Fortran Miscellany

Fixed-format column rules

1:	C or *, comment line indicator
2-5:	statement_labels (1 to 5 digits)
6:	Any character except 0 indicates this is a continuation line.
7-72:	Fortran statement.

Anything on a line after ! (exclamation) is also a comment. Blank lines and spaces can generally be added anywhere to improve readability.

Symbolic names

First character must be a letter
Character set: letters (A-Z, a-z), digits (0-9), and underscore (_).
Limit: 31 characters.

Blank spaces and cases are generally irrelevant outside of character strings. Hence, in a symbolic_name, DOG is the same as dog is the same as DoG is the same as DO G.

Representation of literal values.

Real: digits with decimal point or E notation, optionally with sign on either the main value ("mantissa") or exponent. e.g.:

```
1.0,  -3.14159,  1E6,  5.67E-8
```

where 5.67E-8 means 5.67×10^{-8}.

Integer: digits only, optionally with sign. e.g.:

```
0, 12345, -25
```

Character: enclosed in apostrophes. Blank spaces and capitalization matter inside character strings. Apostrophes are character string delimiters, not included in the string. (To indicate an apostrophe inside a character string, type it twice together.)

```
'George', 'temperature', 'Uncle John''s Band'
```

Double precision: D notation, same as E notation but containing more precision.

```
1D6, 5.67D-8, 2D0
```

Logical: Only 2 values exist. Many computers will output the two values as simply T or F, but the normal, standard representation of these values in code is .TRUE. or .FALSE. (case insensitive).

Complex: Complex numbers are represented as an ordered pair of real numbers, contained in parenthesis and separated by a comma. For example, $3.1 + 2.4i$ would be represented as (3.1,2.4).

Fortran Declarations

Miscellaneous declarations

PROGRAM - Should be the first statement of a program; it is used to name the program, which has no effect on the execution of the program.

IMPLICIT NONE - Should be the second statement of a program; causes the compiler to write an error message for any variable name it encounters that has not been declared in a type statement. Useful for catching typing mistakes. In the absence of IMPLICIT NONE, the compiler will *assume* a type of INTEGER for any variable whose name starts in I, J, K, L, M, or N, and REAL for any variable whose name starts with any other letter.

PARAMETER - Sets a variable name to have a constant, unchangeable value. Once a name has been set in a parameter statement, it cannot be on the left side of = in a replacement statement or in the input list of a READ statement. PARAMETERS can be of any type; the PARAMETER statement for a particular variable must *follow* the type declaration for that variable.

END - Must be the last statement of a program. Blank lines after the END statement may cause the compiler to report a program that has nothing but comment lines, indicating that it tried to compile the blank lines as if they were a second program.

SUBROUTINE - Declares the name of a SUBROUTINE module. Must be followed by a *symbolic_name* for the routine and a *dummy_argument_list* in parentheses.

FUNCTION - Declares the name of a user-defined FUNCTION. Must be preceded by a *type_name* for the function, and followed by the *symbolic_name* and the *dummy_argument_list*. On exit from the FUNCTION the symbolic name should be given a value and none of the dummy arguments should be changed.

EXTERNAL - This is a multi-purpose statement. However, some compilers implement IMPLICIT NONE by requiring that the names of all SUBROUTINES and user-defined or library FUNCTIONS called in a program be declared in an EXTERNAL statement.

Type Declarations

CHARACTER - Character strings. Lengths may be declared for the entire statement or for individual names. For example

```
CHARACTER*5 MONTH, DAY, YEAR
```

declares three character strings that are each 5 characters long, whereas

```
CHARACTER HOUR*4, STATION*12, TIME*7, INLAND
```

declares four character strings that respectively contain 4, 12, 7, and 1 characters. (Note that no length designation implies a length of 1 character.)

REAL - Floating point numbers of short precision. These will be stored in an analogue of the scientific notation shown previously, with a mantissa and an exponent.

INTEGER - Whole numbers, which have a limited range but will be stored accurate to the "ones" place.

DOUBLE PRECISION - Floating point numbers which may be handled the same as REAL, but using twice as much computer memory to store. The extra memory is used mostly to improve accuracy. On most Unix-based computers, REAL numbers have about 6 digits of accuracy, whereas DOUBLE PRECISION numbers have about 14 digits of accuracy.

LOGICAL - Variables that may only take on the two values TRUE or FALSE. These may be used to store the result of a logical expression for later use. Not used in this course, but fairly useful.

COMPLEX - Two-part floating point numbers with characteristics of the "real" and "imaginary" parts of a general complex number. Used in relatively advanced mathematical applications.

