NOAA NESDIS
CENTER for SATELLITE APPLICATIONS and RESEARCH

TRAINING DOCUMENT

TD-11.1.A
TRANSITION FROM FORTRAN 77 TO FORTRAN 90

Version 3.0

TITLE: TD-11.1.A: TRANSITION FROM FORTRAN 77 TO FORTRAN 90 VERSION 3.0

AUTHORS:

Ken Jensen (Raytheon Information Solutions)

VERSION HISTORY SUMMARY

Version	Description	Revised Sections	Date
1.0	New Training Document TD-12.1 adapted from "Fortran 90 for the Fortran 77 Programmer" © 1993 and 1996, Bo Einarsson and Yurij Shokin (copy permission granted - http://www.nsc.liu.se/~boein/f77to90/f77to90.html). Adaptation by Ken Jensen (Raytheon Information Solutions).	New Document	03/31/2006
1.1	Revision by Ken Jensen (Raytheon Information Solutions). STAR standard style applied to document.	All	06/02/2006
2.0	Revision by Ken Jensen (Raytheon Information Solutions). Renamed TD-12.1.2. Updated approvals for version 2 of the STAR Enterprise Product Lifecycle (EPL).	Pages 1, 3, 5. Header	09/28/2007
3.0	Renamed TD-11.1.A and revised by Ken Jensen (RIS) for version 3.	1	10/1/2009

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Fortran 90 for the Fortran 77 Programmer

The programming language FORTRAN is the principal language for scientific and technical computations. It was developed originally in 1954 and has been revised several times, to FORTRAN IV and Fortran 77. It has recently been carefully revised, resulting in a modern and powerful language, Fortran 90.

The intent with this new standard is to make Fortran into a useful and efficient language for the scientific and technical computations also towards the end of this decade. The new version contains powerful new features for the treatment of vectors and matrices, several new possibilities to specify the precision, access to environment parameters, intrinsic functions for manipulating floating point numbers, internal procedures and new specifications for storage and interfacing. In addition there is a new improved layout of the source code, new conditional statements, recursion and dynamic memory allocation. Nothing was removed, so Fortran 90 contains the whole of Fortran 77, but offers both easier programming and improved security.

An important requirement at the introduction of a new language is nowadays that the language should be efficient also on parallel systems. This tutorial therefore contains an Appendix on the recent proposed addition to Fortran, High Performance Fortran.

This tutorial assumes that the reader is experienced in Fortran 77. Those parts of Fortran 90 that are already in Fortran 77 are not discussed here. The many programming examples and user exercises illustrate the programming technique and available commands, and makes it easy to get started with writing programs using the new facilities in Fortran 90. All examples have been tested on Sun SPARC (UNIX) and DEC Station (ULTRIX), and on IBM PC with MS-DOS 5 and 6.2.

1.2 Preface

This tutorial is written in order to ease the transition from the very common and popular programming language Fortran 77 to the more modern Fortran 90. This transition uses the fact that Fortran 77 is a pure subset of Fortran 90. There are, however, two very important reasons to go over to as much Fortran 90 as possible. One is that it includes new and powerful constructs, the other is that Fortran 90 gives us more facilities for correctness checking of the program. This means that more reliable programs are obtained. During 1995 Fortran 90 will be offered by most computer manufacturers and the language will be a success.

It is required that the reader is knowledgeable in Fortran 77. Those parts of Fortran 90 who already are parts of Fortran 77 are not treated systematically in full, they are assumed known by the reader. Those who don't know Fortran 77, can read a tutorial book on Fortran 77, or a complete textbook on Fortran 90. Note especially that Fortran 90 is much larger than Fortran 77 in all respects. Therefore it is difficult to describe it as short as we do it here. All examples in this tutorial have been run on a Sun SPARC, DEC station Ultrix and the IBM PC with the Fortran 90 system from NAG. They were also tested on the Power Macintosh using the Absoft Fortran 90 compiler. It is recommended that the reader has access to a Fortran 90 system.

We assume that the reader can write programs in Fortran 77 and wishes to learn to use the new facilities in Fortran 90. All statements in Fortran 77 are explained in Appendix 2 and the summary of the news in Fortran 90 are in Appendix 3.

The greater power in Fortran 90 means that the statements in many cases have a combined effect, and it is therefore not so useful to only describe the language statement for statement.

The examples in the tutorial are considered to illustrate the programming technology and the statements being discussed.

The purpose is, however, not to give an optimized application program. This is especially true for the sections supplying comments to the exercises. Please note that some of the later examples are very complete with respect to interfaces and specifications that are required at the use of functions and subroutines.

1.3 Transition from Fortran 77 to Fortran 90

- Everything that is in Fortran 77 is also in Fortran 90.
- Many new statements have been added, some replacing older statements.
- Many new statements have been added and give new possibilities.
- There are now two forms of the source code. The old source code form, which is based on the punched card, and now called fixed form and the new free form.

  These two forms may not be mixed independently but must be separated at the compilation; an old program, (one or several program units in the form of a main program, subroutines or functions), which is compiled in free form, has the possibility to give a different result compared with earlier when it was compiled with fixed form, compilation errors are possible.

  In the same program we can mix program units written in fixed form and free form, but each unit must be only in one form and at the compilation both forms may not usually be in the same source file. Some systems using a special directive may permit program units in different form in the same file. The directive then tells the compiler which form is valid.

- Some old statements are to be avoided.
- It is possible to mix old and new statements, but it is advised to try to be consistent, which means using either the old or the new form for the statement.
- Note that when you switch from one compiler to another new errors may occur, because the old compiler was not as strict as the new one. Normally, a new compiler discovers some old errors that were not found earlier. In the specification for Fortran 90 it is required that errors already be found at compilation if this is at all possible.

2. SPECIFICATIONS

The Fortran 90 statement `IMPLICIT NONE` means that the implicit declaration is no longer used. This statement has been available in some implementations of Fortran 77. The use of that statement means that the probability of errors because of incorrectly spelled variable names is drastically reduced. The first difference regarding specification of variables is that these can now be put together in one statement for each variable. Using Fortran 77 you can declare, for example, as follows

```
     REAL A, B, C
     PARAMETER (A = 3.141592654)
     DIMENSION B(3)
     DATA B /1.0, 2.0, 3.0 /
     DIMENSION C(100)
     DATA C /100*0.0/
```

that is one variable can occur in several lines. Using Fortran 90 you can instead write

```
     REAL, PARAMETER        :: A = 3.141592654
     REAL, DIMENSION(1:3)   :: B = (/1.0, 2.0, 3.0 /)
     REAL, DIMENSION(1:100) :: C = (/ ( 0.0, I = 1, 100 ) /)
```

The difference is not so large here but it's much larger in more complicated examples with many variables, especially since in Fortran 90 you have access to more properties. In the last example an extra parenthesis `( )` is required.

The last example can also be generalized to assign different values to different parts of the vector.

```
 REAL, DIMENSION(5) :: D = (/ (4.0, I = 1, 3) , (17.0, I = 4, 5) /)
```

Since there are variables of different types there is an intrinsic function that shows the exact subtype of the variable used. This function is called `KIND(x)`. This function can also be used for particular types of subtypes or kinds:

```
    KIND (0)                   integer
    KIND (0.0)                 floating point number
    KIND (.FALSE.)             logical variable
    KIND ("A")                 string of characters
```

There is an intrinsic function `SELECTED_REAL_KIND`, which returns the kind of the type `REAL` that has a representation with (at least) the precision and the exponential range requested. For example, the function `SELECTED_REAL_KIND (8,70)` is that kind of `REAL` that has at least 8 decimal digits accuracy and permits an exponent between `10**-70` and `10**70`. The corresponding function for integers is called `SELECTED_INT_KIND` and of course has only one argument. With the choice `SELECTED_INT_KIND (5)` all integers between (but not including the limits) `-100 000` and `+100 000` are permitted. The kind of a type can be given a name.

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

This kind of integers can be used in constants according to the following line

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

which is a rather unnatural specification, after the value we have to give an underscore _ followed by the name of the kind.

Use of variables of the new integer type can be declared in a nicer way

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

The corresponding is true for floating-point variables, if we first introduce a high-precision kind `LONG` with

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

then we get the floating-point kind with at least 15 decimal digits accuracy and with an exponent range from `10**-99` to `10**+99`. The corresponding constants are obtained as

```
     2.6_LONG
     12.34567890123456E30_LONG
```

and variables are declared with

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

The old type conversions `INT, REAL` and `CMPLX` have been extended with the functions

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

which converts a floating-point number `X` to an integer of the kind `K5`, if `Z` is a complex number with

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

you get it converted to a floating-point number of the real type and of the same kind of `Z` (that is of course the real part of `Z`).

Double precision is not included in the new Fortran 90 in any other way than in the "old" Fortran 77, but it is assumed that the compiler supports the double or quadruple precision that may be available in the hardware. You can then define a suitable kind of the `REAL`, named `DP` or `QP`. You can of course use the old concept of `DOUBLE PRECISION`.

The reason for this rather cumbersome convention is that it is not desirable to have too many compulsory precisions (for example single, double, quadruple, perhaps for both the cases `REAL` and `COMPLEX`) and also that the old concept `DOUBLE PRECISION` did not give a specified machine accuracy. Now you can relatively easily specify both which precision and which range of exponent you wish to use. Additional information about the kind is given in Appendix 6, where the different data types and their normal kinds for the NAG compiler on DEC, SUN, and the IBM PC, the Cray compiler, and the Absoft compiler on the Power Macintosh.

Exercises

(2.1) What does the specification `LOGIC ALL` mean?

(2.2) Specify a constant `K` with the value `0.75`.

(2.3) Specify an integer matrix `PELLE` with 3 rows and 4 columns.

(2.4) Specify a floating-point number which corresponds to the double precision on an IBM and a single precision on Cray.

(2.5) Specify some variables of the type above.

(2.6) Specify some constants of the type above.

(2.7) Is the following specification correct?

```
REAL DIMENSION(1:3,2:3) :: AA
```

(2.8) Is the following specification correct?

```
REAL REAL
```

(2.9) Is the following specification correct?

```
COMMON :: A
```

3. FREE FORM AND FIX FORM

Sometimes we require more than one line for a statement

```
   Print *, 'This is a long output line',&
            ' this is the second part',&
            ' and this is the third part!'
```

Nowadays, in the free form, we continue a line with the symbol "&" (called ampersand), i.e. with the sign & at the end of the old line instead of an almost arbitrary character in column 6 of the new line. With the compiler we are now using it is possible to include the Swedish characters in character strings and in comments.

Sometimes a certain identifier or a certain numerical number does not fit on one line. We can then interrupt the identifier anywhere with the character "&" and then on the next line give a new "&" as the first non-blank character. You continue then directly from this "&" without any extra blank. The character "&" therefore works as a kind of delimiting or syllabification sign. You can write

```
     PI = 3.141592653589793
```

or you can write equivalently

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

Please note that comment lines can not be continued. The reason for this is that in a comment line the sign "&" is also treated as belonging to the comment. However, you can also add comment lines inside the continuation lines. Note that it is the final "&" of a line that indicates a continuation line, it is therefore possible to write a text string of characters including the character "&".

Sometimes you may wish to do it in the opposite way, to have several statements on the same line. This is done with the use of semicolon. And with the exclamation mark we can include a comment on the same line.

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

A line may include up to 132 characters, a statement may have up to 39 lines of continuation.

Note that in the free form, blanks are significant, as can be seen in my favorite example:

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

This gives a compilation error since the compiler does not find a comma between the lower and upper the limits, but the compressed version

```
     DO25I=1.25
```

gives the same result as the non-compressed form and as in Fortran 77 or using the old form (fix form) of Fortran 90, namely that the variable `DO25I` is being assigned the value 1.25.

Comments are started with "!" (exclamation mark) and ended with the end of line. The old types of comments introduced with `C` or `*` in column 1 are no longer permitted if you use free form, but are, of course, if you use the fix form. Upper case and lower case characters are equivalent except in character strings.

The above applies to the new free form. In the old, column-oriented or fix form, we can also use a semicolon or exclamation mark between columns 7 and 72 but you can not continue with "&" in columns 1 to 6 or 73 to 80 or write comments in columns 1 to 6. Exclamation mark in column 1 of course means a comment line also in the old fix form. Some possibilities of longer lines do exist within the old fix form (implementation dependent).

Exercises

(3.1) What does the following line mean?

```
  A = 0.0 ; B = 370 ! First variables ; C = 17.0 ; D = 33.0
```

(3.2) Are the following lines correct according to Fortran 90?

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

4. FORMAT

The old Fortran programs used numbering of the format statements. However, that doesn't look very good in pure Fortran 90, where you don't use statement numbers except in a few exceptional cases. In Fortran 77 there already was a facility, which however was not used very much, to use a format variable (of type `CHARACTER`) instead of a numbered format. The Format variable was put directly in the input/output statement. Now we will show three different ways of doing this assignment. They both have their advantages and disadvantages. The program follows

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

In the `PRINT` statement we use the character string variable `FORM1` with the length 11, which is assigned its value in an explicit assignment statement. The difficulty with this method is essentially that you have to manually count the number of characters, if it is too small the NAG compiler will not give a compilation error, but the error will show up at execution.

In the first `WRITE` statement we use a character string constant `FORM2` instead. The advantage is that with the `PARAMETER` statement it is not necessary to give an explicit length of the constant, but it can be given the length with the statement `LEN=*`. The bad thing is that we can not assign the constant a new value.

In the second `WRITE` statement we use an explicit character string directly. The difficulty is that the string can not be reused.

Exercises

(4.1) What does this statement give as its output?

```
     WRITE(*, "( HI )")
```

(4.2) What does the following statement perform?

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

5. USE OF THE SAME SOURCE CODE

This is possible if you do it in the following way, that is if you use the new continuation sign "&" at the end of the old line, but in position 73 so that it doesn't conflict with Fortran 77, and also choose the "&" sign as the almost arbitrary character in column 6, in order to get continuation according to Fortran 77. An introductory "&" is "in principle" neglected by Fortran 90.

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

This is really not standard Fortran 77 since neither "&" nor "!" are in the standardized character set. On the other hand, these constructs are perfect for program segments that are to be included in Fortran 90 program units using the `INCLUDE` statement (refer to Appendix 3, section 1), sometimes using the old form and sometimes using the new form of the source code. The program segment is then supposed to be written as if blanks were significant.

Comments are an incompatibility problem between Fortran 77 and Fortran 90, but of course not between fixed and free forms of Fortran 90 since the "!" is permitted in both. The exclamation mark is also permitted in Sun and DEC Fortran 77 (both DEC Station ULTRIX and VAX VMS), the Cray compiler CF77, and the Absoft Fortran 77 compiler.

Another important condition is of course that the program really obeys the standard.

6. CONTROL STATEMENTS

As conditional or control statements you have IF in many variants (but essentially not changed from Fortran 77), `DO` (with some new variants) and the completely new statement `CASE`.

The DO-loop should now be ended with the statement `END DO` and we no longer need any statement number. In addition, we can use the statement `EXIT` to jump out of the `DO`-loop and `CYCLE` in order to go to the next iteration of the present `DO`-loop. A `DO`-loop can be assigned a name, which is done by giving the name before the `DO` and followed by a colon. In addition the final `END DO` can be followed by the name of the `DO`-loop.

```
       SUMMA = 0.0
ADAM : DO I = 1, 10
               X = TAB(I)
EVA :          DO J = 1, 20
                      IF (X > TAB(J)) CYCLE ADAM
                      X = X + TAB(J)
               END DO EVA
               SUMMA = SUMMA + X
               IF (SUMMA >= 17.0) EXIT ADAM
       END DO ADAM
```

In the example above, the execution of the inner loop will be interrupted with a jump to the next cycle of the outer loop, and thus the variable sum or `SUMMA` will not be increased, if `X` is greater than the given table value. As soon as the sum is at least 17 the outer loop is also interrupted.

If no name is given in the `EXIT` or `CYCLE` statements the present inner loop is automatically used. With the present inner loop I mean the one where the `EXIT` or `CYCLE` statements being executed are. These statements then replace the `GOTO` to the final statement, which was often used in the old `DO`-loop. This final statement usually was a `CONTINUE` statement.

An `IF` statement can also be given a name. In that case the corresponding `END IF` ought to be followed by that name.

A new construct in standard Fortran is `CASE`. It appeared, however, in many Fortran dialects before. It can choose a suitable case for a scalar argument of type `INTEGER`, `LOGICAL` or `CHARACTER`. A simple example is based on an integer `IVAR`.

```
SELECT CASE (IVAR)
   CASE (:-1)                        !  all negative numbers
          WRITE (*,*)  'Negative number'
   CASE (0)                          !  zero case
          WRITE (*,*) ' Zero'
   CASE (1:9)                        !  one-digit number
          WRITE (*,*) ' Digit ', IVAR
   CASE (10:99)                      !  two-digit number
          WRITE (*,*) ' Number ', IVAR
   CASE DEFAULT                      !  all remaining cases
          WRITE (*,*) ' Number too big'
END SELECT
```

It is not permitted with overlapping arguments. This means that one single argument may not satisfy more than one of the cases of `CASE`. The default case does not have to be included. If no valid case is found the execution will continue with the first statement after the `END SELECT`. I recommend that you include a `DEFAULT` and then give an error message if an argument has a not permitted value.

It is recommended to use the `CASE` instead of an assigned or computed `GOTO` statement, or an arithmetic `IF`-statement.

Exercises

(6.1) Write a `CASE`-statement that performs three different calculations depending on whether the variable is negative, zero or any of the first odd prime numbers (3, 5, 7, 11, 13) and performs nothing in all other cases.

(6.2) Write a `DO`-loop that adds the square roots of 100 given numbers, but skips negative numbers and concludes the addition if the present value is zero.

7. PROGRAM UNITS

In addition to the four old program units: `PROGRAM` (that is the main program), `SUBROUTINE`, `FUNCTION` and `BLOCK DATA`, the new concept `MODULE` has been added, as well as some new things in the old units. Once again, subprogram is the common concept for both `SUBROUTINE` and `FUNCTION`.

Again I wish to emphasis that under Fortran 77 all program units are essentially on the same level, even if the main program logically is superior to the subroutines and functions that are called, and even though you could write a call map that looks like a tree. In reality the `BLOCK DATA` is on a higher level and all the other program units are on the same level, from the Fortran system viewpoint with the main program just a little above. The exception are the so-called statement functions with definitions that have to be first in a program unit, directly after the specification, and are internal to that unit and therefore on a logically lower level. Regrettably, the typical Fortran 77 programmer does not use statement functions.

