával. Ossza fel a számítási tartományt az x tengely mentén egyforma részekre. Minden processzor csak a saját részéről és az azt körülvevő szellemcellákról tud. A határfeltételeket a szellemcellák segítségével kell biztosítani.