Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

- Day 1:
  - Introduction.
  - Fortran Compilers.
  - Language Elements.
  - Expressions and Assignments.
  - Control Constructs.

Introduction

Early Programming

- In the early days of computing the programmers had the difficult task of programming directly in *octal code* – which requires a detailed knowledge of the instructions, registers and other aspects of the **C**entral **P**rocessing **U**nit (**CPU**).
- Developed mnemonic codes: *assembly* code which was transformed into octal code by the *assembler*.
- **FOR**mula **TRAN**slation = **FORTRAN**
  – John Backus (IBM) 1954.

John Backus

Example of assembler code

- FORTRAN:

```
program hello
i = 1
j = i + 4
print*,'j = ',j
end program hello
```

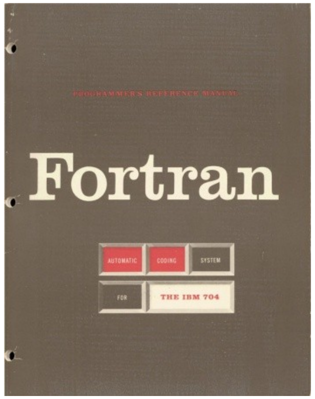
- Assembler →

```
/ FILE main.f

/    1              !      program hello
/    2              !      i = 1

{        2  }   movl    $1,-64(%rbp)            / sym=i

/    3              !      j = i + 4

{        3  }   movl    -64(%rbp),%ecx          / sym=i
{        3  }   addl    $4,%ecx
{        3  }   movl    %ecx,-60(%rbp)          / sym=j

/    4              !      print*,'j = ',j

{        4  }   movq    $.L.MAIN.SRC_LOC$1,-40(%rbp)            / sym=AUTO5
{        4  }   movl    $8,-48(%rbp)            / sym=AUTO5
{        4  }   leaq    -48(%rbp),%rcx          / sym=AUTO5
{        4  }   movq    %rcx,%rdi
{        4  }   xorq    %rax,%rax

        .cfi_GNU_args_size      0x0
{        4  }   call    __f90_sslw
{        4  }   leaq    -48(%rbp),%rax          / sym=AUTO5
{        4  }   leaq    .L.MAIN.STR$2(%rip),%rcx
{        4  }   movq    %rax,%rdi
{        4  }   movq    %rcx,%rsi
{        4  }   movq    $4,%rdx
{        4  }   xorq    %rax,%rax

        .cfi_GNU_args_size      0x0
{        4  }   call    __f90_slw_ch
{        4  }   leaq    -48(%rbp),%rax          / sym=AUTO5
{        4  }   movl    -60(%rbp),%edi          / sym=j
{        4  }   movq    %rdi,-72(%rbp)          / sym=.CV3
{        4  }   movq    %rax,%rdi
{        4  }   movq    -72(%rbp),%r15          / sym=.CV3
{        4  }   movl    %r15d,%esi
{        4  }   xorq    %rax,%rax

        .cfi_GNU_args_size      0x0
{        4  }   call    __f90_slw_i4
{        4  }   leaq    -48(%rbp),%rax          / sym=AUTO5
{        4  }   movq    %rax,%rdi
{        4  }   xorq    %rax,%rax

        .cfi_GNU_args_size      0x0
{        4  }   call    __f90_eslw
```

History of FORTRAN

- FORTRAN (1957) - IBM 704
  - 32 statements (IF, DO, GOTO, PAUSE, STOP, IO, TAPE, PUNCH).
- FORTRAN II (1958)
  - SUBROUTINE (CALL, RETURN, END).
  - COMMON (for global data)
  - DOUBLE PRECISION data types.
  - COMPLEX data types.
- FORTRAN III (1958, not released)
  - Inline assembler code.

IBM 704



Programmer's Reference Manual for Fortran on the IBM 704 (1956)

Procedural programming

History of FORTRAN

- IBM: The 704 FORTRAN, FORTRAN II, and FORTRAN III included *machine-dependent features* that made code importable from machine to machine !
- Early versions of FORTRAN provided by other vendors suffered from the same disadvantage.
- In 1961, as a result of customer demands, IBM began the development of the machine independent *FORTRAN IV* (from FORTRAN II).