The above means that all routine names are on the same logical level, which means that two different routines, and two different parts of a big program are not permitted to have the same name. Quite often numerical and graphical libraries include thousands of functions and subroutines, and each routine name consists of at most six characters under old Fortran standards. Therefore, there is a very great risk of a conflict of names. This problem could be partially solved by the old statement functions, since these are internal to the respective unit, and therefore different statement functions can have the same name if they are in different units. The disadvantage is that they can only treat what is in only one program line. But they can call each other in such a way that a later statement function can call an earlier statement function, but of course not the opposite.

F90 adds internal functions and internal subroutines, providing greater flexibility. They are specified at the end of each program unit (but not in the `BLOCK DATA`) after the new command `CONTAINS` and before the `END`. An internal subprogram can have access to the same variables as the unit it belongs to, including the possibility of calling the unit's other internal subprograms. It is written as an ordinary subprogram, but it is not permitted to have any internal functions or subroutines. The internal function is a modern replacement for the statement function.

The usual subroutines and functions remain the same as the earlier external subroutines and external functions, but there is now a more compelling reason for this name (that is calling them external) than earlier, since now you have also internal subprograms. Previously you only had the built in (intrinsic) functions as an alternative. In addition, the number of intrinsic functions has greatly increased, and a few intrinsic subroutines have been added.

For every argument in the specification of variables for subprograms we can now give its `INTENT` as `IN, OUT` or `INOUT`. If `IN` is valid, then the actual argument can be an expression like `X+Y` or `SIN(X)` or a constant like 37, since the value is only to be transferred to the subprogram, but a new value is not to be returned to the calling unit. The variables in this case may not be assigned a new value in the subprogram. If `OUT` is valid, on the other hand, the actual argument has to be a variable. At entry to the subprogram the variable is at this stage considered to be not defined. The third case covers both possibilities, one value on input and another on output, or possibly the same value. In this case the actual argument must also be a variable. If a variable has a pointer attribute then `INTENT` may not be given. The implementation of `INTENT` is not yet complete in all compilers.

One use for the new program unit `MODULE` is to take care of global data. As such it replaces the `BLOCK DATA`. Its other use is to make a package of new data types. A rather long example would be a package for interval arithmetic. Corresponding to each value `X` you have an interval `(X_lower; X_upper)`. When using the package, you want to give only the variable name `X` when you mean the interval. The variable `X` is then supposed to be a new data type, interval. The following is in the file `interval_arithmetics.f90` or `intv_ari.f90`.

```
MODULE INTERVAL_ARITHMETICS
       TYPE INTERVAL
              REAL LOWER, UPPER
       END TYPE INTERVAL
       INTERFACE OPERATOR (+)
              MODULE PROCEDURE INTERVAL_ADDITION
       END INTERFACE
       INTERFACE OPERATOR (-)
              MODULE PROCEDURE INTERVAL_SUBTRACTION
       END INTERFACE
       INTERFACE OPERATOR (*)
              MODULE PROCEDURE INTERVAL_MULTIPLICATION
       END INTERFACE
       INTERFACE OPERATOR (/)
              MODULE PROCEDURE INTERVAL_DIVISION
       END INTERFACE
CONTAINS
       FUNCTION INTERVAL_ADDITION(A, B)
              TYPE(INTERVAL), INTENT(IN) :: A, B
              TYPE(INTERVAL) :: INTERVAL_ADDITION
              INTERVAL_ADDITION%LOWER = A%LOWER + B%LOWER
              INTERVAL_ADDITION%UPPER = A%UPPER + B%UPPER
       END FUNCTION INTERVAL_ADDITION

       FUNCTION INTERVAL_SUBTRACTION(A, B)
              TYPE(INTERVAL), INTENT(IN) :: A, B
              TYPE (INTERVAL) :: INTERVAL_SUBTRACTION
              INTERVAL_SUBTRACTION%LOWER = A%LOWER - B%UPPER
              INTERVAL_SUBTRACTION%UPPER = A%UPPER - B%LOWER
       END FUNCTION INTERVAL_SUBTRACTION

       FUNCTION INTERVAL_MULTIPLICATION(A, B)
!             POSITIVE NUMBERS ASSUMED
              TYPE(INTERVAL), INTENT(IN) :: A, B
              TYPE (INTERVAL) :: INTERVAL_MULTIPLICATION
              INTERVAL_MULTIPLICATION%LOWER = A%LOWER * B%LOWER
             INTERVAL_MULTIPLICATION%UPPER = A%UPPER * B%UPPER
       END FUNCTION INTERVAL_MULTIPLICATION
       FUNCTION INTERVAL_DIVISION(A, B)
!             POSITIVE NUMBERS ASSUMED
              TYPE(INTERVAL), INTENT(IN) :: A, B
              TYPE(INTERVAL) :: INTERVAL_DIVISION
              INTERVAL_DIVISION%LOWER = A%LOWER / B%UPPER
              INTERVAL_DIVISION%UPPER = A%UPPER / B%LOWER
       END FUNCTION INTERVAL_DIVISION
END MODULE INTERVAL_ARITHMETICS
```

Compilation of the above results in the creation of the file `interval_arithmetics.mod`. This file includes an interesting modified version of the code above. Any program that makes use of this package, must contain the statement `USE INTERVAL_ARITHMETICS` at the beginning of the specification statements. This makes the data type `INTERVAL` and the four arithmetic calculations on this type directly available. In some cases it is desirable to only include some of the facilities in a module. In this case you use the `ONLY` attribute within the new `USE` statement.

```
       USE module_name, ONLY : list_of_chosen_routines
```

The following is an example of a very simple main program for the test of interval arithmetics. It is from the file `interval.f90` or `intv.f90`.

```
USE INTERVAL_ARITHMETICS
IMPLICIT NONE
TYPE (INTERVAL) :: A, B, C, D, E, F
A%LOWER = 6.9
A%UPPER = 7.1
B%LOWER = 10.9
B%UPPER = 11.1
WRITE (*,*) A, B
C = A + B
D = A - B
E = A * B
F = A / B
WRITE (*,*) C, D
WRITE (*,*) E, F
END
```

Running this program on a Sun-computer with the NAG compiler results in the following output:

```
% f90 interval_arithmetics.f90 interval.f90
interval_arithmetics.f90:
interval.f90:
% a.out
   6.9000001  7.0999999  10.8999996  11.1000004
  17.7999992 18.2000008  -4.2000003  -3.7999997
  75.2099991 78.8100052   0.6216216   0.6513762
% exit
```

We compiled the program with the compiler `f90`, and the executable program was automatically named `a.out`. With the order above (the module first) the compilation also works with the SunSoft and Digital compilers!

In a module some concepts can be defined as `PRIVATE`, which means that the program units outside of this module are not able to use this concept. Sometimes an explicit `PUBLIC` declaration is used, normally `PUBLIC` is default. The following statements

```
PRIVATE
PUBLIC :: VAR1
```

result in all variables being local, except `VAR1`, which is global. Note that both these concepts (`PUBLIC` and `PRIVATE`) either can be given as a statement, for example

```
INTEGER                           :: IVAR
PRIVATE                           :: IVAR
```

or as an attribute

```
INTEGER, PRIVATE                  :: IVAR
```

Exercises

(7.1) Generalize the modules so that the package can accomodate both positive and negative numbers even with multiplication and division.

(7.2) Generalize the modules so that the package produces a suitable error message when it divides by an interval that contains zero.

(7.3) Extend the modules so that the local rounding error at the operation is also appropriately dealt with. (This is not the case at the moment.)

8. KEYWORD AND DEFAULT ARGUMENTS

Routines can now be called with keyword arguments and can use default arguments. This means that some arguments can be given with keywords instead of their position, and some arguments do not have to be given at all, in which case a standard or default value is used.

The use of keyword and default arguments is not just as simple as it appears in the Appendix 3, section 6 program units, where an explicit interface is required. Therefore, we give a complete example here. The formal parameters of the interface are used as keywords. They need not be the same names as in the actual subprogram. They are not specified in the calling program unit.

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

        REAL X
!       Note that A, B and N are not specified as REAL
!       or INTEGER in this unit.

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

        SUBROUTINE SOLVE(A, B, N)
        REAL, OPTIONAL, INTENT (IN)  :: B
        IF (PRESENT(B)) THEN
               TEMP_B = B
        ELSE
               TEMP_B = 20.0
        END IF
        A = TEMP_B + N
```

```
      RETURN
      END
```

Note that the statement `IMPLICIT NONE` in the main program does not transfer automatically to the subroutine `SOLVE`. This subroutine, therefore, ought to be given its own `IMPLICIT NONE` statement and all variables used (`A, B, N,` and `TEMP_B`) specified.

The program is compiled and run with the following statements

```
      % f90 program.f90
      % a.out
         60.0000000
          1.1000000E+02
          1.2000000E+02
      %
```

In the last call above, where the variable B is not given explicitly, the default argument is used. This means that the default value 20 is added to the actual argument `N = 100`, which results in `A = 120`.

It is convenient to place the interface `INTERFACE` in a module so the user does not have to worry so much about it. The interface is a natural complement to the routine library. Fortran 90 looks automatically for modules in the present directory, in the directories given in the `I`-list and also in `/usr/local/lib/f90:` the standard library for Fortran 90 using UNIX. The concept `I`-list can be used to introduce a directory where various modules may be located, as explained in some of the system oriented sub-pages of Appendix 6. If you forget `INTERFACE` or have an incorrect interface, usually compilation or execution gives the error message "Segmentation error", and nothing more.

Note that if an output variable is given as `OPTIONAL` and `INTENT(OUT)`, then you have to have it included in the argument list, if when the program is executed, it assigns a value to this variable. You can not therefore use only `OPTIONAL` in order to choose whether you wish to have a certain variable outputted or not. The solution to this problem is to use the `PRESENT` statement also.

Exercises

(8.1) Write a routine for the calculation of an integral of a function. You will use keyword arguments and default arguments so that

- if there is no left integration limit `A`, the value zero will be used.
- if there is no right integration limit `B`, the value one will be used.
- if there is no tolerance keyword `TOL`, the value 0.001 will be used for the absolute error.

(8.2) Write the interface that is required in the calling routine in order to use the above integration routine.

9. RECURSION

A completely new capability of Fortran 90 is recursion. Note that it requires that you assign a new property `RESULT` to the output variable in the function declaration. This output variable is required inside the function as the "old" function name in order to store the value of the function. At the actual call of the function, both externally and internally, you use the outer or "old" function name. The user can therefore ignore the output variable.

Here follows two examples: first the recursive calculation of factorials, then the recursive calculations of the Fibonacci-numbers. The later is very inefficient. Brainerd, Goldberg and Adams (1990), page 226, propose an efficient, but non-recursive method.

The listings of the routines follow. The output variables are called `FAC_RESULT` and `FIBO_RESULT`, respectively.

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

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

The reason that the above calculation of the Fibonacci-numbers is so inefficient is that each call with a certain value of N generates two calls for the routine, which in its turn generates four calls, and so on. Old values (or calls) are not re-used.

On page 222 Brainerd, Goldberg and Adams (1990) demonstrate an interesting use of recursive technique for the calculation of an exponential function of a matrix. They give the immediate (straight forward) expression, with the successive multiplication with a matrix, as well as a recursive variant, which can pick out the suitable squares to optimize the calculation. Recursion is also excellent to code an adaptive algorithm, see exercise 9.2 below.

Another very important usage of the `RESULT` property and the output variable is with array valued functions. It is very easy to specify an output variable so that it can store all the values of such a function. Actually, it is the combination of recursive functions and array valued functions that have forced the committee to introduce the `RESULT` property.

We note that not only functions, but subroutines too, can be recursive.

Exercises

(9.1) Write a routine for the calculation of Tribonacci numbers. These are formed like the Fibonacci-numbers, but you start with three numbers (all 1 at the start). At each step you then add the last three numbers to get the next one. Run and calculate `TRIBONACCI(15)`. Note that the calculation time increases very quickly with the argument.

(9.2) Write an adaptive routine for quadrature, i.e. calculation of a definite integral on a certain interval.

11. ARRAYS AND ARRAY SECTIONS

The English word "array" is translated into Swedish as "fält", which, retranslated into English, is "field". It is possible therefore, that we may use the word field either by mistake, or as a suitable name of a specific array.

A new feature of Fortran 90 is that you can work directly with a whole array or an array section without explicit (or implicit) `DO`-loops. In the old Fortran you could in some circumstances work directly with a whole array, but then only during I/O processing.

An array is defined to have a shape, given by its number of dimensions, called "rank", and the extent of each dimension. Two arrays agree if they have the same shape. Operations are normally done element by element. Please note that the rank of an array is the number of dimensions and has nothing to do with the mathematical rank of a matrix!

In the following simple example I show how you can assign matrices with simple statements like `B = A`, how you can use the intrinsic matrix multiplication `MATMUL` and the addition `SUM`, and how you can use the array sections (in the example below I use array sections which are vectors).

```
PROGRAM ARRAY_EXAMPLE
       IMPLICIT NONE
       INTEGER                    :: I, J
       REAL, DIMENSION (4,4)      :: A, B, C, D, E
       DO I = 1, 4                ! calculate a test matrix
               DO J = 1, 4
                      A(I, J) = (I-1.2)**J
               END DO
       END DO

       B = A*A          ! element for element multiplication
       CALL PRINTF(A,4) ;  CALL PRINTF(B,4)
       C = MATMUL(A, B) ! internal matrix multiplication
       DO I = 1, 4      ! explicit matrix multiplication
               DO J = 1, 4
                       D(I, J) = SUM( A(I,:)*B(:,J)  )
               END DO
       END DO
       CALL PRINTF(C,4) ;  CALL PRINTF(D,4)
       E = C - D        ! comparison of the two methods
       CALL PRINTF(E,4)
CONTAINS
       SUBROUTINE PRINTF(A, N) ! print an array
       IMPLICIT NONE
       INTEGER                    :: N, I
       REAL, DIMENSION (N, N)     :: A
       DO I = 1, N
              WRITE(*,' (4E15.6)')  A(I,:)
       END DO
       WRITE(*,*)       ! write the blank line
       END SUBROUTINE PRINTF
END PROGRAM ARRAY_EXAMPLE
```

As mentioned in chapter 9 about recursion, functions in Fortran 90 can be array valued. In that case the use of the `RESULT` property is recommended to specify a result variable that is supposed to store the array.

Fortran 90 has many more possibilities than Fortran 77 permitting the dynamic allocation of memory. In Fortran 77 this could only could be done when a sufficient storage area had been allocated in the calling program unit, and both the array name and the required dimension(s) had to be included as parameters in the call of the subprogram. This is the adjustable array concept. A very simple case is where the last dimension is given simply with a *, or assumed-size array.

Now we also have allocatable arrays, automatic arrays, and assumed-shape arrays. Dynamic allocation using pointers is discussed in a section of the next chapter. An overview is given in Appendix 3 (section 10). Also see Appendix 9 for an explanation of certain terms.

Exercise

(11.1) Write a routine for the solution of a system of linear equations using Gaussian elimination with partial pivoting.

12. POINTERS

12.1 Introduction.

Pointers have been included in Fortran 90, but not in the usual way as in most other languages, with pointer as a specific data type. Here they are rather understood as an attribute to the other data types. The reason for this new way to introduce them is that by having a special data type for pointers, the risk of erroneous use of the pointer is very large. A variable with a pointer attribute can be used as a usual variable and in some new ways. Pointers in Fortran 90 are thus not memory addresses as in other programming languages (and in certain Fortran implementations) but rather an extra name (alias).

The increased security is obtained not only through that each variable, which shall be used as a pointer, must be given an attribute `POINTER`, but also that all variables, that will be pointed to, must be given an attribute `TARGET`. An example explaining how to do this follows.

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

The matrix `B` has been specified completely, i.e. with the dimensions given explicitly. In addition, it has been stated that it can be the target of a pointer. The matrix `A`, which can be used as a pointer, has to be declared as a matrix, i.e. to be given a correct number of dimensions, a correct rank, but the extent for this is decided later, at the assignment (and in reality the assignment is a pointer-association) which is done with the symbols =>. Please note that the pointer assignment does not mean that the data in the matrix `B` is copied over to the matrix `A` (which would have taken relatively large resources), but it is merely a new address that is generated. To "move" data with the pointer concept will therefore be very efficient. As an alternative the pointer can become associated with the statement `ALLOCATE`, and be disassociated with `DEALLOCATE`, as in the following example.

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

There is also an internal function `ASSOCIATED` in order to investigate if a pointer is associated (and if it is associated with a certain target) and a statement `NULLIFY` in order to terminate the association.

```
IF ( ASSOCIATED (A) )  WRITE(*,*) ' A is associated '
IF ( ASSOCIATED (A, B) )  WRITE(*,*) ' A is associated with B'
NULLIFY  (A)
```

Please remember that a pointer in Fortran 90 has both type and rank, and that these must agree with the corresponding target. This increases the security at the use of pointers, it is therefore not possible by mistake to let a pointer change values of variables of other (different) data types. The fact that you have to specify that a variable can be a target also increases both security and efficiency of the compilation.

Important application of pointers are lists and trees, and especially dynamic arrays.

12.2 Simple use of pointers.

You have to be careful when you use pointers. In the following simple example we look at ordinary scalar floating-point numbers.

```
       REAL, TARGET   :: A
       REAL, POINTER  :: P, Q
       A = 3.1416
       P => A
       Q => P
       A = 2.718
       WRITE(*,*) Q
```

Here the value of `Q` equals 2.718 since both `P` and `Q` point towards the same variable `A` and that one has just changed its value from 3.1416 to 2.718. We now make a simple variation.

```
       REAL, TARGET  :: A, B
       REAL, POINTER :: P, Q
       A = 3.1416
       B = 2.718
       P => A
       Q => B
```

Now both the values of `A` and `P` are equal to 3.1416 and the values of both `B` and `Q` are 2.718. If we now give the statement

```
       Q = P
```

all four variables will get the value 3.1416, which means that an ordinary assignment of pointer variables has the same effect as the conventional assignment

```
       B = A
```

If we instead give a pointer association

```
       Q => P
```

then the three variables `A, P` and `Q` all have the value 3.1416, while `B` contains the value 2.718. In the second case `Q` only points to the same variable as `P` while in the first case `Q` becomes the same as `P`, and the value addressed by `Q` becomes equal to the value addressed by `P`.

12.3 Pointers and arrays.

A simple use of pointers is to give a name to an array section.