Array designations

Any declared variable name becomes an array if provided with an index range, as in the general declaration of a one-dimensional array:

type_name symbolic_name (*lower_bound* : *upper_bound*)

Index values (*lower_bound* and *upper_bound*) must be INTEGER constants in a program. They may be variables only if setting the range for dummy arguments inside a FUNCTION or SUBROUTINE. The most common circumstance is that *lower_bound* is omitted and is assumed equal to one.

Up to seven dimensions may be included, each with a unique index range. For example, the array declaration

```
REAL A(10), B(-5:5), C(0:3,4), D(-100:1,1931:1990)
```

allocates REAL arrays with 10, 11, 16, and 6000 elements, respectively.

When declaring an array of character strings, the length of each string follows the array declarations. Thus

```
CHARACTER STATION(100)*12
```

declares STATION to be an array consisting of 100 character strings, each of which is 12 characters long.

Within the executable portion of a program, references to an array must always include index references that choose a particular value, rather than the entire array, with one exception: including an array name in the input list of a READ statement or the output list of a WRITE statement implies that every value should be read or written in the sequence stored.

Fortran Arithmetic

Replacement statement form

variable name to be changed = *arithmetic expression*

The arithmetic expression *must* be on the right.

Basic operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation (i.e., x^2 is written as `X ** 2`)

Order of precedence

Function calls are evaluated first.

( ) Subexpressions inside parentheses are evaluated before things outside parentheses. Parentheses can be nested.

** All cases of exponentiation are evaluated next, right to left.
`X ** Y ** Z` is evaluated as `X ** (Y ** Z)`

* / Multiplication and division are evaluated next, left to right.
`A / B * C` is evaluated as `(A / B) * C`

+ - Addition and subtraction are evaluated last, left to right.
`A * B + C * D - E` is `(A * B) + (C * D) - E`

Integer Arithmetic

Operations involving two integers can only have a whole number result. These are obtained by ***truncating*** any result of any division *towards zero*.

```
 4 / 5 --> 0          5 / 4 --> 1
-3 / 2 --> -1         27 / (-10) --> -2
```

Mixed Mode Arithmetic

If a REAL variable and an INTEGER variable are mixed together in a binary calculation, the result will be REAL.

```
4.2 * 2 --> 8.4
```

Mixed mode arithmetic is considered a **bad idea** *except* with raising a real number to an integer power, as in

`A ** 2` is better than `A ** 2.0`

Fortran Control Structures

STOP -- End program execution and return to the operating system.

```
STOP
```

DO loop -- Repeat a block of code using a controlled count

```
DO do_index = starting_value, ending_limit, increment
      .
      .
END DO
```

where *do_index* is an integer variable that counts the passes through the loop, *starting_value* is an integer expression to which *do_index* will be set first, when *do_index* goes beyond *ending_limit* the loop terminates, and *increment* is the amount that *do_index* is increased on each pass through the loop. Statements between DO and END DO are executed each pass through the loop.

In the most common loops, *increment* is not included, but is assumed equal to one.

```
DO K=2,5
```

starts a loop that will be executed four times, where K will take the values 2, 3, 4, and 5 in sequence.

```
DO I=1,5,3
```

starts a loop that will be executed twice, with I taking values 1 and 4. Note that the index value need not actually take on the value of *ending_limit.*

```
DO J=3,1,-1
```

will execute three times (J having values 3, 2, and 1 in sequence), whereas

```
DO J=3,1
```

starts a **zero-pass** DO loop in which the code between the DO and the END DO is never executed because the *starting_value* of J is already past the *ending_limit*.

The preceding examples have all used literal integer constants as loop control information, but any INTEGER expression is legal. For example,

```
DO J=N,K+2,I/2
```

represents a legal DO statement in which the number of passes, if any, can only be determined by knowing the values that N, K, and I will have at the time the DO statement is actually executed.

The *do_index* variable can not be modified by any READ or arithmetic statement within the range of the loop. It can be used in any way, so long as it is not changed.

IF blocks -- Make choices among which blocks of code will actually be run.

```
IF ( logical_expression_1 ) THEN
     .
     .
ELSE IF ( logical_expression_2 ) THEN
     .
     .
... (as many ELSE IF blocks as needed)
ELSE
     .
     .
END IF
```

Logical expressions (see next page) can be evaluated to one of two values: TRUE or FALSE. An IF block evaluates the logical expressions in sequence, executes the code following the first TRUE value it encounters, and then jumps out of the IF block to the code following the END IF statement.