History of FORTRAN

- FORTRAN IV (1962) – IBM 7030, 7090, 7094
  - By 1965 FORTRAN IV was supposed to be compliant with the standard developed by the American Standard Association (now ANSI) X3.4.3 FORTRAN Working Group.
  - Removed machine dependent features.
  - New features:
    - Logical IF statement.
    - LOGICAL data types.

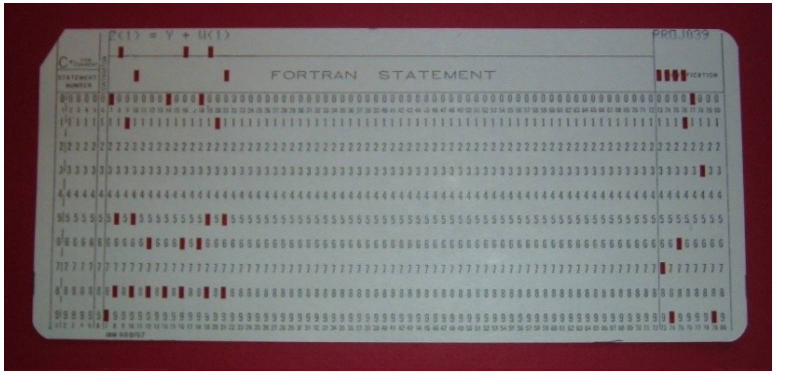
IBM 7030

In 1965 the Danish Ministry of Education bought an IBM 7090 for 1 DKK → starting NEUCC (Nordic Europe Computing Center) at DTH (now DTU).

History of FORTRAN

- FORTRAN 66 (ANSI, 1966) – two standards:
  - **FORTRAN (based on FORTRAN IV): the first “industry-standard” version of FORTRAN**
  - **Basic FORTRAN** (based on FORTRAN II, but excluding machine dependent features).

FORTRAN until FORTRAN 90 uses fixed format to support punch card storage devices.



History of FORTRAN

- FORTRAN 77 (ANSI 1978):
  - IF, THEN, ELSEIF, ELSE, ENDIF.
  - DO WHILE.
  - IMPLICIT statement.
  - CHARACTER data types.
  - DO loop extensions (fx. DO i=10,1,-1))
  - SAVE statement on local variables.
  - PARAMETER statement for specifying constants.
  - INCLUDE statement.
  - Generic names for intrinsic functions.
  - BIT manipulation.

```
SUBROUTINE SUB(info)
IMPLICIT NONE
INCLUDE 'include.inc'
REAL*8 dx
PARAMETER(dx=0.1)
REAL*8 x
INTEGER info
SAVE x
x = x + dx
y = ACOS(x)
```

Structural programming

History of FORTRAN

- FORTRAN 90 (ISO 1991, ANSI 1992):
  - **Dynamic memory allocation.**
  - Free source format – incl. 31 char. identifiers.
  - Inline comments.
  - Ability to operate on arrays (or array sections).
  - **MODULES (replacing COMMON).**
  - INTERFACE – and operator overloading.
  - DO, ENDDO construct.
  - **POINTER.**
  - **Derived data types.**
  - User define specifications of precision.
  - Support of FORTRAN 77.

History of FORTRAN

- FORTRAN 95 – minor revision:
  - FORALL + nested WHERE (to aid vectorization)
  - PURE and ELEMENTAL procedures.
  - Pointer initialization.
  - **ALLOCATABLE arrays as dummy arguments.**

History of FORTRAN

- FORTRAN 2003, 2008, 2018 (ISO: 2004, 2010)
  - Derived type.
  - Object Oriented Programming:
    - Type extensions and inheritance.
    - Polymorphism (handling different data type using a uniform interface)
  - Input/output enhancements.
  - Access to error messages.
  - Enhanced integration with host operation system.
  - Submodules.