```
       REAL, TARGET   :: B(10,10)
       REAL, POINTER  :: A(:), C(:)
        A => B(4,:)   ! vector A becomes the fourth row
        C => B(:,4)   ! and vector C becomes the fourth
                      ! column of the matrix B
```

It is not necessary to take the whole section, you can take only a partial section. In the following example you can take a partial matrix `WINDOW` of a large matrix `MATRIX`.

```
       REAL, TARGET          :: MATRIX(100,100)
       REAL, POINTER         :: WINDOW(:,:)
       INTEGER               :: N1, N2, M1, M2
       WINDOW => MATRIX(N1:M1, N2:M2)
```

If you later wish to change a dimension of the partial matrix `WINDOW` you only need to make a new pointer association. Please note that the indices in `WINDOW` are not from `N1` to `M1` and from `N2` to `M2` but from `1` to `M1-N1+1` and from `1` to `M2-N2+1`.

There does not exist arrays of pointers directly in Fortran 90, but you can construct such facilities by creating a new data type. An example is to store a lower (or left) triangular matrix with rows with varying length. First introduce a new data type `ROW`

```
       TYPE ROW
              REAL, POINTER   :: R(:)
```

```
        END TYPE
```

and then specify the two lower triangular matrices `V` and `L` as vectors of rows with varying length

```
        INTEGER                 :: N
        TYPE(ROW)               :: V(N), L(N)
```

after which you can allocate the matrix `V` as below (and in the corresponding way you can allocate the matrix `L`)

```
        DO I = 1, N
               ALLOCATE (V(I)%R(1:I))
               ! Various length of rows
        END DO
```

The statement

```
        V = L
```

then becomes equivalent with

```
        V(I)%R => L(I)%R
```

for all the components, i.e. all values of `I`. Please note that in this application there is no `TARGET` required.

12.4 Allocation of arrays using pointers.

One implementation of dynamic memory allocation is to use pointers to specify an array. In the following example we specify a vector in such a way, that it can be given its size (its extent) in a subroutine, but can be used in the main program. It is the only way we have found to move an actual dimension upwards.

An alternative method has however been suggested by *Arie ten Cate*, using a module with an `ALLOCATE`d and `SAVE`d array. An example is available.

We however use an `INTERFACE` with pointers in the main program and allocate, also using pointers, a vector in the subroutine. In this way we get a dynamically allocated vector.

```
      PROGRAM MAIN_PROGRAM
      INTERFACE
         SUBROUTINE SUB(B)
         REAL, DIMENSION (:), POINTER :: B
         END SUBROUTINE SUB
      END INTERFACE
      REAL, DIMENSION (:), POINTER :: A
      CALL SUB(A)
```

```
!     Now we can use the vector A.
!     Its dimension was determined in the subroutine,
!     the number of elements is available as SIZE(A).
      END PROGRAM MAIN_PROGRAM

      SUBROUTINE SUB(B)
      REAL, DIMENSION (:), POINTER :: B
      INTEGER M
!     Now we can assign a value to M, for example
!     through an input statement.
!     When M has been assigned we can allocate B
!     as a vector.
      ALLOCATE (B(M))
!     Now we can use the vector B.
      END SUBROUTINE SUB
```

Note: The method above is even more useful for allocating matrices, see exercise 12.3.

Exercises

(12.1) Use pointers in order to assign all even elements of a vector the value 13 and all odd elements of a vector the value 17.

(12.2) Specify two pointers, and let one of them point to a whole vector and the other one point to the seventh element of the same vector.

(12.3) Use pointers to specify a matrix in such a way, that it is given its size (its extent) in a subroutine but can be used in the main program.

13. THE NEW PRECISION CONCEPT

The problem with the older versions of Fortran was that simple precision on one computer could correspond to a higher precision or `DOUBLE` precision on another computer and that the data type `DOUBLE PRECISION COMPLEX` or `COMPLEX*16` was not available in all systems (and of course not in the standard).

In Fortran 90 there are standard functions to check the precision of variables (see Appendix 5, section 8, where for example `PRECISION(X)` gives the number of significant digits in numbers of the same kind as the variable `X`). In Fortran 90 there are also possibilities to specify for each variable how many significant digits can be used with the floating-point numbers of this kind. The two common precisions single precision (SP) and double precision (DP) on a system based on IEEE 754 can specified with

```
  INTEGER, PARAMETER     :: SP = SELECTED_ REAL_ KIND(6,37)
  INTEGER, PARAMETER     :: DP = SELECTED_REAL_KIND(15,307)

  REAL(KIND=SP)          ::  single_precision_variables
  REAL(KIND=DP)          ::  double_precision_variables
```

If we wish to work with at least 14 decimal digits accuracy and at least decimal exponents between - 300 and + 300 we can choose the following integer parameters

```
  INTEGER, PARAMETER     :: WP = SELECTED_REAL_KIND(14,300)

  REAL(KIND=WP)          :: working_precision_variables
```

Regrettably now we have to give all floating point constants with the additional suffix `_WP`, for example

```
  REAL(KIND=WP)   :: PI
  PI = 3.141592653589793_WP
```

while since the intrinsic functions are generic, they will automatically use the correct data type and kind, which means that the argument determines which kind the result should have (usually the same as the argument).

With this method you will in practice obtain double precision on systems based on IEEE 754 and single precision on computers like Cray or computers based based on the Digital

Equipment Alpha-processor, which in all cases means a precision of about 15 significant digits.

14. ADDITIONAL PROBLEMS AT THE TRANSITION

- **Removal of automatic generation of new lines at input**

  A problem, that can arise when moving from Fortran 77 to Fortran 90, is dependent on a usual deviation from the standard, and has to do especially with user programs. What we consider here, is the use in the `FORMAT` of the dollar symbol $ in order to remove the generation of a new line (Line Feed/Carriage Return), before the user gives a new value to a variable. This is a common extension of Fortran 77, but it generates a compilation error in Fortran 90 for the dollar symbol and therefore another solution has to be found.

  With many Fortran 77 implementations we wrote

```
       PROGRAM TEST
       REAL X
       WRITE(*,10)
10     FORMAT('Give X = ',$)
       READ(*,*) X
       WRITE(*,*) X
       END
```

  In Fortran 90 we use "non-advancing I/O" instead. We therefore write the following

```
       PROGRAM TEST
       IMPLICIT NONE
       REAL X
       WRITE(*,'(A)',ADVANCE='NO') 'Give X = '
       READ(*,*) X
       WRITE(*,*) X
       END
```

  Both those programs give the same result, you may give the value of the variable on the same row as the text "`Give X = `". Non-advancing I/O can not be used with list-directed I/O or on `NAMELIST`.

- **Varying system for the treatment of matrices**

In Fortran 77 there is no dynamic memory allocation, you therefore have to give a sufficiently large dimension in the calling program and keep in mind the "leading dimension" in the called program unit. Now when you use Fortran 90 you prefer to have an array of the same shape and size as the logical size of a matrix. An assignment that performs this transformation is easy to achieve. We assume that a quadratic matrix from the calling program unit is available in the subprogram under consideration as the array `A` with the dimensions `A(IA, *)` and that we wish to move this to an array `B` with the dimension specification `B(N, N)`, where `N` at the same time is the mathematical dimension of the matrix. The following assignment statement gives what you want, provided `IA` is not less than `N`.

```
B = A(1:N, 1:N)
```

- **Differences in the use of logical variables**

The new standard contains many more words and is more explicit, while some of the compiler or rather the compiler writer is much greater. An example is comparison of a logical variables, which under Fortran 90 has to be done with `.EQV.` or `.NEQV.` while this in practice often was possible in Fortran 77 also with `.EQ.` and `.NE.` If you try to perform such a comparison in Fortran 90 with the new equality symbol `= =` you can get a confusing error message, complaining that `.EQ.` may not be used in this context. The reason is of course that `= =` is just an alternative spelling of `.EQ.`

- **Small things of importance**

It has been rather common to use the variable name `SUM` in order to store the temporary value at the summation in a `DO`-loop. This name is now less convenient to use, since `SUM` is also the name of an automatic summation, see Appendix 5, section 14 (on array functions). Other dangerous variable names can be `ALL, HUGE, INDEX, INT, KIND, MASK, SCALE, SIZE, TINY` and `TRIM`. If you use a name that is being used by the Fortran language, the normal effect is that the intrinsic function is no longer available.

In some old Fortran dialects the statement `TYPE` was used to write on a typewriter terminal and `PRINT` was used to write on the line printer. The concept `TYPE` now has a completely new meaning in Fortran 90, to declare user-defined data types.

15. USE OF PROGRAM LIBRARIES

The two main numerical libraries in Fortran 90 are

- NAG Numerical Libraries for Fortran 90 Users
- IMSL Fortran 90 MP Library

Both of these offer both a modified version of the Fortran 77 library, recompiled for Fortran 90, and a completely new library, using all the new facilities of Fortran 90.

15.1 Using Old Libraries

A problem with Fortran 90 is that so far not so many programs or program libraries are available in the language. We can therefore be forced to use "old" program libraries usually from Fortran 77. This can be done in two ways.

1. Recompile the library with a Fortran 90 compiler
2. Use the Fortran 77 library directly

Method 1.

A simple method is to recompile the Fortran 77 routines with the Fortran 90 compiler using fixed form. This should work since Fortran 77 is a pure subset of Fortran 90. The problem is that the library does not always exactly follow the standard. NAG recognizes specifically that although the data type `DOUBLE PRECISION COMPLEX` or `COMPLEX*16` is not in the standard, it is in many implementations of Fortran 77. A subprogram which uses this extension therefore has to be modified. Also possible assignments of binary, octal or hexadecimal constants is difficult since they are standardized only in Fortran 90. Also note that the very common notation `REAL*8` for `DOUBLE PRECISION` is not permitted in Fortran 90.

Method 2.

The second method is to link the Fortran 77 library directly with the Fortran 90 programs. The practical problem was previously that the NAG's Fortran 90 system f90 under version 1.1 did not include support for linkage of libraries, therefore you first had to compile with the Fortran compiler using `f90 -c` and then link with the C compiler using `cc`. NAG recommended the following commands

```
 f90 -c test.f

cc test.o -lnag -o a.out /usr/local/lib/f90/f90rt0.o\
        /usr/local/lib/f90/libf90.a -lI77 -lF77 – lm
```

which worked on Sun after that we had removed `-lI77` since we did not find this library on the system here in Linköping.

From version 1.2 it is much simpler because linkage support is now included and on the Sun you only give the following command

```
           f90 test.f -lnag - lF77
```

while on DEC ULTRIX (MIPS) you give the command

```
           f90 test.f lnag -lfor -lutil -li –lots
```

Thus a few more libraries have to be included on the DEC.

In addition to the problems mentioned above we can also note that Fortran allows routine names as arguments, which may be treated differently in the NAG's Fortran 90 compiler based on translation to C and in the existing Fortran 77 compiler. There is therefore a danger for linkage errors.

On the Sun using Sun OS 4.2.1 we also got completely wrong results when using library functions in single precision, since the C system on Sun converted all function calls into double precision. NAG has *solved* this problem in release 2.1.

The method can thus be used when neither complex variables in double precision nor routine names are used as arguments, and provided that you are not using single precision library functions on a system which incorrectly converts the calls to double precision.

It is very essential to get all the required libraries at the linkage process, those are for example the mathematical libraries in Fortran 77, Fortran 90 and C. Sometimes the libraries have to be given in the correct order.

16. PECULIARITIES IN FORTRAN 90

- Using the free source form it is not permitted to have comments starting with a `C` or a `*` in column 1. This would be a violation of the free format. The new character ! (exclamation mark) has to be used.
- An advantage compared with earlier Fortran standards is that the standard now requires that the compiler can signal if the user deviates from the permitted standard.

  It is required that a Fortran 90 compiler can signal

  - use of syntax not defined in the standard.
  - violation of the syntax rules.
  - use of kinds not available.
  - use of obsolete constructs (or statements).
  - use of non-Fortran characters (for example, Swedish or Cyrillic characters) outside of character strings or comments.
  - violation of the area of validity for variable names, names of the `DO`-loops and the corresponding names like `IF, CASE` and operators.
  - the reason that a program is not accepted by the compiler.

  The above means that it is permitted to include extensions to Fortran 90. It has to be possible to ask the program to signal for any extensions outside the Fortran 90 standard.

- The compiler often performs some data flow analysis.

17. FORTRAN 95

Fortran 95 was published as ISO/IEC 1539-1:1997 on December 15, 1997.

The preliminary revision has been published as "Special Issue, Fortran 95, Committee Draft, May 1995" in the **Fortran Forum**, Vol. 12, No. 2, June 1995. The draft is available on the net, see my Fortran page.

Two important issues, **exception handling**, especially for floating point, and **interoperability between languages** (mixed language programming) are not addressed in Fortran 95. Exception handling is discussed in some papers by John Reid, see the proceedings of the Kyoto Workshop on *Current Directions in Numerical Software and High Performance Computing.*

Fortran 2003 was published as ISO/IEC 1539-1:2004 on November 18, 2004.

Work on the next version of Fortran is going well, see the official home of Fortran Standards.

17.1 New features

1. The statement `FORALL` as an alternative to the `DO`-statement
2. Partial nesting of `FORALL` and `WHERE` statements
3. Masked `ELSEWHERE`
4. Pure procedures
5. Elemental procedures
6. Pure procedures in specification expressions
7. Revised `MINLOC` and `MAXLOC`
8. Extensions to `CEILING` and `FLOOR` with the `KIND` keyword argument
9. Pointer initialization
10. Default initialization of derived type objects
11. Increased compatibility with IEEE arithmetic
12. A `CPU_TIME` intrinsic subroutine
13. A function `NULL` to nullify a pointer

14. Automatic deallocation of allocatable arrays at exit of scoping unit
15. Comments in `NAMELIST` at input
16. Minimal field at input
17. Complete version of `END INTERFACE`

Pure functions are functions without side effects, and elemental functions are pure functions with only scalar arguments and with scalar result.

17.2 Deleted features

Five features are deleted from Fortran 90.

1. real and double precision `DO` loop index variables
2. branching to `END IF` from an outer block
3. `PAUSE` statements
4. `ASSIGN` statements and assigned `GO TO` statements and the use of an assigned integer as a `FORMAT` specification
5. Hollerith editing in `FORMAT`

These are all from the list of obsolescent features of Fortran 90 (but not the complete list).

It is permitted for a compiler to have extensions to the standard, provided that these *can* be flagged by the system. It is very common for compilers to include the two features (see Appendix 4) that were removed from Fortran when Fortran 77 was introduced, and it is expected that this tradition will continue.

17.3 Obsolescent features

In the obsolescence list below those items that were not in the corresponding list of Fortran 90 are indicated NEW . All items in the list below are being considered for removal at the next revision, and should therefore be avoided.

The most probable candidates, in our opinion, for removal at the next revision are the first six.

1. Arithmetic `IF`-statement
2. Terminating several `DO`-loops on the same statement or terminating the `DO`-loop in some other way than with `CONTINUE` or `END DO`
3. Alternate return
4. Computed `GO TO` statement NEW
5. Statement functions NEW
6. `DATA` statements *among executable statements* NEW
7. Assumed character length functions NEW
8. Fixed form source code NEW
9. `CHARACTER*` form of `CHARACTER` declaration NEW

To replace `CHARACTER*LENGTH` with `CHARACTER(LEN=LENGTH)` or `CHARACTER(LENGTH)` is simple, as are the other items above. For example, item 6 is just to move all `DATA` statements to the top of the program unit, before the executable statements. Statement functions are of course replaced with internal functions.

17.4 Description of the new features

An explanation of the new features follows.

- **The statement FORALL**

  The statement `FORALL` is introduced as an alternative to the `DO`-statement. The main difference is that while the execution order of the various parts of the `DO`-loop is very strict, the execution order of the `FORALL` is less strict, thus permitting parallel execution. For further information we refer to our HPF Appendix or directly to the HPFF home page.

- **Partial nesting of FORALL and WHERE statements**

  A `FORALL` statement can include a `WHERE` statement.

- **Masked ELSEWHERE**

  It is now permitted to mask not only the `WHERE` statement of the `WHERE` construct, but also its `ELSEWHERE`, which now may be repeated.

```
WHERE (condition_1)
...
ELSEWHERE (condition_2)
...
ELSEWHERE
...
END WHERE
```

- **Pure procedures**

  Pure functions are functions without side effects. That a function has no side effects can be indicated with the new `PURE` prefix.

  A long list of constraints is given in the standard proposal, section 12.6.

  Pure subroutines are defined in a similar way. The only major difference is that "side effects" are of course permitted for arguments associated with dummy arguments specified `INTENT(OUT)` or `INTENT(INOUT)`.

  The advantage with knowing that a function is pure is that this fact simplifies parallel execution.

- **Elemental procedures**

  Elemental functions are pure functions with only scalar dummy arguments (not pointers or procedures) and with scalar result (not pointer). That a function is elemental can be indicated with the new `ELEMENTAL` prefix. The `RECURSIVE` prefix my not be combined with the `ELEMENTAL` prefix. The `PURE` prefix is automatically implied by the `ELEMENTAL` prefix.

  An elemental function may be used with arrays as actual arguments. These must then be conformable. The result is the same array as if the function had been used individually with each of the array elements, in any order.

  Elemental subroutines are defined in a similar way. The only major difference is that "side effects" are of course permitted for arguments associated with dummy arguments specified `INTENT(OUT)` or `INTENT(INOUT)`.

The advantage with knowing that a function is elemental is that this fact simplifies parallel execution, even more than if it is only pure.

- **Pure procedures in specification expressions**

  Pure functions can be used in specification expressions if certain conditions are fulfilled, see the standard proposal, section 7.1.6.2.

  Specification expressions can be used to specify array bounds and character lengths of data objects in a subprogram.

- **Revised MINLOC and MAXLOC**

  The array location functions `MINLOC` and `MAXLOC` are extended with the optional argument `DIM` corresponding to those for the array functions `MINVAL` and `MAXVAL`.

- **Extensions to CEILING and FLOOR**

  The new numerical functions `CEILING` and `FLOOR` are extended with the `KIND` keyword argument, in the same way as for `INT` and `NINT`. The result is an integer, but of the specified `KIND` or subtype, not necessarily the standard (default) integer subtype.

- **Pointer initialization**

  The new function `NULL` can be used at specification to define a pointer to be initially disassociated, see below.

- **Default initialization of derived type objects**

  Means are now available to specify default initial values for derived type components. It is the usual way of using the equal sign (or pointer assignment) followed by the value (perhaps an array constructor). Initialization does not have to apply to all components of a certain derived type.

  A simple example. In Appendix 3, section 12, we introduced a sparse matrix.

A numerically interesting example is a sparse matrix `A` with at most one hundred non-zero elements, which can be specified with the following statement, where now initialization is done to 2.0 for all elements.