One can include any number of ELSE IF branches, or none, but only the code following the first TRUE expression will be executed. The ELSE statement precedes a block of code that will be executed if none of the preceding logical expressions are TRUE Only one ELSE statement can be included in an IF block, and it must follow all the ELSE IF statements. Examples:

```
IF ( logical_expression ) THEN
     .
     .
END IF
```

contains no ELSE or ELSE IF clauses, so the entire block will be run if *logical_expression* is TRUE, and none of it will be run if *logical_expression* is FALSE.

```
IF ( logical_expression_1 ) THEN
     .
     .
ELSE IF ( logical_expression_2 ) THEN
     .
     .
END IF
```

This block allows three possibilities:

1) If *logical_expression_1* is TRUE, the code between IF and ELSE IF will be run, *logical_expression_2* will never be evaluated, and the code between ELSE IF and END IF will not be run regardless of the value of *logical_expression_2*.

2) If *logical_expression_1* is FALSE, then *logical_expression_2* will be evaluated. If it is true the code between ELSE IF and END IF will be run.

3) If neither *logical_expression* is TRUE, none of the code between IF and END IF will be run.

```
IF ( logical_expression ) THEN
     .
     .
ELSE
     .
     .
END IF
```

This block allows only two possibilities.

1) If *logical_expression* is TRUE, the code between IF and ELSE will be run.

2) If *logical_expression* is FALSE, the code between ELSE and END IF will be run.

Logical expressions

Logical expressions normally consist of comparisons, in which one numeric value is compared to another and found to be equal, not equal, greater than, and so forth, or in which one character string is found to be equal or not equal to another. The **logical operators** consist of

(old)		(new)
.EQ.	is EQual to	==
.NE.	Not Equal to	/=
.LT.	is Less Than	<
.LE.	is Less than or Equal to	<=
.GT.	is Greater Than	>
.GE.	is Greater than or Equal to	>=

Each of the following poses a question to the computer, such as "Does A equal B?" in the first case. If the answer is yes, the expression takes on the value TRUE.

```
A .EQ. B
ANS .EQ. 'y'
```

```
G .GT. 0.0
I .LE. 100
```

Additional logical operators can be used to string together expressions.

.AND.	Logical "and"
.OR.	Logical "or" (equivalent to "and/or")

Using these connectors:

```
A .GT. 0.0 .AND. A .LT. 1.0
```

would be true only if A is between 0 and 1, and

```
ANS .EQ. 'Y' .OR. ANS .EQ. 'y'
```

would be true if ANS is either uppercase or lowercase y.

Single-statement IF -- If one has a simple IF block of the form

```
IF ( logical_expression ) THEN
     one executable_statement
END IF
```

then one can delete the THEN and END IF, reducing the three-statement block into a single statement:

```
IF ( logical_expression ) one executable_statement
```

GO TO statement -- direct transfer to another part of the program.

```
GO TO statement_label
```

where *statement_label* is a literal positive integer that is used to "label" some other line of the program. A statement_label is a positive number of five or fewer digits placed in columns 1 through 5 of the Fortran statement. It is an arbitrary number--it does not have to refer to line number, nor do statement_labels in a program have to be in increasing order. They must simply be unique within a program unit.

```
GO TO 100
```

requires that somewhere else in the program, there is a statement of the form

```
100 executable_statement
```

and execution of the GO TO statement transfers control directly to the statement_labeled 100. Transfers may be forward or backward within a program. One may use a GO TO to get out of a DO loop or an IF block, but one may not use GO TO to enter one of these other blocks.

DO WHILE loop -- looping based on a logical_expression evaluated at the top of the loop.

```
DO WHILE ( logical_expression )
     .
     .
END DO
```

Every time control reaches the DO WHILE statement, *logical_expression* will be evaluated. If it is TRUE, code between DO WHILE and END DO will be executed and the control will return to the DO WHILE for re-evaluation of the *logical_expression*. If *logical_expression* is FALSE, control will transfer out to the first statement after the END DO. As with GO TO, this structure can generate infinite loops.

Fortran Input/Output

Statements

`READ` -- Input information into the computer from an external device. Its usual form is

`READ (` *unit_number* `,` *format_information* `)` *input_list*

where *unit_number* is any positive integer (some computer systems restrict the range, but Unix systems typically do not), *format_information* is discused below, and *input_list* is a list of variable names that can be filled with information from the external source that is being read. Variables in the *input_list* will be changed by the `READ` process, so they must be unrestricted variables: they cannot be `PARAMETERS`, `DO` indexes, or literal constants. For example,

```
READ (5,*) TIME, DATE
```

An optional third slot is useful when reading from files:

```
READ (10,*,END=100) input_list
```

"jumps" to a statement labeled `100` if one tries to read the end of a file. This is very useful if a program does not know in advance how many lines a file includes.