FORTRAN standards: www.nag.co.uk/sc22wg5

FORTRAN COMPILERS

Not all vendors provide support for FORTRAN 2003 (less for 2008)
(www.ibm.com, www.nag.com, www.intel.com, gfortran)

General FORTRAN compilers

- Licensed:
  - Absoft (‘95 ANSI). ~700 USD.
  - Pathscale ('95, ('03) OpenMP, CUDA, GPGPU). ~1800 USD
  - Portland (‘03, HPF, OpenMP). ~320 USD.
  - Intel: (’95, ‘03 ('08), OpenMP) ~700 USD.
- Free:
  - Intel: (95', '03 ('08), OpenMP) – free for private use only.
  - gfortran (GNU compilers; ’95, ’03, (‘08) OpenMP) – free.
  - G95 (‘95, OpenMP) – free.
  - See status on:
    - fortranwiki.org/fortran/show/Fortran+2003+status
    - fortranwiki.org/fortran/show/Fortran+2008+status
    - fortranwiki.org/fortran/show/Fortran+2018+status

Vendor FORTRAN compilers

- Licensed:
  - IBM (‘95, ’03, OpenMP): linux (> 3000 USD).
  - HP (‘95, OpenMP): HP-UX.
  - NEC (‘95, OpenMP): SUPER-UX.
  - Cray ('95,'03, '08, OpenMP).
- Free:
  - **Sun/Oracle (’95 (‘03) OpenMP) – free !**

Language Elements
FORTRAN 95

Language Elements

- FORTRAN *character* set:
  - A-Z 0-9 = + - * / ( ) , . $ ‘ : BLANK ! “ % & ; < > ?
  - Lower cases are not included in the standard, thus
    Var = 1.0 is identical to var = 1.0 but
    string = ‘Var’ is not identical to string = ‘var’
  - **Tabulation (TAB) is NOT part of the standard.**
- *Tokens* (fx. DO, WRITE).
- *Statements* (sequence of tokens).
- *Group of statements*: program units.

Tokens

- Labels, keywords, names, constants, operators and separators: / ( ) (/ /) , = => : :: ; %
- Blanks:
  - May be used freely betw. tokens fx.: x * y = x*y
  - F77: the variable “var” could be written “v a r”, and DO ing=1,100 as: DOING=1,100
  - F90: the variable “var” may NOT be written “v a r”
  - Optional blank in: “end do”, “end if”, “go to” etc..

Source form

- Lines:
  - F77: maximum 72 characters (fixed format); executable code should start in column 7 or higher.
  - F90: maximum 132 characters (free format).
  - **ADVICE: use max. 80 characters to ensure readable hard copies (printouts).**
- Comments:
  - F77: ‘C’ in column 1.
  - F90: ! Anywhere (except character strings):

```
!  Pure commentary line
WRITE(*,'(A)') 'Warning!'   ! PRINT warning
```

Source form

- Line continuation:
  - F77: any letter in column 6.
  - F90: ampersand (&) at the end of the line:

```
array(i,j,k) = 0.5*(f(i+1,j,k) – f(i-1,j,k))*rdx   &
               + 0.5*(f(i,j+1,k) – f(i,j-1,k))*rdy
```

  - F90: maximum 39 continuations.
  - Continuation within a character string: use two &:

```
WRITE(*,'(A)') 'Example of a long text extending &
               &across multiple lines'
```

Source form

- The semi-colon (;) may be used as statement separator:

```
a = 0; b = 0; c = 0
```

- Any FORTRAN statement may be labeled:

```
100  CONTINUE ! The label: 1-5 digits.

     REAL :: buffer
     INTEGER :: n = 0
     DO
       n = n + 1
       READ(*,*,END=100) buffer
     ENDDO
100 PRINT*,'read ',n-1,' lines'
```

Type: INTEGER

- INTEGER: signed or unsigned integer value:
  – 1, 0,-999, 32767, +10
  - The default range in 32bit is: -2^{31} to $+2^{31}$
- To selected range use:
  – INTEGER, PARAMETER :: k6=SELECTED_KIND_INTEGER(6)
    here the range of k6 is (**at least**) -999999 to 999999.
  – To define constant of that range use:
    INTEGER(k6) :: k
    k = 143_k6