```
TYPE NONZERO
       REAL :: VALUE = 2.0
       INTEGER :: ROW, COLUMN
END TYPE
```

and

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

You then get the value (which will be 2.0) of `A(10)` by writing `A(10)%VALUE`. The default value can be individually changed with an assignment, for example

```
A(15) = NONZERO(17.0,3,7)
```

- **Increased compatibility with IEEE arithmetic**

  The IEEE arithmetic has for floating point numbers one bit pattern for *plus zero* and another one for *minus zero*. Processors that distinguish between them shall treat them as identical

  - In all relational operations
  - As input arguments to all intrinsics except `SIGN`
  - As the scalar expression in the arithmetic `IF`-statement

  In order to distinguish between the two cases, the function `SIGN` has to be used. It is generalized so that the sign of the second argument is considered also if the value is a floating-point zero.

- **A CPU_TIME intrinsic subroutine**

  The subroutine `CPU_TIME(TIME)` belongs of course to intrinsic subroutines. In the scalar real variable `TIME` the present processor time is returned in seconds. If the processor is unable to provide a timing, a negative value is returned instead. As usual, the time for a certain computation is obtained by subtracting two different calls to the timing routine.

  The exact nature of the timing is implementation dependent, a parallel processor might reurn an array of times corresponding to the various processors. The difference between CPU-time and system time is also implementation dependent.

- **A function NULL to nullify a pointer**

  This function can be used at specification to define a pointer to be initially disassociated, in the example below the array `VECTOR`.

  ```
    REAL, POINTER, DIMENSION(:) :: VECTOR => NULL()
  ```
  The argument is not necessary, if present it determines the characteristics of the pointer, if not present, the characteristics are determined from context.

  The function belongs to section 20, pointer inquiry functions, which is now renamed "Pointer association status functions".

- **Automatic deallocation of allocatable arrays at exit of scoping unit**
  If the user has not explicitly deallocated local allocatable arrays at the exit of the scoping unit, this deallocation is now performed automatically, thus decreasing the memory required.

- **Comments in NAMELIST at input**

  It is now permitted to use comments in the usual way with `!`

- **Minimal field at output**

  In order to obtain an optimized use of the available positions it is now possible to only give the number of decimals, if any, and not the total field width, at the `FORMAT`s `B, F, I, O`, and `Z`. Examples are `I0` and `F0.6`. The result is of course not a field with zero digits, but with a suitable number of digits.

- **Complete version of END INTERFACE**

  Also `END INTERFACE` can now be given in a complete variant, so the second interface in chapter 10 can be given either in the old version as

```
INTERFACE SWAP
         MODULE PROCEDURE SWAP_R, SWAP_I, SWAP_C
  END INTERFACE
```

  or in the new preferred way as

```
  INTERFACE SWAP
         MODULE PROCEDURE SWAP_R, SWAP_I, SWAP_C
  END INTERFACE SWAP
```

This can also include an optional generic specification.

17.5 Different Fortran standards

Properties	Fortran 66	Fortran 77	Fortran 90		Fortran 95	
	Fixed form	Fixed form	Fixed	Free	(Fixed)	Free
Whole Fortran 66	=	-2	-2	-2	-5	-5
Whole Fortran 77	-	=	=	=	-5	-5
Whole Fortran 90	-	-	=	=	-5	-5
Whole Fortran 95	-	-	-14	-14	=	=
Continuation line indicated in column 6 on the next line	+	+	+	-	+	-
Continuation line indicated with & at the end of the present line	-	-	-	+	-	+
Blank line as comment	-	+	+	+	+	+
Significant blanks	-	-	-	+	-	+
Generic functions	-	+	+	+	+	+
User-defined generic functions	-	-	+	+	+	+
REAL*8	-	-	-	-	-	-
Comment symbol	C	C *	C * !	!	C * !	!
Extension in Unix	.f	.f	.f	.f90	.f	.f90
Extension in DOS	.FOR	.FOR	.FOR	.F90	.FOR	.F90

Above, for example, **-2** in the position *Whole Fortran 66 / Fortran 90* means that two properties have disappeared at the transition from Fortran 66 to Fortran 90. This applies to both fixed form and free form source code.

In the first four lines an equality sign = indicates that nothing has changed, a minus - that several properties are missing.

In the following seven lines a plus + indicates that the property is present, a minus - that it is absent.

Fixed form of the source code is discouraged in Fortran 95!

`REAL*8` is an alternative to `DOUBLE PRECISION`, introduced by IBM and used also by Digital. Several variants exist.

As comment symbol the exclamation mark ! is recommended, it is valid also in several implementations of Fortran 77.

In UNIX there is no concept called extension, and the extensions are not specified in the Fortran standard, but in informal manufacturer standards.

18. SUMMARY OF NEW FEATURES

Summary of the new features in Fortran 90

Below follows a short summary of the new standard. Please note that only some more essential parts and not the whole new standard are discussed here.

Source code form:

As an alternative to the old source-code (punched card oriented) form, there is a free form, which does not take any consideration to columns and where blanks are significant. Comments can be given on the line if preceded by an exclamation mark !, several statements can be given on the same line if using the semicolon ; as a separator, and statements that continue on the next line are continued on the present line (not the next line) with an ampersand &.

The underline symbol _ is permitted inside the names of variables, and the length of variables must have at most 31 characters (instead of at most 6 in the Fortran 77 standard).

Blanks become significant in the free form of the source code. Old commands like `ENDIF` and `GOTO` can also in the future be written either as `END IF` or `GO TO` respectively, but of course not like `EN DIF` or `GOT O`. To permit significant blanks in the old (fixed) form of the source code would not be possible, since it for example is permitted to write `END` in the following silly way

```
                  E              N              D
```

`INCLUDE` can be used to include source code from an external file. The construct is a line where there is `INCLUDE` and a character string and, perhaps, some concluding comments. Interpretation is implementation-dependent, normally the character string is treated as the name of the file that holds the source code that should be included. Nesting is permitted (and the number of levels is implementation-dependent), but recursion is not permitted for the `INCLUDE` statement.

As in most Algol-type languages the word `END` can be complemented with the name of the routine or function like `END FUNCTION GAMMA`.

Alternative representations:

The special characters < and > can now be used

```
    <         .LT.                    >        .GT.
    <=        .LE.                    >=       .GE.
    ==        .EQ.                    /=       .NE.
```

Specifications:

These can now be written on one line

```
      REAL, DIMENSION (3), PARAMETER :: &
      a = (/ 0.0, 0.0, 0.0 /), b = (/ 1.0, 1.0, 1.0 /)

      COMPLEX, DIMENSION(10) :: john
```

while the variables a and b become constant vectors with 3 elements and the floating-point values 0.0 and 1.0, respectively, while john becomes a complex vector with 10 complex elements, not yet assigned any values.

If you wish to use the Algol principle to specify all variables, this is simplified by the command `IMPLICIT NONE` which switches off the implicit-type rules.

Double precision has been implemented with a more general method to give the desired precision, namely the parameter `KIND` for which precision we wish, useful on all variable types.

```
      INTEGER, PARAMETER :: LP = SELECTED_REAL_KIND(20)
      REAL (KIND = LP)   :: X, Y, Z
```

The above two statements thus declare the variables `X, Y` and `Z` to be `REAL` floating-point variables with at least 20 decimal digits accuracy with a data type that is called `LP` (where `LP` stands for `LONG PRECISION`).

Conditional statement:

The new command is

```
SELECT CASE (expression)
CASE block-switch
     block
CASE block-switch
     block
CASE DEFAULT
     default block
END SELECT
```

Typical construct:

```
SELECT CASE(3*I-J)   !  the control variable is 3*i-j
       CASE(0)       !  for the value zero
       :             !  you execute the code here
       CASE(2,4:7)   !  for the variables 2, 4, 5, 6, 7
       :             !  you execute the code here
       CASE DEFAULT  !  and for all other values
                     !  you execute the code here
       :             !
END SELECT
```

If the `CASE DEFAULT` is missing and none of the alternatives is valid, the execution continues directly with the next statement following the `END SELECT`, without any error message.

Another example:

```
   INTEGER FUNCTION SIGNUM(N)
   SELECT CASE (N)
     CASE (:-1)
       SIGNUM = -1
     CASE (0)
       SIGNUM = 0
     CASE (1:)
       SIGNUM = 1
     END SELECT
   END
```

DO-loop:

Three new constructs are included. The first one gives in principle an infinite loop, which however can be terminated with a conditional `GOTO`-statement.

```
name:   DO
        executable statements
        END DO name
```

The usual `DO`-loop has the following new simplified form without statement number,

```
name:   DO i = integer_expr_1, integer_expr_2 ,integer_expr_3
        executable statements
        END DO name
```

where `i` is called control variable, and where `,integer_expr_3` is optional. Finally there is also the `DO WHILE` loop

```
name:   DO WHILE (logical_expression)
        executable statements
        END DO name
```

The name is optional but can be used for nested loops in order to indicate which one that is to be iterated once again with the `CYCLE` statement or terminated with the `EXIT` statement.

```
S1:  DO
             IF (X > Y ) THEN
                   Z = X
                   EXIT S1
             END IF
             CALL NEW(X)
     END DO

             N = 0
     LOOP1: DO I = 1, 10
               J= I
     LOOP2:       DO K =1, 5
                        L = K
                        N = N +1
                  END DO LOOP2
            END DO LOOP1
```

In the latter case the final values from the variables will be as follows, in full accordance with the standard, `I = 11, J = 10, K = 6, L = 5`, and `N = 50.`

To name the loop is completely optional. Also note that this type of name is limited to `DO`-loop, `CASE` or `IF...THEN...ELSE...ENDIF` constructs. The old possibilities with statement numbers are still available, also in the free form.

Program units:

Routines can be called with keyword arguments and can use default arguments

```
       SUBROUTINE solve (a, b, n)
       REAL, OPTIONAL, INTENT (IN)  :: b
```

can be called with

```
       CALL solve (n = i, a = x)
```

where two of the arguments are given with keywords instead of position and where the third one has a default value. If `SOLVE` is an external routine it requires making use of an `INTERFACE` block in the calling program. Routines can be specified to be recursive.

```
   RECURSIVE FUNCTION factorial (n) RESULT (fac)
```

but must then have a special `RESULT` name in order to return the result.

Character string variables:

`CHARACTER` has been expanded to include also an empty string

```
        a = ''
```

and assignment of an overlapping string is now permitted

```
        a(:5) = a(3:7)
```

The new intrinsic function `TRIM` which removes concluding blanks is an important addition. You can now make a free choice between the apostrophe ' and the quotation mark " in order to indicate a character string. This can among other things be used in such a way that if you wish to write an apostrophe inside the text, then you use the quotation mark as indicators, and in the opposite case if you wish to have a quotation mark inside the text you use the apostrophe as the indicator.

Input:

At last the `NAMELIST` is included in the standard! This statement, however, has to be among the specifications. In the example below `list2` is the name of the list, `a` and `b` are real variables and i is an integer variable.

```
NAMELIST /list2 / a, i, x
:
READ (unit, NML = list2)
```

which wishes to get input data of the following form, but all variables do not have to given, and they can be given in any order.

```
&list2 X = 4.3, A = 1.E20, I = -4 /
```

Vector and matrix management:

This is one of the most important parts of the new standard. An array is defined to have a shape given by its number of dimensions, called "rank", and the extent for each one of these. Two arrays agree if they have the same shape. Operations are normally done element by element. Please remember that the rank for an array is the number of dimensions and has nothing at all to do with the mathematical rank of a matrix!

```
REAL, DIMENSION(5,20)     :: x, y
REAL, DIMENSION(-2:2,20)  :: z
:
z = 4.0*y*sgrt(x)
```

We perhaps here wish to protect against negative elements of X. This is done with the following construct

```
WHERE ( x >= 0.0 )
       z = 4.0*y*sgrt(x)
ELSEWHERE
       z = 0.0
END WHERE
```

Please note that ELSEWHERE has to be in one word! Compare also with the function SUM which is discussed at the end of the next section.

You can pick out a part of an array. Assume that the array A is specified in the following way.

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

With A(-3, :) you pick the second row, while with A(0:-4:-2, 1:7:2) you pick (in reverse order) its each other element in each other column. Just as variables can form arrays, also constants can form arrays.

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

where the first argument to the intrinsic function RESHAPE gives the value and the second argument gives the new shape. Two additional, but optional, arguments are available to this function.

The above can also be written in a more compressed form using the PARAMETER attribute. In the first line below the PARAMETER attribute is compulsory (if the assignment is to be made on the same line), but in the second line it is optional. Remember that the PARAMETER attribute means that the quantity can not be changed during execution of the program.

```
REAL, DIMENSION(6), PARAMETER :: B = (/ 11, 12, 13, 14, 15, 16 /)
REAL, DIMENSION(2,3), PARAMETER :: C = RESHAPE( B, (/ 2, 3 /) )
```

Any statements for real parallel computation are not included in Fortran 90. The committee believes it is necessary with additional experience before the standardization of parallelization. See also HPF discussed in the Appendix 8.

Dynamic storage:

Fortran 90 contains four different ways to make dynamical access. The first one is to use a pointer. See an example on a vector and for an example on a matrix see exercise 12.3

The second is to use an "allocatable array", i.e. with the statements `ALLOCATE` and `DEALLOCATE` you get and return a storage area for an array with type, rank and name (and possible other attributes) which had been specified earlier with the additional attribute `ALLOCATABLE`.

```
REAL, DIMENSION(:), ALLOCATABLE  :: x
:
Allocate(x(N:M))  ! N and M are the integer expressions here.
:
x(j) = q          ! Some assignment of the array.
CALL sub(x)       ! Use of the array in a subroutine.
:
DEALLOCATE (x)
```

Deallocation occurs automatically (if the attribute `SAVE` has not been given) when you reach `RETURN` or `END` in the same program unit.

The third variant is an "automatic array", it is almost available in the old Fortran, where x in the example below has to be in the list of arguments. This is not required any more.

```
SUBROUTINE sub (i, j, k)
REAL, DIMENSION (i, j, k)   :: x
```

Dimensions for x are taken from the integers in the calling program. Finally there is an "assumed-shape array" where the storage is defined in the calling procedure and for which only the type, rank and name are given.

```
SUBROUTINE sub(a)
REAL, DIMENSION (:,:,:)  :: a
```

According to Metcalf and Reid (1990, 1992), section 6.3 you here require an explicit interface. This has to look as follows

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

If you forget the `INTERFACE` or if you have an erroneous interface, then you will usually get "segmentation error", it means that a program unit may be missing.

Some intrinsic functions are available to determine the actual dimension limits

```
     DO (i = LBOUND(a,1), UBOUND(a,1))
             DO (j = LBOUND (a,2), UBOUND (a,2))
                     DO (k = LBOUND(a,3),UBOUND (a,3))
```

where `LBOUND` gives the lower limit for the specified dimension and `UBOUND` gives the upper one.

The sum of the positive value of a number of elements in an array is written

```
     SUM ( X, MASK = X .GT. 0.0)
```

These statements can not be used in order to avoid division by zero at for example summation of `1/X`, that is the mask works only with determining which numbers that are to be included in the summation, and not whether a certain value has to be calculated or not. But in this later case you can use the construct `WHERE`, see section 9.

Intrinsic functions:

Fortran 90 defines about 100 intrinsic functions and a few intrinsic subroutines. Many of these functions can be used for arrays, e.g. for reduction (`SUM`), construct (`SPREAD`), manipulation (`TRANSPOSE`). Other functions permit attributes or available parameters in the programming environment to be determined, as the largest positive floating-point number and the largest positive integer, as well as access to the system clock. The random numbers generator is included. Finally, the function `TRANSFER` permits a certain physical area to be transmitted to another area without type conversion.

The function `SPREAD` is discussed more fully in the solution of exercise (11.1).

All intrinsic functions and subroutines are discussed in Appendix 5.

Use of the user-defined data type:

Fortran had earlier not permitted use of any user-defined type. This has now become possible.