`WRITE` -- Output information from the computer to an external device.

`WRITE (` *unit_number* `,` *format_information* `)` *output_list*

output_list is more flexible than *input_list*, because information in the *output_list* is not changed by the `WRITE` process. Thus, `PARAMETERS` and literal constants may be included. For example,

```
WRITE (3,*) 'The answer is ', X
```

`OPEN` -- Connect a file (or other device) to a particular unit number. (Unit 5 is connected to the keyboard and unit 6 is connected to the terminal screen by default.)

`OPEN (` *unit_number*`, FILE=` *file_name* `)`

file_name is a character string, specified either as a literal or as a character variable. Its exact nature will vary with the operating system being used. On Unix, it may include a complete or partial path. Examples:

```
OPEN (3,FILE='census.data')

OPEN (12,FILE='~/census/1990/delaware/sussex.data')

CHARACTER*20 CENSUSFILE
READ (5,*) CENSUSFILE
OPEN (3,FILE=CENSUSFILE)
```

CLOSE -- Disconnect a file from a particular unit number.

```
CLOSE ( unit_number )
```

For example, CLOSE (2) will disconnect unit 2 from whatever file or device to which it was connected. This is only needed if the unit number will subsequently be reconnected to another file. Otherwise, all connections are broken at the end of program execution.

REWIND -- Restart reading a file from the top. (Only usable with files that are being read.)

```
REWIND ( unit_number )
```

BACKSPACE -- Move the file position pointer back one line, so that the previously read line can be re-read. Useful for fixing error conditions.

```
BACKSPACE ( unit_number )
```

Format information

Format information controls how information is translated between the binary codes stored within the computer and the character-based information typically dealt with on external devices, such as keyboards, terminal screens, and files. Formats always consist of lists of *edit_descriptors* contained within parentheses. These may be specified as character strings (either literal or character variables) or on separate FORMAT statements that are tied to the READ or WRITE statement by reference to a statement label.

The default format obtained by using * instead of a format description will be adequate for many WRITE statements and nearly all READ statements. Normally, unformatted READ statements are preferable because they are less error-prone than formatting. However, formatted READ may be made necessary by various data-compression schemes and by a need to skip information on a line.

```
WRITE ( unit_number, '(list of edit descriptors)' ) output_list

WRITE ( unit_number, character variable name ) output_list

WRITE ( unit_number, statement_label ) output_list
statement_label FORMAT ( list of edit descriptors )
```

Edit descriptors

In the descriptions that follow, lowercase bold-italic letters must be replaced by integer constants in actual code. I.e., F*w.d* must be replaced with something like `F10.2` in actual code. Integer **variables** can **not** be used to control format edit descriptors.

If more width than necessary is provided for a numeric value, the number will be right justified, with blanks added to the left to occupy the requested width. If extra width is provided to a character variable, it will be left justified, with blanks added to the right.

Input and Output edit descriptors

*n*X — Skip *n* spaces.

F*w.d* — Read or write a floating point (`REAL` or `DOUBLE PRECISION`) number, using *w* spaces (width) and leaving *d* digits after the decimal point. Width must include places for a decimal point and, if needed, for a sign. If reading a number, the decimal point need not be included, but will be inferred by the format.

I*w* — Read or write an `INTEGER` number in *w* spaces, including space for a sign if needed.

A*w* — Read or write a character string constant or variable (alphanumeric information), using *w* spaces.

A — Read or write a character string constant or variable, using whatever width is needed to accomodate the character string.

Output edit descriptors associated with output list items.

G*w.s* — Write floating point information in the general format, using *w* spaces and writing at least *s* significant figures. *w* must be at least 7 greater than *s* to accomodate the most general format. The computer will choose whatever `F` format is best for the actual number, but if no `F` format works, it will automatically go into exponential format. `G` is the most useful format when you have no idea what magnitude of numbers might be produced. For example, if `X=-1234000.0` then

```
WRITE (6,'(G11.4)') X
```

will produce as output `-0.1234E+07`.

Literal character strings can be included within formats intended for output. If the format is already a character string included in a `WRITE` statement, then a doubled apostrophe must be used to delimit included literal character strings.

I*w.n* — Write an `INTEGER` number in *w* spaces, and fill in the front of the number with zeros if needed to fill at least *n* nonblank spaces. Useful in such applications as clock times. For example, if `HOUR=12` and `MINUTE=5` then

```
WRITE (6,'(I2,'':'',I2.2)') HOUR, MINUTE
```

would produce `12:05`, whereas using a second `I2` would produce `12: 5`.