Notice:
2^{31} = 2.147.483.648
$1291^3 > 2^{31}$

Type: REAL

- REAL: floating point number:
  1.06E-11, 1., -0.1, 1E-1, 1.0E3, 3.1415, 1.0D0
- To select the desired range:
  INTEGER, PARAMETER :: long=SELECTED_KIND_REAL(9,99)
  1.0_long, 1.0e3_long will have 9 significant decimals and an exponent range of -99 to 99.
  - ADVICE: stick to single: KIND(1.0E0) or double precision: KIND(1.0D0). Example:
    INTEGER, PARAMETER :: MKS = KIND(1.0E0)
    INTEGER, PARAMETER :: MKD = KIND(1.0D0)
    INTEGER, PARAMETER :: MK   = MKS
    REAL(MK) :: mydata

Type: REAL

- The FORTRAN standard requires at least one additional precision than the default:
  - REAL: single precision (IEEE: 6 decimals and a range of the exponent of -37 to 37: 4 bytes)
  - DOUBLE PRECISION (8 bytes; 15 decimals; range of -307 to 307)
- Intrinsic functions: PRECISION and RANGE return the precision and range of the argument.

Type: COMPLEX

- COMPLEX: complex literal constant:
  (1.0,-3.0): the real part is 1.0 and the complex part is -3.0
  - single precision COMPLEX requires 8 bytes.
  - double precision COMPLEX (1.0D0,-3.0D) requires 16 bytes.

```
INTEGER, PARAMETER :: MKS = KIND(1.0E0)
INTEGER, PARAMETER :: MKD = KIND(1.0D0)
COMPLEX :: complex_also_single ! Single precision
COMPLEX(MKS) :: complex_single
COMPLEX(MKD) :: complex_double
complex_double = (1.0_MKD,0.0_MKD)
complex_single = (1.0,0.0) ! OR (1.0_MKS,0.0_MKS)
```

Type: CHARACTER

- CHARACTER: character literal constant
  ‘hello world’, “another hello world”
  - The characters are not limited to the FORTRAN character set.
  - Newline (fx. \n) is not permitted.
  - Spaces and case are significant.
  - Apostrophes may be included using:
    “Mary’s lamb”
    ‘he said: “anything goes” and so it &
    & does’

Type: LOGICAL

- LOGICAL: logical literal constant:
  .TRUE. or .FALSE.

```
LOGICAL:: vari
vari = .TRUE.
```

Names

- Reference to an entity by its name:
  – 1 to 31 alphanumeric characters (letters, underscores, and numerals) of which the first MUST be a letter.
  – Valid names: a, a_thing, mass, q123, real
  – Invalid names: 1a, a string, $sign
  – There are NO reserved words in FORTRAN.

Intrinsic data types

- Type declaration:

```
INTEGER, PARAMETER :: MK = KIND(1.0E0)
INTEGER :: I
REAL(MK) :: a
COMPLEX :: current
COMPLEX(MK) :: previous
LOGICAL :: first
CHARACTER(LEN=256) :: string
```

The variable MK is here a fixed value – a parameter – that cannot change during the program. KIND(1.0E0) will usually return 4 and KIND(1.0D0) the number 8 – but is compiler dependent!

Derived data types

- Definition of data type:

```
TYPE person
    CHARACTER(LEN=256) :: name
    REAL(MK) :: age
    INTEGER :: id
END TYPE person
```

- Declaration of a variable of that type:

```
TYPE(person) :: you
```

Derived data types

- Reference to elements of derived type:
  you%id
- Valid operations on derived type:
  you%id + 9
  TYPE (person) :: me
  you%age – me%age
  you – me ! Requires overloading of ‘-’ for the type person
  you = person(‘Smith’,23.5,12345)

Derived data types

- Nested types:

```
TYPE point
    REAL :: x,y,z
END TYPE point
TYPE triangle
    TYPE (point) :: a,b,c
END TYPE triangle
TYPE (triangle) :: t
x_coord_of_triangle_a = t%a%x
```

Arrays of intrinsic type

- Another compound object is an *array:*

    REAL, DIMENSION(10) :: x

Defines the vector (x(1), x(2), … x(10))

    REAL, DIMENSION(-1:8) :: y

Defines a vector of length 10: y(-1), y(0), y(1), … y(8)

    REAL, DIMENSION(5,3) :: M

Is a ***rank*-**two array of ***size****:* 5 * 3 = 15 elements.