```
TYPE staff_member
       CHARACTER(LEN=20) :: first_name, last_name
       INTEGER           :: identification, department
END TYPE
```

which can be used in order to describe an individual. A combination of individuals can also be formed

```
TYPE(staff_member), DIMENSION(100)  :: staff
```

Individuals can be referred to as staff(number) and a field can be referred as `staff(number)%first_name`. You can also nest definitions

```
TYPE company
     CHARACTER(LEN=20)                    :: company_name
     TYPE(staff_member), DIMENSION(100)   :: staff
END TYPE
:
TYPE(company), DIMENSION(10)   :: several_companies
```

A numerically more interesting example is a sparse matrix `A` with at most one hundred non-zero elements, which can be specified with the following statement

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

and

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

You then get the value of `A(10)` by writing `A(10)%VALUE`. Assignment can be done, for example with

```
        A(15) = NONZERO(17.0,3,7)
```

In order to use user-defined data types in for example `COMMON`, or to make sure that two data types which look the same are treated as identical, you can use the `SEQUENCE` statement, in the latter case it is also required that no variable is specified `PRIVATE`.

Modules:

Modules are collections of data, type definitions and procedure definitions, which give a more secure and general replacement for the `COMMON` concept.

The data type "bit":

The data type "bit" is not included in the standard, but there are available various bit operations of integers according to an earlier military standard MIL-STD 1753. In addition, binary, octal and hexadecimal constants are included, as well as the possibility to use these quantities in input/output through the three new format-letters. In a `DATA` statement you can use an assignment to

```
     B'01010101010101010101010101010101'
```

for binary,

```
     O'01234567'
```

for octal, and

```
     Z'ABCDEF'
```

for hexadecimal numbers.

Pointers:

Pointers have been included, but not in the usual way as a new data type but as an attribute to the other data types. A variable with a pointer attribute can be used as an ordinary variable and in a number of new ways. Pointers in Fortran 90 are not memory addresses as in many languages or in certain Fortran variants (dialects), but are more like extra names (aliases).

Pointers are discussed in chapter 12.

User extensions:

The new language contains a possibility for the user to extend it with his own concepts, for example interval arithmetics, rational arithmetics or dynamic character strings. By defining a new data type or operator, and overloading operations and procedures (so that you can also use the plus + as the symbol of addition of intervals and not only of ordinary numbers), we can create a package (a module) without using a preprocessor. We can soon expect a number of extensions for different applications in the form of modules from different manufacturers. Some are already available from NAG.

19. BACKWARD AND FORWARD COMPATIBILITY

Very important in the introduction of a new programming language standard is that old programs (at least those who obey the outgoing standard) can be used once again, with the new standard.

19.1 Backward

When we went from Fortran 66 to Fortran 77 the extended `DO`-loop was removed (the extended `DO`-loop means that if you do not change any of the `DO`-loop parameters you can jump out of the loop and then jump in again (this is somewhat contrary to the concept of structured programming). In addition Hollerith constants were removed (except in `FORMAT`). That means that there are some programs that obey Fortran 66 but do not obey Fortran 77. Most manufacturers have, however, chosen to let these two concepts be included as extensions in their Fortran implementations. For Fortran 90 nothing has been removed from Fortran 77. An interesting practical question is however manufacturers still continue to include those old things that really should have been thrown away when Fortran 77 came. It is permitted to have these old concepts as extensions.

There is one further incompatibilty between Fortran 66 and Fortran 77, which is related to assumed-size allocation of dummy arrays.

On the other hand, the concept "obsolescence" is introduced. This means that some constructs may be removed at the next change of Fortran. These constructs are:

- Arithmetic `IF`-statement
- Control variables in a `DO`-loop which are floating point or double-precision floating-point
- Terminating several `DO`-loops on the same statement
- Terminating the `DO`-loop in some other way than with `CONTINUE` or `END DO`
- Alternate return
- Jump to `END IF` from an outer block
- `PAUSE`
- `ASSIGN` and assigned `GOTO` and assigned `FORMAT` , that is the whole "statement number variable" concept.
- Hollerith editing in `FORMAT`.

Further information on obsolescence is available in Status of Fortran 95, which describes the present suggestions for the next standard. See especially the new list of deleted features and the revised list of obsolescent features.

19.2 Parallel extensions

A group "High Performance Fortran Forum" has worked at the development of an extension to Fortran 90 with parallel extensions. The purpose of this project is to offer a portable language which can be used efficiently on different parallel systems. The project was ready, with a complete proposal, in May 1993 and aims at a de facto standard (and not at a formal standard). See Appendix 8 for a summary.

Somewhat simplified you can say that Fortran 90 works effectively on vector processors but not on parallel processors.

19.3 Forward

Among the new things that are being considered for the next version of Fortran are improved parallel treatment, interrupt handling, parametrized data types, and data types with inherited properties. It is the aim of the committee to have a slight revision available in 1996, with some carefully chosen new properties.

In addition, a few corrections have been accepted earlier, especially some explanations of ambitious parts of the standard. An early version of these corrections appeared in a special issue of the Fortran Forum, Vol. 11, No. 1, March 1993, with the title "Fortran 90; Errata, Amendments, and Interpretations, progress to date". Some updates to this special issue appeared in No. 2, June 1993, page 1.

Two revisions have been officially adopted, see the official ISO page on Fortran 90.

20. INTRINSIC FUNCTIONS IN FORTRAN 90

There is a large a number of intrinsic functions and five intrinsic subroutinesin Fortran 90. I treat the numeric and mathematical routines very shortly, since they are not changed from Fortran 77 and therefore should be well-known.

This section is based on section 13 of the ISO standard (1991), which contains a more formal treatment. We follow the arrangement of the different functions and subroutines in the standard, but explain directly in the list. For a more detailed treatment we refer to Metcalf and Reid (1990, 1993).

When a parameter below is optional it is given in lower case characters. When an argument list contains several arguments the function can be called either by position related arguments or by a keyword. Keyword must be used if some previous argument is not included. Keywords are normally the names that are given below.

We have not always given all the natural limitations to the variables, for example that the rank is not permitted to be negative.

Function which determines if a certain argument is in an actual argument list:

The function `PRESENT(A)` returns `.TRUE.` if the argument `A` is in the calling list, `.FALSE.` in the other case. The use is illustrated in the example program in chapter 8 of the main text.

Numerical functions:

The following are available from Fortran 77: `ABS, AIMAG, AINT, ANINT, CMPLX, CONJG, DBLE, DIM, DPROD, INT, MAX, MIN, MOD, NINT, REAL` and `SIGN`.

In addition, `CEILING, FLOOR` and `MODULO` have been added to Fortran 90. Only the last one is difficult to explain, which is most easily done with the examples from ISO (1991)

```
    MOD (8,5)    gives  3     MODULO (8,5)    gives  3
    MOD (-8,5)   gives -3     MODULO (-8,5)   gives  2
    MOD (8,-5)   gives  3     MODULO (8,-5)   gives -2
    MOD (-8,-5)  gives -3     MODULO (-8,-5)  gives -3
```

The following functions from Fortran 77 can use a kind-parameter like in `AINT(A, kind)`, namely `AINT, ANINT, CMPLX, INT, NINT` and `REAL`.

A historic fact is that the numerical functions in Fortran 66 had to have specific (different) names in different precisions, and these explicit names are still the only ones which can be used when a function name is passed as an argument.

A complete table of all the numerical functions follow. Those names that are indicated with a star * are not permitted to be used as arguments. Some functions, like `INT` and `IFIX` have two specific names, either can be used. On the other hand, some functions do not have any specific name. Below I use `C` for complex floating point values, `D` for floating point values in double precision, `I` for integers, and `R` for floating point values in single precision.

Function	**Generic**		**Specific**	**Data type**	
	name		**name**	**Arg**	**Res**
Conversion	INT		-	I	I
to integer		*	INT	R	I
		*	IFIX	R	I
		*	IDINT	D	I
(of the real part)			-	C	I
Conversion	REAL	*	REAL	I	R
to real		*	FLOAT	I	R
			-	R	R
		*	SNGL	D	R
(real part)			-	C	R
Conversion	DBLE		-	I	D
to double			-	R	D
precision			-	D	D
(real part)			-	C	D
Conversion	CMPLX		-	I (2I)	C
to complex			-	R (2R)	C
			-	D (2D)	C
			-	C	C
Truncation	AINT		AINT	R	R
			DINT	D	D

Rounding	ANINT	ANINT	R	R
		DNINT	D	D
	NINT	NINT	R	I
		IDNINT	D	I
Absolute value	ABS	IABS	I	I
		ABS	R	R
		DABS	D	D
		CABS	C	R
Remainder	MOD	MOD	2I	I
		AMOD	2R	R
		DMOD	2D	D
	MODULO	-	2I	I
		-	2R	R
		-	2D	D
Floor	FLOOR	-	I	I
		-	R	R
		-	D	D
Ceiling	CEILING	-	I	I
		-	R	R
		-	D	D
Transfer of sign	SIGN	ISIGN	2I	I
		SIGN	2R	R
		DSIGN	2D	D
Positive difference	DIM	IDIM	2I	I
		DIM	2R	R
		DDIM	2D	D
Inner product	-	DPROD	R	D
Maximum	MAX	* MAX0	I	I
		* AMAX1	R	R
		* DMAX1	D	D
	-	* AMAX0	I	R
	-	* MAX1	R	I
Minimum	MIN	* MIN0	I	I
		* AMIN1	R	R
		* DMIN1	D	D
	-	* AMIN0	I	R
	-	* MIN1	R	I

```
Imaginary part  -        AIMAG    C     R

Conjugate       -        CONJG    C     C
```

Truncation is towards zero, `INT(-3.7)` becomes `-3`, but rounding is correct, `NINT(-3.7)` becomes `-4`. The new functions `FLOOR` and `CEILING` truncate towards minus and plus infinity, respectively.

The function `CMPLX` can have one or two arguments, if two arguments are present these must be of the same type but not `COMPLEX`.

The function `MOD(X,Y)` calculates `X - INT(X/Y)*Y`.

The sign transfer function `SIGN(X,Y)` takes the sign of the second argument and puts it on the first argument, `ABS(X)` if `Y >= 0` and `-ABS(X)` if `Y < 0`.

Positive difference `DIM` is a function I have never used, but `DIM(X,Y)` gives `X-Y` if this is positive and zero in the other case.

Inner product `DPROD` on the other hand is a very useful function which gives the product of two numbers in single precision as a double precision number. It is both fast and accurate.

The two functions `MAX` and `MIN` are unique in that they may have an arbitrary number of arguments, but at least two. The arguments have to be of the same type, but are not permitted to be of type `COMPLEX`.

Mathematical functions:

Same as in Fortran 77. All trigonometric functions work in radians. The following are available: `ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP, LOG, LOG10, SIN, SINH, SQRT, TAN` and `TANH`.

A historic fact is that the mathematical functions in Fortran 66 had to have specific (different) names in different precisions, and these explicit names are still the only ones which can be used when a function name is passed as an argument.

A complete table of all the mathematical functions follow. Below I use `C` for complex floating point values, `D` for floating point values in double precision, `I` for integers, and `R` for floating point values in single precision.

Function	**Generic**	**Specific**	**Data type**	
	name	**name**	**Arg**	**Res**
Square root	SQRT	SQRT	R	R
		DSQRT	D	D
		CSQRT	C	C
Exponential	EXP	EXP	R	R
		DEXP	D	D
		CEXP	C	C
Natural logarithm	LOG	ALOG	R	R
		DLOG	D	D
		CLOG	C	C
Common logarithm	LOG10	ALOG10	R	R
		DLOG10	D	D
Sine	SIN	SIN	R	R
		DSIN	D	D
		CSIN	C	C
Cosine	COS	COS	R	R
		DCOS	D	D
		CCOS	C	C
Tangent	TAN	TAN	R	R
		DTAN	D	D
Arcsine	ASIN	ASIN	R	R
		DASIN	D	D
Arccosine	ACOS	ACOS	R	R
		DCOS	D	D
Arctangent	ATAN	ATAN	R	R
		DATAN	D	D
	ATAN2	ATAN2	2R	R
		DATAN2	2D	D
Hyperbolic	SINH	SINH	R	R

```
 sine                   DSINH    D     D

Hyperbolic      COSH     COSH     R     R
 cosine                 DCOSH    D     D

Hyperbolic       TANH    TANH     R     R
 tangent                DTANH    D     D
```

The purpose of most of these functions is obvious. Note that they are all only defined for floating point numbers, and not for integers. You can therefore not calculate the square root of 4 as `SQRT(4)`, but instead you can use `NINT(SQRT(REAL(4)))`. Please also note that all complex functions return the principal value.

The square root gives a real result for a real argument in single or double precision, and a complex result for a complex argument. So `SQRT(-1.0)` gives an error message (usually already at compile time), while you can get the complex square root using the following statements.

```
COMPLEX, PARAMETER    :: MINUS_ONE = -1.0
COMPLEX               :: Z
Z = SQRT(MINUS_ONE)
```

The argument for the usual logarithms has to be positive, while the argument for `CLOG` must be different from zero.

The modulus for the argument to `ASIN` and `ACOS` has to be at most 1. The result will be within [-pi/2, pi/2] and [0, pi], respectively.

The function `ATAN` will return a value in [-pi/2, pi/2].

The function `ATAN2(Y,X) = arctan(y,x)` will return a value in (-pi, pi]. If `Y` is positive the result will be positive. If `Y` is zero the result will be zero if `X` is positive, and pi if `X` is negative. If `Y` is negative the result will be negative. If `X` is zero the result will be plus or minus pi/2. Both `X` and `Y` are not permitted to be zero simultaneously. The purpose of the function is to avoid division by zero.

A natural limitation for the mathematical functions is the limited accuracy and range, which means that for example `EXP` can cause underflow or overflow at rather common values of the argument. The trigonometric functions will get very low accuracy for large arguments. These limitations are implementation dependent, and should be given in the vendor's manual.

Character string functions:

The functions below perform operations from and to character strings. Please note that `ACHAR` works with the standard ASCII character set while `CHAR` works with the representation in the computer you are using.

```
ACHAR(I)          Returns the ASCII character which has number I
ADJUSTL(STRING)   Adjusts to the left
ADJUSTR(STRING)   Adjusts to the right
CHAR(I, kind)     Returns the character that has the number I
IACHAR(C)         Returns the ASCII number of the character C
ICHAR(C)          Returns the number of character C

INDEX(STRING, SUBSTRING, back)  Returns the starting position for a
    substring within  a  string.  If BACK  is  true then you get the
    last starting position, in the  other case, the first one.

LEN_TRIM(STRING)  Returns the length of the string without the possibly
    trailing blanks.

       LGE(STRING_A, STRING_B)
       LGT(STRING-A, STRING_B)
       LLE(STRING_A, STRING_B)
       LLT(STRING_A, STRING_B)
```

The above routines compare two strings using sorting according to ASCII. If a string is shorter than the other, blanks are added at the end of the short string. If a string contains a character outside the ASCII character set, the result is implementation-dependent.

```
REPEAT(STRING, NCOPIES)    Concatenates a character string NCOPIES
                           times with itself.
SCAN(STRING, SET, back)    Returns the position of the first occurrence
                           of any character in the string SET in the
string
                           STRING. If BACK is true, you will get
                           the rightmost such character.
TRIM(STRING)               Returns the character string STRING without
                           trailing blanks.
VERIFY(STRING, SET, back)  Returns the position of the first character
                           in STRING which is not in SET.  If BACK
                           is TRUE, you get the last one!
                           The result is zero if all characters are
                           included!
```

Character string function for request:

`LEN(STRING)` returns the length of a character string. There does not have to be assigned a value to the variable `STRING`.

Kind functions:

```
KIND(X)
SELECTED_INT_KIND(R)
SELECTED_REAL_KIND(p, r)
```

The first returns the kind of the actual argument, which can be of the type `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` or `CHARACTER`. The argument `X` does not have to be assigned any value. The second returns an integer kind with the requested number of digits, and the third returns the kind for floating-point numbers with numerical precision at least `P` digits and one decimal exponent range between `-R` and `+R`. The parameters `P` and `R` must be scalar integers. At least one of `P` and `R` must be given.

The result of `SELECTED_INT_KIND` is an integer from zero and upward, if the desired kind is not available you will get -1. If several implemented types satisfy the condition, the one with the least decimal range is used. If there still are several types or kinds that satisfy the condition, the one with the smallest kind number will be used.

The result of `SELECTED_REAL_KIND` is also an integer from zero and upward; if the desired kind is not available, then -1 is returned if the precision is not available, -2 if the exponent range is not available and -3 if no one of the requirements are available. If several implemented types satisfy the condition, the one with the least decimal precision is returned, and if there are several of them, the one with the least kind number is returned.

Examples are given in chapter 2 of the main text. Examples of kinds in a few different implementations (NAG and Cray) are given in Appendix 6.

Logical function:

`LOGICAL(L, kind)` converts between different kinds of logical variables. Logical variables can be implemented in various ways, for example with a physical representation occupying one bit (not recommended), one byte, one word or perhaps even one double word. This difference is important if `COMMON` and `EQUIVALENCE` with logical variables have been misused in a program in the traditional way of Fortran 66 programming.

Numerical inquiry functions:

These functions work with a certain model of integer and floating-point arithmetics, see ISO (1991), section 13.7.1. The functions return properties of numbers of the same kind as the variable `X`, which can be real and in some cases integer. Functions that return properties of the actual argument `X` are available in section 12 below, floating-point manipulation functions.

```
DIGITS(X)       The number of significant digits
EPSILON(X)      The  least  positive  number  that added
                to 1 returns a number that is greater than 1
HUGE(X)         The largest positive number
MAXEXPONENT(X)  The largest exponent
MINEXPONENT     The smallest exponent
PRECISION(X)    The decimal precision
RADIX(X)        The base in the model
RANGE(X)        The decimal exponent
TINY(X)         The smallest positive number
```

Bit inquiry function:

`BIT_SIZE(I)` returns the number of bits according to the model of bit representation in the standard ISO (1991), section 13.5.7. Normally we get the number of bits in a (whole) word.

Bit manipulation functions:

The model for bit representation in the standard ISO (1991), section 13.5.7 is used.

```
BTEST(I, POS)         .TRUE. if the position number POS of I is 1
IAND(I, J)            logical  addition  of  the  bit characters in
                      variables I and J

IBCLR(I, POS)         puts a zero in the bit in position POS
IBITS(I, POS, LEN)    uses LEN bits of the word I with
```

```
                      beginning in position POS,  the additional bits
                      are set to zero.  It requires that
                      POS + LEN <= BIT_SIZE(I)
IBSET(I, POS)         puts the bit in position POS to 1
IEOR(I, J)            performs logical exclusive OR
IOR(I, J)             performs logical OR
ISHIFT(I, SHIFT)      performs logical shift (to the right if the
number
                      of steps SHIFT < 0 and to the left if SHIFT > 0).
                      Positions that are vacated are set to zero.
ISHIFTC(I, SHIFT, size) performs  logical  shift  a  number  of  steps
                      circularly to   the   right  if  SHIFT  <  0,
                      circularly to the left if SHIFT > 0.  If SIZE
                      is given, it is required that 0 < SIZE <=
                      BIT_SIZE(I).  Shift is only done for the bits
                      that are  in the SIZE rightmost positions, but
                      for all positions if SIZE is not given.
NOT(I)                returns a logical complement
```

Transfer functions:

`TRANSFER(SOURCE, MOULD, size)` specifies that the physical representation of the first argument `SOURCE` shall be treated as if it had type and parameters as the second argument `MOULD`, but without converting it. The purpose is to give a possibility to move a quantity of a certain type via a routine that does not have exactly that data type.

Floating-point manipulation functions:

These functions work in a certain model of integer and floating-point arithmetic, see the standard ISO(1991), section 13.7.1. The functions return numbers related to the actual variable `X` of the type `REAL`. Functions that return properties for the numbers of the same kind as the variable `X` are under section 8 (Numerical inquiry functions).

```
EXPONENT(X)         exponent of the number
FRACTION(X)         the fractional part of the number
NEAREST(X, S)       returns the next representable number in
                    the direction of the sign of S
RRSPACING(X)        returns the inverted value of the distance
                    between the two nearest possible numbers
SCALE(X, I)         multiplies X by the base to the power I
SET_EXPONENT(X, I)  returns the number that has the fractional
                    part of X and the exponent I
SPACING(X)          the distance between the two nearest
                    possible numbers
```

Vector- and matrix-multiplication functions:

`DOT_PRODUCT(VECTOR_A, VECTOR_B)` makes a scalar product of two vectors, which must have the same length (same number of elements). Please note that if `VECTOR_A` is of type `COMPLEX` the result is `SUM(CONJG(VECTOR_A)*VECTOR_B)`.

`MATMUL(MATRIX_A, MATRIX_B)` makes the matrix product of two matrices, which must be consistent, i.e. have the dimensions like `(M, K)` and `(K, N)`. Used in chapter 11 of the main text.

Array functions:

`ALL(MASK, dim)` returns a logical value that indicates whether all relations in `MASK` are `.TRUE.`, along only the desired dimension if the second argument is given.

`ANY(MASK, dim)` returns a logical value that indicates whether any relation in `MASK` is `.TRUE.`, along only the desired dimension if the second argument is given.

`COUNT(MASK, dim)` returns a numerical value that is the number of relations in `MASK` who are `.TRUE.`, along only the desired dimension if the second argument is given.

`MAXVAL(ARRAY, dim, mask)` returns the largest value in the array `ARRAY`, of those that obey the relation in the third argument `MASK` if that one is given, along only the desired dimension if the second argument `DIM` is given.

`MINVAL(ARRAY, dim, mask)` returns the smallest value in the array `ARRAY`, of those that obey the relation in the third argument `MASK` if that one is given, along only the desired dimension if the second argument `DIM` is given.

`PRODUCT(ARRAY, dim, mask)` returns the product of all the elements in the array `ARRAY`, of those that obey the relation in the third argument `MASK` if that one is given, along only the desired dimension if the second argument `DIM` is given.

`SUM (ARRAY, dim, mask)` returns the sum of all the elements in the array `ARRAY`, of those that obey the relation in the third argument `MASK` if that one is given, along only the desired dimension if the second argument `DIM` is given. An example is given in Appendix 3, section 10.

Array inquiry functions:

See also Appendix 3, section 10.

`ALLOCATED(ARRAY)` is a logical function which indicates if the array is allocated.

`LBOUND(ARRAY, dim)` is a function which returns the lower dimension limit for the `ARRAY`. If `DIM` (the dimension) is not given as an argument, you get an integer vector, if `DIM` is included, you get the integer value with exactly that lower dimension limit, for which you asked.

`SHAPE(SOURCE)` is a function which returns the shape of an array `SOURCE` as an integer vector.

`SIZE(ARRAY, dim)` is a function which returns the number of elements in an array `ARRAY`, if `DIM` is not given, and the number of elements in the relevant dimension if `DIM` is included.

`UBOUND(ARRAY, dim)` is a function similar to `LBOUND` which returns the upper dimensional limits.

Array construct functions:

`MERGE(TSOURCE, FSOURCE, MASK)` is a function which joins two arrays. It gives the elements in `TSOURCE` if the condition in `MASK` is `.TRUE.` and `FSOURCE` if the condition in `MASK` is `.FALSE.` The two fields `TSOURCE` and `FSOURCE` have to be of the same type and the same shape. The result is also of this type and this shape. Also `MASK` must have the same shape.

I here give a rather complete example of the use of `MERGE` which also uses `RESHAPE` from the next section in order to build suitable test matrices.

Note that the two subroutines `WRITE_ARRAY` and `WRITE_L_ARRAY` are test routines to write matrices which in the first case are of a `REAL` type, in the second case of a `LOGICAL` type.

```
IMPLICIT NONE
```

```
INTERFACE
      SUBROUTINE WRITE_ARRAY (A)
             REAL :: A(:,:)
      END SUBROUTINE WRITE_ARRAY
      SUBROUTINE WRITE_L_ARRAY (A)
             LOGICAL :: A(:,:)
      END SUBROUTINE WRITE_L_ARRAY
END INTERFACE

REAL, DIMENSION(2,3)      :: TSOURCE, FSOURCE, RESULT
LOGICAL, DIMENSION(2,3)   :: MASK
TSOURCE = RESHAPE( (/ 11, 21, 12, 22, 13, 23 /), &
                   (/ 2, 3 /) )
FSOURCE = RESHAPE( (/ -11, -21, -12, -22, -13, -23 /), &
                   (/ 2,3 /) )
MASK = RESHAPE( (/ .TRUE., .FALSE., .FALSE., .TRUE., &
                   .FALSE., .FALSE. /), (/ 2,3 /) )

RESULT = MERGE(TSOURCE, FSOURCE, MASK)
CALL WRITE_ARRAY(TSOURCE)
CALL WRITE_ARRAY(FSOURCE)
CALL WRITE_L_ARRAY(MASK)
CALL WRITE_ARRAY(RESULT)
END

SUBROUTINE WRITE_ARRAY (A)
REAL :: A(:,:)
DO I = LBOUND(A,1), UBOUND(A,1)
   WRITE(*,*) (A(I, J), J = LBOUND(A,2), UBOUND(A,2) )
END DO
RETURN
END SUBROUTINE WRITE_ARRAY

SUBROUTINE WRITE_L_ARRAY (A)
LOGICAL :: A(:,:)
DO I = LBOUND(A,1), UBOUND(A,1)
   WRITE(*,"(8L12)") (A(I, J), J= LBOUND(A,2), UBOUND(A,2))
END DO
RETURN
END SUBROUTINE WRITE_L_ARRAY
```

The following output is obtained

```
       11.0000000   12.0000000   13.0000000
       21.0000000   22.0000000   23.0000000

       -11.0000000 -12.0000000  -13.0000000
```

```
   -21.0000000 -22.0000000  -23.0000000

             T           F            F
             F           T            F

   11.0000000  -12.0000000  -13.0000000
  -21.0000000   22.0000000  -23.0000000
```

`PACK(ARRAY, MASK, vector)` packs an array to a vector with the control of `MASK`. The shape of the logical array `MASK` has to agree with the one for `ARRAY` or `MASK` must be a scalar. If `VECTOR` is included, it has to be an array of rank 1 (i.e. a vector) with at least as many elements as those that are true in `MASK` and have the same type as `ARRAY`. If `MASK` is a scalar with the value `.TRUE.` then `VECTOR` instead must have the same number of elements as `ARRAY`.

The result is a vector with as many elements as those in `ARRAY` that obey the conditions if `VECTOR` is not included (i.e. all elements if `MASK` is a scalar with value `.TRUE.`). In the other case the number of elements of the result will be as many as in `VECTOR`. The values will be the approved ones, i.e. the values which fulfill the condition, and will be in the ordinary Fortran order. If `VECTOR` is included and the number of its elements exceeds the number of approved values, the lacking values required for the result are taken from the corresponding locations in `VECTOR`.

The following example is based on the modification of the one for `MERGE` , but I give now only the results.

```
ARRAY
      11.0000000   12.0000000   13.0000000
      21.0000000   22.0000000   23.0000000

VECTOR
     -11.0000000
     -21.0000000
     -12.0000000
     -22.0000000
     -13.0000000
     -23.0000000


MASK
       T                F               F
       F                T               F
```

```
 PACK(ARRAY, MASK)
       11.0000000
       22.0000000

 PACK(ARRAY, MASK, VECTOR)
       11.0000000
       22.0000000
      -12.0000000
      -22.0000000
      -13.0000000
      -23.0000000
```

`SPREAD(SOURCE, DIM, NCOPIES)` returns an array of the same type as the argument `SOURCE` with the rank increased by one. The parameters `DIM` and `NCOPIES` are integer. If `NCOPIES` is negative the value zero is used instead. If `SOURCE` is a scalar, then `SPREAD` becomes a vector with `NCOPIES` elements that all have the same value as `SOURCE`. The parameter `DIM` indicates which index is to be extended. It has to be within the range `1` and `1+(rank of SOURCE)`, if `SOURCE` is a scalar then `DIM` has to be one. The parameter `NCOPIES` is the number of elements in the new dimensions. Additional discussion is given in the solution to exercise (11.1).

`UNPACK(VECTOR, MASK, ARRAY)` scatters a vector to an array under control of `MASK`. The shape of the logical array `MASK` has to agree with the one for `ARRAY`. The array `VECTOR` has to have the rank 1 (i.e. it is a vector) with at least as many elements as those that are true in `MASK`, and also has to have the same type as `ARRAY`. If `ARRAY` is given as a scalar then it is considered to be an array with the same shape as `MASK` and the same scalar elements everywhere.

The result will be an array with the same shape as `MASK` and the same type as `VECTOR`. The values will be those from `VECTOR` that are accepted (i.e. those fulfilling the condition in `MASK`), taken in the ordinary Fortran order, while in the remaining positions in `ARRAY` the old values are kept.

ARRAY reshape function.

`RESHAPE(SOURCE, SHAPE, pad, order)` constructs an array with a specified shape `SHAPE` starting from the elements in a given array `SOURCE`. If `PAD` is not included then the size of `SOURCE` has to be at least `PRODUCT (SHAPE)`. If `PAD` is included it has to have the same type as `SOURCE`. If `ORDER` is included, it has to be an `INTEGER` array with the same shape as `SHAPE` and the values must be a permutation of (1,2,3,...,N), where N is the number of elements in `SHAPE` , it has to be less than, or equal to 7.

The result has of course a shape `SHAPE` and the elements are those in `SOURCE`, possibly complemented with `PAD`. The different dimensions have been permuted at the assignment of the elements if `ORDER` was included, but without influencing the shape of the result.

A few simple examples are given in the previous and the next section and also in Appendix 3, section 9. A more complicated example, illustrating also the optional arguments, follows.

```
! PROGRAM TO TEST THE OPTIONAL ARGUMENTS TO RESHAPE
  INTERFACE
     SUBROUTINE WRITE_MATRIX(A)
         REAL, DIMENSION(:,:) :: A
     END SUBROUTINE  WRITE_MATRIX
  END INTERFACE

  REAL, DIMENSION (1:9) :: B = (/ 11, 12, 13, 14, 15, 16, 17, 18, 19 /)
  REAL, DIMENSION (1:3, 1:3) :: C, D, E
  REAL, DIMENSION (1:4, 1:4) :: F, G, H

  INTEGER, DIMENSION (1:2) :: ORDER1 = (/ 1, 2 /)
  INTEGER, DIMENSION (1:2) :: ORDER2 = (/ 2, 1 /)
  REAL, DIMENSION (1:16)   :: PAD1 = (/ -1, -2, -3, -4, -5, -6, -7, -8,
&
                                 &   -9, -10, -11, -12, -13, -14, -15,
-16 /)

  C = RESHAPE( B, (/ 3, 3 /) )
  CALL WRITE_MATRIX(C)

  D = RESHAPE( B, (/ 3, 3 /), ORDER = ORDER1)
  CALL WRITE_MATRIX(D)

  E = RESHAPE( B, (/ 3, 3 /), ORDER = ORDER2)
  CALL WRITE_MATRIX(E)

  F = RESHAPE( B, (/ 4, 4 /), PAD = PAD1)
  CALL WRITE_MATRIX(F)

  G = RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER1)
  CALL WRITE_MATRIX(G)
```

```
H = RESHAPE( B, (/ 4, 4 /), PAD = PAD1, ORDER = ORDER2)
CALL WRITE_MATRIX(H)

END

SUBROUTINE WRITE_MATRIX(A)
REAL, DIMENSION(:,:) :: A
WRITE(*,*)
DO I = LBOUND(A,1), UBOUND(A,1)
   WRITE(*,*) (A(I,J), J = LBOUND(A,2), UBOUND(A,2))
END DO
END SUBROUTINE WRITE_MATRIX
```

The output from the above program is as follows.

```
11.0000000  14.0000000  17.0000000
12.0000000  15.0000000  18.0000000
13.0000000  16.0000000  19.0000000

11.0000000  14.0000000  17.0000000
12.0000000  15.0000000  18.0000000
13.0000000  16.0000000  19.0000000

11.0000000  12.0000000  13.0000000
14.0000000  15.0000000  16.0000000
17.0000000  18.0000000  19.0000000

11.0000000  15.0000000  19.0000000  -4.0000000
12.0000000  16.0000000  -1.0000000  -5.0000000
13.0000000  17.0000000  -2.0000000  -6.0000000
14.0000000  18.0000000  -3.0000000  -7.0000000

11.0000000  15.0000000  19.0000000  -4.0000000
12.0000000  16.0000000  -1.0000000  -5.0000000
13.0000000  17.0000000  -2.0000000  -6.0000000
14.0000000  18.0000000  -3.0000000  -7.0000000

11.0000000  12.0000000  13.0000000  14.0000000
15.0000000  16.0000000  17.0000000  18.0000000
19.0000000  -1.0000000  -2.0000000  -3.0000000
-4.0000000  -5.0000000  -6.0000000  -7.0000000
```

ARRAY manipulation functions.

The shift functions return the shape of an array unchanged, but move the elements. They are rather difficult to explain so I recommend to study also the standard ISO (1991).

`CSHIFT(ARRAY, SHIFT, dim)` performs circular shift by `SHIFT` positions to the left if `SHIFT` is positive and to the right if it is negative. If `ARRAY` is a vector the shift is being done in a natural way, if it is an array of a higher rank then the shift is in all sections along the dimension `DIM`. If `DIM` is missing it is considered to be 1, in other cases it has to be a scalar integer number between 1 and `n` (where n equals the rank of `ARRAY`). The argument `SHIFT` is a scalar integer or an integer array of rank `n-1` and the same shape as the `ARRAY`, except along the dimension `DIM` (which is removed because of the lower rank). Different sections can therefore be shifted in various directions and with various numbers of positions.

`EOSHIFT(ARRAY, SHIFT, boundary, dim)` performs shift to the left if `SHIFT` is positive and to the right if it is negative. Instead of the elements shifted out new elements are taken from `BOUNDARY`. If `ARRAY` is a vector the shift is being done in a natural way, if it is an array of a higher rank, the shift on all sections is along the dimension `DIM`. If `DIM` is missing, it is considered to be 1, in other cases it has to have a scalar integer value between 1 and `n` (where `n` equals the rank of `ARRAY`). The argument `SHIFT` is a scalar integer if `ARRAY` has rank 1, in the other case it can be a scalar integer or an integer array of rank `n-1` and with the same shape as the array `ARRAY` except along the dimension `DIM` (which is removed because of the lower rank).

The corresponding applies to `BOUNDARY` which has to have the same type as the `ARRAY`. If the parameter `BOUNDARY` is missing you have the choice of values zero, `.FALSE.` or blank being used, depending on the data type. Different sections can thus be shifted in various directions and with various numbers of positions. A simple example of the above two functions for the vector case follows, both the program and the output.

```
REAL, DIMENSION(1:6)  :: A = (/ 11.0, 12.0, 13.0, 14.0, &
                                15.0, 16.0 /)
REAL, DIMENSION(1:6)  :: X, Y
WRITE(*,10) A
X = CSHIFT ( A, SHIFT = 2)
WRITE(*,10) X
Y = CSHIFT (A, SHIFT = -2)
WRITE(*,10) Y
X = EOSHIFT ( A, SHIFT = 2)
WRITE(*,10) X
Y = EOSHIFT ( A, SHIFT = -2)
```

```
WRITE(*,10) Y
10  FORMAT(1X,6F6.1)
END

       11.0  12.0  13.0   14.0  15.0  16.0
       13.0  14.0  15.0   16.0  11.0  12.0
       15.0  16.0  11.0   12.0  13.0  14.0
       13.0  14.0  15.0   16.0   0.0   0.0
        0.0   0.0  11.0   12.0  13.0  14.0
```

A simple example of the above two functions in the matrix case follows. I have here used `RESHAPE` in order to create a suitable matrix to start work with. The program is not reproduced here, only the main statements.

```
B = (/ 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /)

  11.0  12.0  13.0   Z = RESHAPE( B, (/3,3/) )
  14.0  15.0  16.0
  17.0  18.0  19.0

  17.0  18.0  19.0   X = CSHIFT (Z, SHIFT = 2)
  11.0  12.0  13.0
  14.0  15.0  16.0

  13.0  11.0  12.0   X = CSHIFT ( Z, SHIFT = 2, DIM = 2)
  16.0  14.0  15.0
  19.0  17.0  18.0

  14.0  15.0  16.0   X = CSHIFT (Z, SHIFT = -2)
  17.0  18.0  19.0
  11.0  12.0  13.0

  17.0  18.0  19.0   X = EOSHIFT ( Z, SHIFT = 2)
   0.0   0.0   0.0
   0.0   0.0   0.0

  13.0   0.0   0.0   X = EOSHIFT ( Z, SHIFT = 2, DIM = 2)
  16.0   0.0   0.0
  19.0   0.0   0.0

   0.0   0.0   0.0   X = EOSHIFT ( Z, SHIFT = -2)
   0.0   0.0   0.0
  11.0  12.0  13.0
```

`TRANSPOSE (MATRIX)` transposes a matrix, which is an array of rank 2. It replaces the rows and columns in the matrix.

Array location functions:

`MAXLOC(ARRAY, mask)` returns the position of the greatest element in the array `ARRAY`, if `MASK` is included only for those which fulfill the conditions in `MASK`. The result is an integer **vector!** It is used in the solution of exercise (11.1).

`MINLOC(ARRAY, mask)` returns the position of the smallest element in the array `ARRAY` , if `MASK` is included only for those which fulfill the conditions in `MASK`. The result is an integer **vector!**

Pointer inquiry functions:

`ASSOCIATED(POINTER, target)` is logical function that indicates if the pointer `POINTER` is associated with some target, and if a specific `TARGET` is included it indicates if it is associated with exactly that target. If both `POINTER` and `TARGET` are pointers, the result is `.TRUE.` only if both are associated with the same target. I refer the reader to chapter 12 of the main text, Pointers.

21. ANSWERS AND COMMENTS TO THE USER EXERCISES

It is assumed that when nothing else is stated, the implicit rules about integers and floating-point numbers are used, i.e. `IMPLICIT NONE` has not been used. In those cases where runs on computers shall be done, I refer the reader to Appendix 6, NAG's Fortran 90, for MS-DOS and UNIX computers. For IBM PC there is a very complete documentation in the booklet "NAGware FTN90 compiler". In all other cases please try the manuals from the computer or compiler manufacturer.

(1.1) If the compilation or the execution run fails it is probably an error already in the Fortran 77 program, or you have used some extension to standard Fortran 77.

(1.2) If this fails it probably depends on that some incorrect commands were interpreted as variables when using fixed form, but now when blanks are significant these syntax errors are discovered. Also note that with fix form text in positions 73 to 80 was considered to be a comment.

(1.3) Fortran 77 does not give any error either on the compilation or execution. Compilation in Fortran 90 fixed form may give a warning from the compiler that the variable `ZENDIF` is used without being assigned any value. The program is interpreted in such a way that `THENZ, ELSEY`, and `ZENDIF` becomes ordinary floating-point variables. Compilation in Fortran 90 free form, however, gives a number of syntax errors. The correct version of the program shall contain three extra carriage returns as below.