- FORTRAN allows a maximum rank of 7 (Fortran 2008: 15).
- The ***extent*** of an array is the number of elements along the particular direction.

Arrays of intrinsic type

- The **sequence of extents** is known as ***shape***:
```
REAL(MK), DIMENSION(-2:2,7) :: array
```
  has the shape: (5,7)
- A derived type may contain arrays:
```
TYPE triangle
    REAL(MK), DIMENSION(3) :: point
    REAL(MK) :: areal
END TYPE triangle
```

Arrays of intrinsic type

- The memory layout of a FORTRAN array is in **column major order:**
  thus the array:
  REAL(MK), DIMENSION(3,2) :: M
  is stored sequentially in memory as:
  M(1,1), M(2,1), M(3,1), M(1,2), … M(3,2)
- **Thus, to access the memory in unit stride: make the inner loop increment the first index!**

C/C++: row major order i.e, last index is unit stride

Arrays of intrinsic type

```
! Example: GOOD performance:
DO j=1,N
    DO i=1,M
        array(i,j) = …
    ENDDO
ENDDO
! Example: BAD performance:
DO i=1,M
    DO j=1,N
        array(i,j) = …
    ENDDO
ENDDO
! But it also depends on the array access on the right hand side
```

Arrays of intrinsic type

- ***Sections*** (subarrays) reference part of an array:
  M(3:5): rank-one array of size 3
  M(1:i,1:j,k): rank-two array with extent: i,j
- Character arrays:
  CHARACTER(20), DIMENSION(5) :: string
  contains 5 strings each of length 20

  string(5)(2:3)
  references the 2nd and 3rd characters in line 5

Pointers

- A pointer is an *object* that can point to other objects:

  ```
  REAL, POINTER :: son
  REAL, POINTER, DIMENSION(:) :: x,y
  REAL, POINTER, DIMENSION(:,:) :: a
  ```
- Point to nothing:

  ```
  NULLIFY(son,x,y,a)
  ```

In the case of an array, only the rank (number of dimensions) is declared, and the bounds (and hence shape) are taken from the object to which it points. Given such a declaration, the compiler arranges storage for a descriptor that will later hold the address of the actual object (known as the target) and holds, if it is an array, its bounds and strides.

Pointers

- Give new storage to pointers:

```
ALLOCATE(son, x(10), y(-10:10), a(n,n))
```

- ALLOCATE was NOT available in F77 !
- Elements in derived types may be a pointer:

```
TYPE entry
    REAL :: value
    INTEGER :: index
    TYPE (entry), POINTER :: next
END TYPE entry
```

More on arrays on Wednesday

Expressions and Assignments

Expressions and Assignments

- Combine constants, keywords, and names into: *expressions* (~ phrases).
- Combine expressions to form *statements* (~sentences).
- Expression:

  operand *operator* *operand*

  Example:
  - x + y (dyadic / binary operator)
  - -y (unary / monadic operator)

Precedence

- Precedence (in order):
  1. ** (exponential).
  2. * / (multiplication and division).
  3. + - (addition, subtraction).
  4. Precedence change with ().

     Example:
     - 5*10 – 2+4/2 = 50
     - x**(-y)   (x**-y is not allowed)

Operations with integers

- For integer data, the result of any division will be truncated towards zero:
  - 6/3 is 2
  - 8/3 is 2
  - -8/3 is -2
- Exponentiation of integer numbers:
  - 2**3 is 8
  - 2**(-3) is 1/(2**3) which is zero !

    if what you need is 0.125, then write: 2.0**(-3)

Mix mode operation

- Mix mode operation:
  - The object of the weaker (or simpler) of the two data types will be converted into the same data type as the stronger one.
  - The result will also be the that of the stronger.