```
LOGICAL L
L = .FALSE.
IF (L) THEN
       Z = 1.0
ELSE
       Y=Z
END IF
END
```

REMARK: Also certain Fortran 77 compilers give a warning about the variable `ZENDIF`, which has not been assigned any value.

(2.1) Using fixed form it means `LOGICAL L`, i.e. the variable `L` is specified as logical. Using free form you will get a syntax error.

(2.2) `REAL, PARAMETER :: K = 0.75`

(2.3) `INTEGER, DIMENSION(3,4) :: PELLE`

(2.4)

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

(2.5) `REAL (KIND=DP) :: E, PI`

(2.6)

```
REAL (KIND=DP), PARAMETER  :: E = 2.718281828459045_DP, PI =
3.141592653589793_DP
```

(2.7) No, it is not correct since a comma is missing between `REAL` and `DIMENSION`. In the form it has been written, the statement is interpreted as a specification of the old type of the floating-point matrix `DIMENSION` (with the specified dimensions), and an implicit specification of the new type of a scalar floating-point number `AA`. Formally, it is a correct specification. The variable name `DIMENSION` is permitted in Fortran 90, just as the variable name `REAL` is permitted in both Fortran 77 and Fortran 90, but both should be avoided. The variable name `DIMENSION` is of course too long in standard Fortran 77.

(2.8) Yes, it is correct, but it is not suitable since it kills the intrinsic function `REAL` for explicit conversion of a variable of another type to the type `REAL`. It is however nothing that prevents you from using a variable of the type `REAL` with the name `REAL`, since Fortran does not have reserved words.

(2.9) No, it is not correct, at `COMMON` you do not use the double colon at the specification. The correct specification is the old familiar one: `COMMON A`

(3.1) Variables `A` and `B` are assigned the specified values, but the whole rest of the line becomes a comment.

(3.2) No, on the second row the blank space after the ampersand (&) is not permitted. It interrupts the identifier `ATAN` into two identifiers `AT` and `AN`. If the blank is removed the two lines become correct. Free form is assumed, since & is not a continuation character in fixed form.

(4.1) The statement is not permitted, but might not be detected until execution time. You can instead write

```
            WRITE(*,*) ' HI '
```
or
```
            WRITE(*,'(A)') ' HI '
```
which both write out the text `HI` on the standard unit for output. If you wish to give the text, which you wish to print, directly where the output format is to be given, this can be done with either apostrophe editing as
```
            WRITE(*, "(' HI ')")
```
or with the obsolescent Hollerith editing
```
            WRITE(*, "(4H HI )")
```

(4.2) They write large and small numbers with an integer digit, six decimals and an exponent, while numbers in between are written in the natural way. In this case we thus get

```
         1.000000E-03
         1.00000
         1.000000E+06
```
Numbers from `0.1` to `100 000` are written in the natural way and with six significant digits.

(6.1)

```
       SELECT CASE (N)
       CASE(:-1)
              ! Case 1
       CASE(0)
              ! Case 2
       CASE(3,5,7,11,13)
              ! Case 3
       END SELECT
```
(6.2)
```
       SUMMA = 0.0
       DO I = 1, 100
```

```
             IF ( X(I) == 0.0) EXIT
             IF ( X(I) <  0.0) CYCLE
             SUMMA = SUMMA + SQRT (X(I))
        END DO
```

The English word `sum` is not suited as the variable name in this case, since this is also an intrinsic function. Summa is the Swedish word for sum.

(7.1) Use the functions `MIN` and `MAX` to find the smallest and largest values of all the combinations

```
A%LOWER * B%LOWER,  A%LOWER * B%UPPER,  A%UPPER * B%LOWER, A%UPPER *
B%UPPER
```

at multiplication and the corresponding at division.

(7.2) Test if `B%LOWER <= 0 <= B%UPPER` in which case an error message shall be given.

(7.3) Since we do not have direct access to machine arithmetics (i.e. commands of the type round down or round up) you can get a reasonable simulation through subtraction and addition with the rounding constant. In principle the effect of rounding is then doubled.

(8.1)

```
        SUBROUTINE SOLVE(F, A, B, TOL, EST, RESULT)
        REAL, EXTERNAL                :: F
        REAL, OPTIONAL, INTENT (IN)   :: A
        REAL, OPTIONAL, INTENT (IN)   :: B
        REAL, OPTIONAL, INTENT (IN)   :: TOL
        REAL, INTENT(OUT), OPTIONAL   :: EST
        REAL, INTENT(OUT)             :: RESULT
        IF (PRESENT(A)) THEN
               TEMP_A = A
        ELSE
               TEMP_A = 0.0
        END IF
        IF (PRESENT(B)) THEN
               TEMP_B = B
        ELSE
               TEMP_B = 1.0
        END IF
```

```
       IF (PRESENT(TOL)) THEN
              TEMP_TOL = TOL
       ELSE
              TEMP_TOL = 0.001
       END IF

! Here the real calculation should be, but it is here replaced
! with the  simplest possible approximation,  namely the middle
! point approximation without an error estimate.

       RESULT = (TEMP_B - TEMP_A)&
        * F(0.5*(TEMP_A+TEMP_B))
       IF (PRESENT(EST)) EST = TEMP_TOL

       RETURN
       END SUBROUTINE SOLVE
```

The very simple integral calculation above can be replaced by the adaptive quadrature in exercise (9.2).

(8.2)

```
       INTERFACE
       SUBROUTINE SOLVE (F, A, B, TOL, EST, RESULT)
              REAL, EXTERNAL                 :: F
              REAL, INTENT(IN), OPTIONAL     :: A
              REAL, INTENT(IN), OPTIONAL     :: B
              REAL, INTENT(IN), OPTIONAL     :: TOL
              REAL, INTENT(OUT), OPTIONAL    :: EST
              REAL, INTENT(OUT)              :: RESULT
              END SUBROUTINE SOLVE
       END INTERFACE
```

(9.1)

```
       RECURSIVE FUNCTION TRIBONACCI (N) RESULT (T_RESULT)
       IMPLICIT NONE
       INTEGER, INTENT(IN)   :: N
       INTEGER               :: T_RESULT
       IF ( N <= 3 ) THEN
             T_RESULT = 1
       ELSE
             T_RESULT = TRIBONACCI(N-1 )+ &
             TRIBONACCI(N-2) + TRIBONACCI(N-3)
       END IF
       END FUNCTION TRIBONACCI
```

The calling program or main program can be written

```
       IMPLICIT NONE
       INTEGER       :: N, M, TRIBONACCI
       N = 1
       DO
              IF ( N <= 0 ) EXIT
                     WRITE (*,*) ' GIVE N '
                     READ(*,*) N
                     M = TRIBONACCI  (N)
                     WRITE(*,*) N, M
       END DO
       END
```

and gives the result `TRIBONACCI(15) = 2209.`

(9.2) The file `quad.f90` below contains a function for adaptive numerical quadrature (integration). We use the trapezoidal formula, divide the step size with two, and perform Richardson extrapolation. The method is therefore equivalent to the Simpson formula. As an error estimate we use the model in Linköping, where the error is assumed less than the modulus of the difference between the two not extrapolated values. If the estimated error is too large, the routine is applied once again on each of the two subintervals, in that case the permitted error in each one of the subintervals becomes half of the error previously used.

```
RECURSIVE FUNCTION ADAPTIVE_QUAD (F, A, B, TOL, ABS_ERROR) &
                RESULT (RESULT)
IMPLICIT NONE

       INTERFACE
              FUNCTION F(X) RESULT (FUNCTION_VALUE)
              REAL, INTENT(IN) :: X
              REAL             :: FUNCTION_VALUE
              END FUNCTION F
       END INTERFACE

       REAL, INTENT(IN)        :: A, B, TOL
       REAL, INTENT(OUT)       :: ABS_ERROR
       REAL                    :: RESULT

       REAL                    :: STEP, MIDDLE_POINT
       REAL                    :: ONE_TRAPEZOIDAL_AREA,
TWO_TRAPEZOIDAL_AREAS
       REAL                    :: LEFT_AREA, RIGHT_AREA
       REAL                    :: DIFF, ABS_ERROR_L, ABS_ERROR_R

       STEP = B-A
       MIDDLE_POINT= 0.5 * (A+B)

       ONE_TRAPEZOIDAL_AREA = STEP * 0.5 * (F(A)+ F(B))
       TWO_TRAPEZOIDAL_AREAS = STEP * 0.25 * (F(A) + F(MIDDLE_POINT))+&
                           STEP * 0.25 * (F(MIDDLE_POINT) + F(B))
       DIFF = TWO_TRAPEZOIDAL_AREAS - ONE_TRAPEZOIDAL_AREA

       IF ( ABS (DIFF) < TOL ) THEN
              RESULT = TWO_TRAPEZOIDAL_AREAS + DIFF/3.0
              ABS_ERROR = ABS(DIFF)
       ELSE
              LEFT_AREA = ADAPTIVE_QUAD (F, A, MIDDLE_POINT, &
                      0.5*TOL, ABS_ERROR_L)
              RIGHT_AREA = ADAPTIVE_QUAD (F, MIDDLE_POINT, B, &
                       0.5*TOL, ABS_ERROR_R)
              RESULT = LEFT_AREA + RIGHT_AREA
              ABS_ERROR = ABS_ERROR_L + ABS_ERROR_R
       END IF
END FUNCTION ADAPTIVE_QUAD
```

The file test_qua.f90 for the test of the above routine for adaptive numerical quadrature requires an INTERFACE both for the function F and for the quadrature routine ADAPTIVE_QUAD. Note that for the latter you must specify the function both REAL and EXTERNAL and that routine follows.

```
PROGRAM TEST_ADAPTIVE_QUAD
IMPLICIT NONE
       INTERFACE
              FUNCTION F(X) RESULT (FUNCTION_VALUE)
              REAL, INTENT(IN)       :: X
              REAL                   :: FUNCTION_VALUE
              END FUNCTION F
       END INTERFACE
       INTERFACE
              RECURSIVE FUNCTION ADAPTIVE_QUAD &
                      (F, A, B, TOL, ABS_ERROR) RESULT (RESULT)
              REAL, EXTERNAL         :: F
              REAL, INTENT (IN)      :: A, B, TOL
              REAL, INTENT (OUT)     :: ABS_ERROR
              REAL                  :: RESULT
              END FUNCTION ADAPTIVE_QUAD
       END INTERFACE
       REAL          :: A, B, TOL
       REAL          :: ABS_ERROR
       REAL          :: RESULT, PI
       INTEGER       :: I

       PI = 4.0 * ATAN(1.0)
       A= -5.0
       B = +5.0
       TOL =0.1

       DO I = 1, 5
              TOL = TOL/10.0
              RESULT = ADAPTIVE_QUAD (F, A, B, TOL, ABS_ERROR)
              WRITE(*,*)
              WRITE(*,"(A, F15.10, A, F15.10)") &
               "The integral is approximately ", &
              RESULT, "with approximate error estimate ", &
              ABS_ERROR
              WRITE(*,"(A, F15.10, A, F15.10)") &
              "The integral is more exactly    ", &
                SQRT(PI), " with real error            ", &
               RESULT - SQRT(PI)
       END DO
END PROGRAM TEST_ADAPTIVE_QUAD
```

We are of course not permitted to forget the integrand, which we prefer to put in the same file as the main program. Declarations are of the new type especially with respect to that the result is returned in a special variable.

```
       FUNCTION F(X) RESULT (FUNCTION_VALUE)
       IMPLICIT NONE
       REAL, INTENT(IN)       :: X
       REAL                   :: FUNCTION_VALUE
       FUNCTION_VALUE = EXP(-X**2)
       END FUNCTION F
```

Now it is time to do the test on the Sun computer. I have adapted the output a little in order to get it more compact. The error estimated is rather realistic, at least with this integrand, which is the unnormalized error function.

If you wish to test the program yourself the source code is directly available in two files. The first `test_qua.f90` contains the main program and the function *f(x)*, while the second `quad.f90` contains the recursive function.

```
% f90 test_qua.f90 quad.f90
test_quad.f90:
quad.f90:
% a.out
The integral is 1.7733453512  with error estimate  0.0049186843
                              with real error      0.0008914471
The integral is 1.7724548578  with error estimate  0.0003375171
                              with real error      0.0000009537
The integral is 1.7724541426  with error estimate  0.0000356939
                              with real error      0.0000002384
The integral is 1.7724540234  with error estimate  0.0000046571
                              with real error      0.0000001192
The integral is 1.7724539042  with error estimate  0.0000004876
                              with real error      0.0000000000
%
```

In the specification above of the `RECURSIVE FUNCTION ADAPTIVE_QUAD` you may replace the line

```
              REAL, EXTERNAL         :: F
```

with a complete repetition of the interface for the integrand function,

```
       INTERFACE
              FUNCTION F(X) RESULT (FUNCTION_VALUE)
              REAL, INTENT(IN)       :: X
              REAL                   :: FUNCTION_VALUE
              END FUNCTION F
       END INTERFACE
```

With this method an explicit `EXTERNAL` statement is no longer required, but you get a nested `INTERFACE`.

Remark.

The program above was written to illustrate the use of recursive functions and adaptive techniques, and was therefore not optimized. The main problem is that the function *f(x)* is evaluated three (or even four) times at each call, once for each of the present boundary points and twice for the middle point. Please note that the function values at each of the boundary points were evaluated already in the previous step.

Thus the obvious change is to include the boundary function values in the list of arguments, and to evaluate the middle point function value only once. In this way the execution time is reduced by a factor of about three.

The revised program is also directly available in two files. The first `test_qu2.f90` contains the main program and the function *f(x)*, while the second `quad2.f90` contains the recursive function.

(11.1)

```
SUBROUTINE SOLVE_SYSTEM_OF_LINEAR_EQNS(A, X, B, ERROR)
IMPLICIT NONE
! Array specifications
REAL, DIMENSION (:, :),           INTENT (IN) :: A
REAL, DIMENSION (:),              INTENT (OUT):: X
REAL, DIMENSION (:),              INTENT (IN) :: B
LOGICAL, INTENT (OUT)                         :: ERROR

! The working area M is A expanded with B
REAL, DIMENSION (SIZE (B), SIZE (B) + 1)      :: M
INTEGER, DIMENSION (1)                        :: MAX_LOC
REAL, DIMENSION (SIZE (B) + 1)                :: TEMP_ROW
INTEGER                                       :: N, K, I

! Initializing M
N = SIZE (B)
M (1:N, 1:N) = A
M (1:N, N+1) = B

! Triangularization
ERROR = .FALSE.
TRIANGULARIZATION_LOOP: DO K = 1, N - 1
        ! Pivoting
        MAX_LOC = MAXLOC (ABS (M (K:N, K)))
        IF ( MAX_LOC(1) /= 1 ) THEN
                TEMP_ROW (K:N+1 ) =M (K, K:N+1)
                M (K, K:N+1)= M (K-1+MAX_LOC(1), K:N+1)
                M (K-1+MAX_LOC(1), K:N+1) = TEMP_ROW(K:N+1)
        END IF

        IF (M (K, K) == 0) THEN
```

```
               ERROR = .TRUE. ! Singular matrix A
               EXIT TRIANGULARIZATION_LOOP
        ELSE
               TEMP_ROW (K+1:N) = M (K+1:N, K) / M (K, K)
               DO I = K+1, N
                      M (I, K+1:N+1) = M (I, K+1:N+1) - &
                        TEMP_ROW (I) * M (K, K+1:N+1)
               END DO
               M (K+1:N, K) =0  ! These values are not used
        END IF
END DO TRIANGULARIZATION_LOOP

IF ( M(N, N) == 0 ) ERROR = .TRUE.  ! Singular matrix A

! Re-substitution
IF (ERROR) THEN
        X = 0.0
ELSE
        DO K = N, 1, -1
               X (K) = (M (K, N+1) - &
                 SUM (M (K, K+1:N)* X (K+1:N)) ) / M (K, K)
        END DO
END IF
END SUBROUTINE SOLVE_SYSTEM_OF_LINEAR_EQNS
```

The input matrix `A` and the vectors `B` and `X` are specified as assumed-shape arrays, i.e. type, rank and name are given here, while the extent is given in the calling program unit, using an explicit interface.

Please note that the intrinsic function `MAXLOC` as a result gives an integer vector, with the same number of elements as the rank (number of dimensions) of the argument. In our case the argument is a vector and therefore the rank is 1 and `MAXLOC` is a vector with only 1 element. This is the reason why the local variable `MAX_LOC` has been declared as a vector with 1 element. If you declare `MAX_LOC` as a scalar you get a compilation error. The value of course is the index for the largest element (the first of the largest if there are several of these).

Also note that the numbering starts with 1, in spite of that we are looking at the vector with the elements running from `K` to `N`. I prefer not to perform the pivoting process (that is the actual exchange of rows) in the special case that the routine finds that the rows already are correctly located, i.e. when `MAX_LOC(1)` is 1.

The calculation is interrupted as soon as a singularity is found. Please note that this can occur so late that it is not noted inside the loop, thus the extra check immediately after the loop, for the final element `M(N, N)`.

At the pivoting process we use the vector `TEMP_ROW` first at the exchange of lines, then also to store the multipliers in the Gauss elimination.

In this first version we only use array sections of vector type at the calculations, but we will now introduce the function `SPREAD` in order to use array sections of matrix type, and in this case the explicit inner loop disappears `(DO I = K+1, N)` .

The function `SPREAD(SOURCE, DIM, NCOPIES)` returns an array of the same type as the argument `SOURCE`, but with the rank increased by one. The parameters `DIM` and `NCOPIES` are integers. If `NCOPIES` is negative the value zero is used instead. If `SOURCE` is a scalar, `SPREAD` becomes a vector with `NCOPIES` elements which all have the same value as `SOURCE`. The parameter `DIM` gives which index is to be increased. That must be a value between `1` and `1+(rank of SOURCE)`. If `SOURCE` is a scalar, then `DIM` has to be 1. The parameter `NCOPIES` gives the total number of elements in the new dimension, thus not only the number of new copies but also the original.

If the variable `A` corresponds to the following array
(/ 2, 3, 4 /) we get

```
SPREAD (A, DIM=1, NCOPIES=3)  SPREAD (A, DIM=2, NCOPIES=3)

               2   3   4                     2   2   2
               2   3   4                     3   3   3
               2   3   4                     4   4   4
```

I now use array sections of matrix type through replacing the inner loop,

```
DO I = K+1, N
       M (I, K+1:N+1) = M (I, K+1:N+1) - &
         TEMP_ROW (I) * M (K, K+1:N+1)
END DO
```

with

```
M (K+1:N, K+1:N+1) = M (K+1:N, K+1:N+1) &
   - SPREAD( TEMP_ROW (K+1:N), 2, N-K+1) &
   * SPREAD( M (K, K+1:N+1), 1, N-K)
```

The reason that we have to make it almost into a muddle with the function `SPREAD` is that in the explicit loop (at a fixed value of `I`) the variable `TEMP_ROW(I)` is a scalar constant, which is multiplied by `N-K+1` different elements of the matrix `M`, or a vector of `M`. On the other hand, the same vector of `M` is used for all `N-K` values of `I`. The rearrangement of the matrices has to be done to obtain two matrices with the same shape as the submatrix `M(K+1:N, K+1:N+1)`, that is `N-K` rows and `N-K+1` columns, since all calculations on arrays in Fortran 90 are element by element.

Unfortunately it is rather difficult to get the parameters to the intrinsic function `SPREAD` absolutely correct. In order to get them correct you can utilize the functions `LBOUND` and `UBOUND` in order to obtain the lower and upper dimension limits, and you can also write the new array with the following statement

```
WRITE(*,*) SPREAD (SOURCE, DIM, NCOPIES)
```

Please note that the output is done column by column (i.e. the first index is varying fastest, as it is usual in Fortran). You can use the lower and upper dimension limits for more explicit output statements that give an output which is better suited to how the array looks. However, here you have to first make an assignment to an array, specified in the usual way with the correct shape, in order to use the indices in the ordinary way. Please remember that the indices in a construct like `SPREAD` automatically go from one as the lower limit. Even when you give something like `A(4:7)` as `SOURCE` then the result will have the index going or ranging from 1 to 4.

The DO-loop

```
        DO K = N, 1, -1
               X (K) = (M (K, N+1) - &
                 SUM (M (K, K+1:N)* X (K+1:N)) ) / M (K, K)
        END DO
```

may cause a signal on out-of-bound index from an empty sum when K = N and index checking (not standard) is on. With the following slight change this potential problem is avoided.

```
        X (N) = M (N, N+1) / M (N, N)
        DO K = N-1, 1, -1
               X (K) = (M (K, N+1) - &
                 SUM (M (K, K+1:N)* X (K+1:N)) ) / M (K, K)
        END DO
```

This version is now included in the single precision and double precision routines stored as Fortran files.

(12.1) We assume that the vector has a fixed dimension, and we perform a control output of a few of the values.

```
      REAL, TARGET, DIMENSION(1:100) :: VECTOR
      REAL, POINTER, DIMENSION(:)    :: ODD, EVEN

      ODD => VECTOR(1:100:2)
      EVEN => VECTOR(2:100:2)

      EVEN = 13
      ODD = 17

      WRITE(*,*) VECTOR(11), VECTOR(64)

      END
```

(12.2) We assume that the given vector has a fixed dimension.

```
      REAL, TARGET, DIMENSION(1:10)   :: VECTOR
      REAL, POINTER, DIMENSION(:)     :: POINTER1
      REAL, POINTER                   :: POINTER2

      POINTER1 => VECTOR
      POINTER2 => VECTOR(7)
```

(12.3) We use an `INTERFACE` with pointers in the main program and allocate, using pointers, a matrix in the subroutine. In this way we get a dynamically allocated matrix.

```
      PROGRAM MAIN_PROGRAM
      INTERFACE
         SUBROUTINE SUB(B)
         REAL, DIMENSION (:,:), POINTER :: B
         END SUBROUTINE SUB
      END INTERFACE
      REAL, DIMENSION (:,:), POINTER :: A
      CALL SUB(A)
!     Now we can use the matrix A.
!     Its dimensions were determined in the subroutine,
!     the number of elements is available as SIZE(A),
!     the extent in each direction as SIZE(A,1) and
!     as SIZE(A,2).
!
      END PROGRAM MAIN_PROGRAM

      SUBROUTINE SUB(B)
      REAL, DIMENSION (:,:), POINTER :: B
      INTEGER M, N
!     Here we can assign values to M and N, for example
!     through an input statement.
!     When M and N have been assigned we can allocate B
!     as a matrix.
      ALLOCATE (B(M,N))
!     Now we can use the matrix B.
      END SUBROUTINE SUB
```

22. REFERENCES

The Fortran Market contains a listing of selected books and free online tutorials on Fortran 90.

The comments below are by Bo Einarsson.

- **Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith and Jerrold L. Wagener:** Fortran 90 Handbook, Complete ANSI/ISO Reference, McGraw-Hill, New York 1992. ISBN 0-07-000406-4. $79.95.
  Complete guide to Fortran 90 and its use. Written by persons that were involved in the development of Fortran 90. Contains hundreds of examples. However, most of these are very short and not complete program units. Much more readable and easier to use than the formal standards, but in spite of this it is not suitable as the only aid to a beginner in Fortran 90.
- **Ed Akin:** Object-Oriented Programming via Fortran 90/95, Cambridge University Press, Cambridge 2003. ISBN 0-521-52408-3.
- **ANSI:** Programming Language Fortran, X3.9-1978, American National Standard. $24.00.
  The official standard for Fortran 77. It is possible to use for reference, but it requires that you know the basics of the language.
  It is now also available in an HTML version of the Fortran 77 Standard.
- **ANSI:** Programming Language Fortran 90, X3.198-1992, American National Standard.
  The official American standard for Fortran 90. The same book as ISO below.
- **Katarina Blom:** Fortran 90 - en introduktion, Studentlitteratur, Lund 1994. ISBN 91-44-47881-X
  A tutorial in Swedish on Fortran 90. The book also describes some basic programming practices and numerical methods. No previous programming experience is required.
- **Walter S. Brainerd, Charles H. Goldberg and Jeanne C. Adams:** Programmer's Guide to Fortran 90, Third Edition, Springer, 1995. DEM 58.00. ISBN 0-387-94570-9
  One of the first books about Fortran 90. Easy to read. Each new concept that is presented is given a simple example and therefore you can easily see how each

concept is used. The book is written by persons that were involved in the development of Fortran 90. The book is recommended.

- **Stephen J Chapman**: Introduction to Fortran 90/95, McGraw-Hill, Boston 1998. ISBN 0-07-011969-4.
- **Thomas F. Coleman and Charles Van Loan:** Handbook for Matrix Computations, Frontiers in Applied Mathematics, Vol. 4, SIAM, Philadelphia 1988. ISBN 0-89871-227-0.
  The first chapter is an excellent introduction to Fortran 77. Very easy to read. Also treats BLAS, LINPACK and MATLAB.
- **Martin Counihan:** Fortran 90, Pitman, London 1990. ISBN 0-273-03073-6.
  I have not seen this book, but it is rumoured to be easy to understand and it gives a lot of examples.
- **Martin Counihan:** Fortran 95, UCL Press, London 1996. ISBN 1-85728-367-8.
- **Cray:** Fortran Language Reference Manual, Volume 1, SR-3902 3.0, Volume 2, SR-3903 3.0, Volume 3, SR-3905 3.0,
  Treats not only the whole language Fortran 90 but also how it is used on the Cray, with some extensions.
- **DEC:** DEC Fortran, Language Reference Manual, AA-PNU0A-TK, March 1992.
  This is a complete manual which also treats Fortran 77 and all the extensions made by Digital. Necessary for DEC-programmers. Very expensive.
- **DEC:** DEC Fortran for ULTRIX RISC Systems, User Manual, AA-PNU1A-TE, March 1992.
  Auxiliary manual on the ULTRIX - environment for Fortran 77. Necessary book for the serious DEC-programmer. Is usually bought together with the book above.
- **Zane Dodson:** A Fortran 90 Tutorial, Computer Science Department, University of New Mexico, 27 June 1994.
  PostScript, 56 pages.
- **Stacey L. Edgar:** FORTRAN for the '90s, Problem Solving for Scientists and Engineers, Computer Science Press, New York, 1992. ISBN 0-7167-8247-2. $19.95.
  Complete textbook in both programming in Fortran 77 and in Fortran 90. Many examples from many different areas from science and technology. In each chapter new features of Fortran 90 are discussed and Fortran 90 is also more fully discussed in the concluding chapter. The book is recommended.
- **Bo Einarsson:** Lärobok i Fortran 90/95, Linus & Linnea, Linköping 1994.
  Fortran 90 Tutorial in Swedish, PostScript version. Available according to instructions on my Fortran page.

- **Bo Einarsson:** Lärobok i Fortran 90, Linköping 1995.
  Fortran 90 Tutorial in Swedish, hypertext version.
- **Bo Einarsson and Yurij Shokin:** FORTRAN-90, Kniga dlja programmiruyushchikh na yazyke Fortran-77, Izdatel'stvo Sibirskogo Otdeleniya Rossijskaya Akademiya Nauk, Novosibirsk 1995. ISBN 5-85826-013-6.
  Fortran 90 for the Fortran 77 programmer, Textbook in Russian, published by the Siberian Division of the Russian Academy of Sciences*, Novosibirsk 1995.*
  Cover and Title page are available as pictures.
- **Bo Einarsson:** Some Experiences from Teaching Fortran 90, Fortran Journal, Volume 8, Number 1, 1996 January/February, pp. 2, 4-6.
- **Torgil Ekman and Göran Eriksson:** Programmering i Fortran 77, Third edition, Studentlitteratur, Lund 1984. ISBN 91-44-16663-X
  An excellent tutorial on Fortran 77. Describes all the commands of Fortran. It is recommended to previously have read a book on another language, like Pascal. Appendix C is both well-done and very important. The book is recommended to those who are fluent in Swedish.
- **T. M. R. Ellis:** Fortran 77 Programming, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts 1990. ISBN 0-201-41638-7.
  Complete book in both programming in general and in Fortran 77. Many examples and good exercises. The last chapter treats Fortran 90.
- **T. M. R. Ellis, I. R. Philips and T. M. Lahey:** Fortran 90 Programming, Addison-Wesley Publishing Company, Reading, Massachusetts 1994. ISBN 0-201-54446-6.
  Complete book in both programming in general and in Fortran 90. Many examples and good exercises. The book is recommended.
- **M. Etzel, K. Dickinson:** Digital Visual Fortran 90 Programmer's Guide, Digital Press, Boston, Massachusetts 1999. ISBN 1-55558-218-4.
  Complete book on programming with the very popular Visual Fortran, now from HP (COMPAQ). The book is recommended.
- **High Performance Fortran Forum:** High Performance Fortran Language Specification, Version 1.0, 3 May 1993. Technical Report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, Texas 77251.
  Available via anonymous ftp from `titan.cs.rice.edu` as the file `/public/HPFF/draft/hpf-v10-final.ps.Z`. Includes 12 + 184 pages. Also available here. It has also been published in the **Fortran Forum**, Vol. 12, No. 4 (December 1993), Vol. 13, No. 2, (June 1994), and Vol. 13, No. 3, (September 1994).

Very easy to read compared with most other standards, and has many good examples.
The latest versions are now available both in PostScript and HTML from Rice University or from the mirror at Vienna University.

- **Wilhelm Gehrke:** Fortran 90 Referenz-Handbuch, Hanser, München 1991. ISBN 3-446-16321-2. DM 168.00.
Complete description in German of Fortran 90. The book can be used as a textbook but it is mainly for reference use. I find it rather easy to read. It treats and explains everything. It is very similar to the book of Adams et al.
- **Wilhelm Gehrke (editor):** Fortran 90 Language Guide, Springer, 1995, ISBN 3-540-19926-8. DM 68.00.
- **Wilhelm Gehrke:** Fortran 95 Language Guide, Springer, 1996, ISBN 3-540-76062-8. DM 64.00.
- **ISO:** ISO/IEC 1539:1991, Information Technology - Programming Languages - Fortran, Second Edition, 1991-07-01, ISO Publications Department, Case Postale 56, CH-1211 Geneva 20, Switzerland. SFR 185.
The standard can also be available in electronic form both in ASCII and PostScript for a certain charge from Walt Brainerd, Unicomp Inc., 235 Mt. Hamilton Avenue, Los Altos, CA 94022, Fax + 1 415 949 4058, E-mail `walt@fortran.com`. Further information is available.
The official standard for Fortran 90. Rather difficult as a dictionary. Requires that you have read a textbook on Fortran 90. The book is recommended.
- **ISO:** ISO/IEC 1539-2:1994, Information Technology - Programming Languages - Fortran - Part 2: Varying length characater strings, ISO Publications Department, Case Postale 56, CH-1211 Geneva 20, Switzerland.
The complete text is available electronically, see further information on Bo Einarsson's Fortran page.
- **James F. Kerrigan:** Migrating to Fortran 90, O'Reilly & Associates, Sebastopol, CA 1993, 389 pages, ISBN 1-56592-049-X. $27.95.
It is a practical guide to Fortran 90 for the current Fortran 77 programmer.
- **Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele and Mary E. Zosel:**The High Performance Fortran Handbook, The MIT Press, Cambridge, Massachusetts 1994. ISBN-0-262-61094-9. $ 24.95.
A very good book, not only about HPF but also with very good explanations of various parts of Fortran 90. The book is recommended.
- **Elliot B. Koffman and Frank L. Friedman:** Problem Solving and Structured Programming in Fortran 77, Fourth Edition, Addison-Wesley Publishing Company, Reading, Massachusetts 1990. ISBN 0-201-51216-5.

A complete textbook in both programming in general and in Fortran 77. Many examples and good exercises. Appendix D treats Fortran 8X (the previous version of Fortran 90). I find this look a little more easy to read than the one of Ellis. The book is recommended.

- **Erasmus Langer:** Programmieren in Fortran, Springer, Vienna 1993, ISBN 3-211-82446-4. DEM 45.
  Tutorial in German on Fortran 90. Contains a unique appendix on the floating point representation on the most commonly used computers.
- **Norman Lawrence:** Compaq Visual Fortran, A Guide to Creating Windows Applications, Digital Press, Boston, Massachusetts 2002. ISBN 1-55558-249-4.
- **John M. Levesque and Joel W. Williamson:** A Guidebook to Fortran on Supercomputers, Academic Press, San Diego, CA, 1989. ISBN 0-12-444760-0.
  This book treats a lot of tricks in order to vectorize Fortran 77 programs, especially on the Cray. Many of these tricks are however already included in the Cray compiler. The book also describes some supercomputer architectures.
- **Mike Loukides:** UNIX for Fortran Programmers, Nutshell Handbooks, O'Reilly & Associates, Sebastopol, CA 1990, ISBN 0-937175-51-X. $24.95.
  An excellent UNIX textbook in Fortran programming. It has taught me how libraries are used in UNIX. The book is recommended.
- **Michael Metcalf:** Fortran Optimization, Academic Press, London and New York 1982. ISBN 0-12-492480-8.
  A classical book how you get efficient Fortran 77 programs on a conventional computer.
- **Michael Metcalf and John Reid:** Fortran 90 Explained, Oxford University Press, Oxford, 1990. ISBN 0-19-853772-7. $29.95.
  This book was reprinted with corrections in 1993. A good and rather easy to read textbook written by persons involved in the development of Fortran 90. The 1993 printing contains a very complete application example.
- **Michael Metcalf and John Reid:** Fortran 90/95 Explained, Second edition, Oxford University Press, Oxford and New York, 1999. ISBN 0-19-850558-2.
  This book is appended with Fortran 95, and is highly recommended. The second edition also contains one chapter on floating-point exception handling and one on allocatable dummy arguments, functions, and derived-type components. These chapers correspond to ISO-approved extensions that will be part of Fortran 2000.
- **Michael Metcalf:** Fortran 90 CNL Articles
- **NAG:** NAGWare f95 Compiler (Unix), Release 5.0, NP3655, November 2003. ISBN 1-85206-203-7.
  A short description of the NAG compiler with the listing of all Fortran 90

commands and the intrinsic functions. It also contains some extensions to the standard, three complete modules, and information on mixing Fortran 90 and C.

- **NAG:** FTN90 User's Guide, July 1995. ISBN 1-85206-118-9.
  A description of NAG's compiler, linker and other utilities. In addition input/output and modules in Fortran 90 are discussed. This PC version handbook is much more complete than the one for UNIX.
- **Rama N. Reddy and Carol A. Ziegler:** FORTRAN 77 with 90: Applications for Scientists and Engineers, Second Edition, West Publishing Company, Minneapolis, 1994. ISBN 0-314-02861-7.
  Basically a textbook on Fortran 77 with Fortran 90 extensions at the end of each chapter.
- **C. Redwine:** Upgrading to Fortran 90, Springer, New York 1995, ISBN-0-387-97995-6, $ 39.95.
- **Patrick D. Terry:** FORTRAN From Pascal, Addison-Wesley, Wokingham, England, 1987. ISBN 0-201-17821-4.
  The purpose of this book is to be a textbook in Fortran 77 for the one who knows Pascal. Regrettably, it has more become a book on how to write such programs, that are in reality more suited for Pascal, in Fortran 77, e.g. simulation of recursion. Fortran ought to be used at what it is good for, large numerical or technical calculations.
- **Christoph Überhuber and Peter Meditz:** Software-Entwicklung in Fortran 90, Springer, Vienna 1993, ISBN 3-211-82450-2. DEM 60.
  The first part of this book in German discusses the foundations of numerical computing, and the second part describes Fortran 90.

David R. Wille: Advanced Scientific Fortran, John Wiley and Sons Ltd, 1995, ISBN 0-471-95383-0.
Author's comments: *Aimed at the general numerical community as a whole, it seeks to provide a stepping stone to better, more efficient and more portable programming for readers who already have a basic knowledge of Fortran. Topics covered include programming style, portability, arrays, memory management, the BLAS and LAPACK, and code optimisation. Also included are NAG, High Performance Fortran and an extensive introduction to Fortran 90.*

Author's addresses:

Bo Einarsson
Mathematics Department
University of Linköping
SE-581 83 LINKÖPING
SWEDEN
Tel. Home + 46 13 151896
Email: boein@nsc.liu.se
WWW: http://www.nsc.liu.se/~boein/

Yurij I Shokin
Institute of Computational Technologies
Prospekt Lavrentyeva 6
Russian Academy of Sciences
Siberian Division
SU-630090 NOVOSIBIRSK 90
RUSSIA
Tel: + 7 383 235 00 50, Fax: + 7 383 235 12 42
Email: shokin@adm.ict.nsc.ru