    Example:
  - a*i (the integer i will be converted into a real before the multiplication is performed).
  - a + 1.0 (if a is double precision, 1.0 will be converted into double: fx. 1.000000987654 and added to a).

Defined and undefined variables

- A declared variable is undefined (has no value) until assigned an explicit value:

  Example:

  ```
  INTEGER :: count
  ```

Has generally no meaningful value (it is NOT zero) unless explicitly assigned:

```
count = 0 ! count MUST be initialized before use
DO
    count = count + 1
ENDDO
```

```
INTEGER :: count = 0
DO
  count = count + 1
ENDDO
```

Scalar numerical assignment

- Scalar numerical assignment:
  variable = expr
- Conversion rules for assignment:
  integer variable = INT(expr,KIND(variable))
  real variable = REAL(expr,KIND(variable))
  complex variable =CMPLX(expr,KIND(variable))

Scalar relational operators

- Relation operators:
  - .LT. or < less than
  - .LE. or <= less than or equal
  - .EQ. or == equal
  - .NE.or /= not equal
  - .GT. or > greater than
  - .GE.or >= greater than or equal
- The result is a logical value: .TRUE. or .FALSE.
- Example: IF (x.GT.0.0) THEN or:
  IF (x>0.0) THEN

Scalar logical expressions and assignments

- Precedence:
  - Unary operator:
    - .NOT. logical negation
  - Binary operators:
    - .AND. logical intersection
    - .OR. logical union
    - .EQV. logical equivalence
    - .NEQV. logical non-equivalence

Scalar logical expressions

Example:

```
LOGICAL :: i,j,k,l,m
m = .NOT.j                 is true if j is false.
m = j.AND.k                is true if j and k are both true
m = i.OR.l.AND..NOT.j      is true if i is true or if l is
                           true and j is false.
k = .FALSE.                set k to false.
```

Scalar character expressions

- Only intrinsic operator is concatenation: //

Example:

‘AB’//’CD’   results in   ‘ABCD’

  - The length is the sum of the two characters.
- Assignment: =

Example:

CHARACTER(LEN=5) :: fill

fill(1:4) = ‘AB’

  - fill(1:4) will have the value ABbb (b stands for blank)
  - fill(5) will be undefined (which is problematic – fx for LEN_TRIM(fill); thus use fill(1:5) = ‘AB’ and LEN_TRIM will here return the value 2)

Array expressions

- FORTRAN has powerful array expressions

Example:

```
        REAL, DIMENSION(10,20) :: a,b
        REAL, DIMENSION(5) :: v
a/b ! Array of shape (10,20) with elements a(i,j)/b(i,j)
v+1. ! Array of shape (5) with elements v(i) + 1.
5/v+a(1:5,5) ! Array of shape (5) with elements 5/v(i)+a(i,5)
```

Array expressions

Example:

a(2:9,5:10) + b(1:8,15:20) ! Array of shape (8,6) with the elements a(i+1,j+4) + b(i,j+14)

- Array notation could assist the compiler in vectorizing and parallelising your code (fx. OpenMP or the Intel Xeon Phi coprocessor).
- **Do NOT use array notation !** (or use it with care !)
  – **It is less efficient than explicit loops !**
  – **It requires additional (hidden) memory!**
  – **It will crash without warning if the shapes are different!**

Pointers in expressions

- Assigning one pointer to another: copy of data

Example:

```
REAL, DIMENSION(:), POINTER :: p1,p2,p3
REAL, DIMENSION(:), ALLOCATABLE, TARGET :: p4
```

… allocate and initialize p1, p2, p3, and p4

```
p1 = p2 ! Will copy the data of p2 to p1
p1 = p4 ! Will copy the data of p4 to p1
```

- Pointer assignment: changes the target

```
p1 => p2 ! Now p1 just points to p2
```

Control Constructs

GOTO statement

- GOTO *label* (the *label* must be an executable statement)
- use it sparingly! As it makes spaghetti code !

Example:

```
      x = y + 4.0
      GOTO 4
3     x = x + 2.0
4     z = x + y
    IF (z.GT.5.4) GOTO 3
```

- ADVICE: use it (only) for error handling.

IF statement

- IF (*scalar-logical-expr*) *action-stmt*
- [name:] IF (*scalar-logical-expr1*) THEN

        action-stmt 1
    ELSEIF (*scalar-logical-expr2*) THEN
        action-stmt 2
    ELSE
        action-stmt 3
    ENDIF [name]

IF statement

- Nested IF statement: the *action-stmt* can contain another IF-THEN-ELSE-ENDIF.

  Example:

```
IF (x.GT.xmax) xmax = x
IF (x.GT.x0) THEN
    IF (y.GT.y0) THEN
        both = .TRUE.
    ENDIF
ENDIF
IF (both) PRINT*,’BOTH ARE TRUE’
```

IF statement

Example:

```
! This may fail if i > nmax:
IF (i.LE.nmax.AND.xp(i).GT.xmax) xmax = xp(i)
! This will not fail:
IF (i.LE.nmax) THEN
    IF (xp(i).GT.xmax) THEN
        xmax = xp(i)
    ENDIF
ENDIF
```

CASE – statement

- [name:] SELECT CASE (*expr*)
  [CASE *selector* [name] block] …
  END SELECT [name]
  - *expr*: scalar and of type CHARACTER, LOGICAL OR INTEGER.
  - *selector*: for CHARACTER or INTEGER, the selector may be specified by a lower and upper scalar initialization expression. Must not overlap!
  - CASE DEFAULT: defines all cases not included in the other selectors.

CASE – statement

Example:

```
SELECT CASE (number)
    CASE (:-1) ! All values below 0
        sign = -1
    CASE (0)
        sign = 0
    CASE (1:) ! All values above 0
        sign = 1
END SELECT
```

DO – construct

- [name:] DO [,] *variable=expr1,expr2[,expr3]*
  block
  ENDDO [name]
  - *variable:* is a named scalar variable; incremented **at the end** of the do loop.
  - *expr1*, *expr2*, *expr3:* are any valid scalar INTEGER expression. *expr3*: denotes the stride.
  - The number of iterations:
    MAX((*expr2-expr1 + expr3*)/*expr3*,0)

DO – construct

- Thus: DO i=1,n will have zero iterations if n < 1
- [name:] DO
  - Unbounded do – will perform an endless loop.
  - EXIT [name] will terminate the endless loop and return to the next executable statement after the [named] ENDDO it refers to.
  - CYCLE [name] will return to the ENDDO of the corresponding construct.

DO – construct

- It is not allowed to jump into a loop !
- Any number of DO constructs may be nested.
- An alternative (old) DO construct:

```
      DO 10 i=1,n
         block
   10 CONTINUE
```

Print and read

- PRINT*,’var1 = ‘,var1,’ var2 = ‘,var2
- READ*,var1,var2

…. Will allow us to start the exercises !

Example

```
! Small program
PROGRAM main
INTEGER, PARAMETER :: Nx = 20, Ny = 20
REAL, DIMENSION(Nx,Ny) :: field1,field2
REAL :: dx,dy,Lx,Ly,rdx2,rdy2,stime,dt,diff
INTEGER :: istep,nstep

! Initialize
Lx = 1.0; Ly = 1.0
dx = Lx/REAL(Nx-1); dy = Ly/REAL(Ny-1)
rdx2 = 1.0/dx**2; rdy2 = 1.0/dy**2
nstep = 200
diff = 1.0
dt = 0.25*MIN(dx,dy)**2/diff

! Perform time stepping
DO istep=1,nstep
    ! compute
    DO j=2,Ny-1
        DO i=1,Nx-1
           field1(I,j) =
        ENDDO
    ENDDO

ENDDO
!Output

END PROGRAM main
```

Available on login.gbar.dtu.dk:
cp   ~jhwa/example.f   .

To edit this file use:

- vi example.f   OR
- emacs example.f OR
- nedit example.f OR
- gedit example.f

To compile and link this file use:

- f90 –free example.f (executable will be: a.out)
- f90 –free example.f –o example.exe

To execute this program use:

- ./a.out
- ./example.exe