Handout Three

September 25, 2013

1 Arithmetic Expressions with Integers & Reals

Writing arithmetic expressions in programming languages like Fortran is mostly intuitive and does not cause many problems. However, there are situations that can occur where uncertainty arises and so you need to be aware of certain rules that exist to remove this.

1.1 Expression Evaluation

In section 2.2 of handout two the intrinsic numeric operators and their order of precedence were listed in a table, for convenience the table is given below. Note that the '`/`' and '`*`' operators have the same order of precedence. Lower in the table the dyadic minus (`-`) operator has the same level of precedence as the plus (`+`) operator .

[1]	**	: Exponentiation operator, dyadic
[2]	/	: Division operator, dyadic
[2]	*	: Multiplicative operator, dyadic
[3]	-	: minus operator monadic for negation
[4]	+	: additive operator, dyadic
[4]	-	: minus operator, dyadic

An example was given to demonstrate how the order is important when writing down numeric expressions in Fortran. The 'left to right' rule was introduced for cases where there was ambiguity as to which order to evaluate sub expressions involving operators with the same level of precedence. There is however '**one**' exception to this 'left to right' rule. The exception involves a special case of the exponentiation operator. The exponentiation operator has a precedence 'right to left'. For example consider the expression

```
  2  1
a**b**c
```

The numbers above the operators indicate the order in which that operation will be evaluated. First '`b**c`' would be calculated and the result used as the exponent in the operation with '`a`'. Consider the following two lengthy expressions, again, above each of the operators is a number identifying the order in which that operator is executed during the calculation of the overall expression.

```
 6 3  2  1 4 7 5
a-b/c**d**e*f+g*h
```

```
 6 3  1    2 4 7 5
a-b/(c**d)**e*f+g*h
```

For the expression on the left, the first calculation is '`d**e`' and the result used as the exponent in the exponentiation operation on '`c`' then the result divided into '`b`' etc. and so on. Work through the above expression and make sure you understand it before moving on! Also, although it is important that you know about correct 'expression evaluation' if you are in doubt you can use parentheses `()` to remove any uncertainty! See how for the expression on the right the inclusion of parentheses around the '`c**d`' alter the default order of the 'right to left' rule for exponentiation. **Remember** sub-expressions inside parentheses have the highest precedence so will be evaluated first!

NOTE : Two numeric operators cannot appear side by side in an expression. For example '`4*-3`' would fail during compilation. You need to include parentheses around the monadic expression : '`4*(-3)`' . You could however have written '`-3*4`' .

1.2 'INTEGER' Division

There is a potentional pitfall with integer arithmetic and it occurs when dividing one integer into another. Consider the following few lines of code and try to predict what you think will be printed to the screen.

```
PROGRAM integer_test
!**
!** Program to highlight problems in integer arithmetic
!**Gerardo

  IMPLICIT NONE

  INTEGER, PARAMETER :: a=1, b=2
  INTEGER :: answer1
  REAL :: answer2,answer3

  answer1=a/b
  answer2=a/b
  answer3=REAL(a)/b

  PRINT*,"When evaluated as integer =",answer1
  PRINT*,"When assigned to a real   =",answer2
  PRINT*,"When a is casted first    =",answer3

END PROGRAM integer_test
```

Exercise One: In a directory '`handout3/exercise1`' Code up the few lines above and run the compiled executable. The result may not be what you expected to see. The reason is that any numeric operation built up from a 'numeric intrinsic operator' '`( **,*, -,+)`' and operands of the same intrinsic data type will always give a result of the same data type as the two operands. Therefore dividing one integer by another integer must result in an integer. This is why in integer arithmetic the operation '`1/2`' gives an answer of zero, even when the result is assigned to a data type that has been declared as a `REAL`. NOTE that for integer division the result is simply 'truncated', so the fractional part is removed, it is not 'rounded' to the nearest integer.

1.3 Data type casting

Note that a single 'dyadic operation' is defined as the result of the evaluation of an operator and its two operands, ie. '`a/b`', here '`a`' and '`b`' are the operands and the operator is division. In fact in Fortran the only monadic operator is the negation operator ('`-`') as in '`b=-a`'. So for the purpose of discussing the casting of an 'operand' the associated operator can always be assumed to be dyadic.
When you use numeric expressions in your codes that involve different data types (referred to as mixed mode expressions) Fortran follows given rules to ensure consistent evaluation of such expressions. In a mixed mode expression Fortran will, if required, automatically 'convert' data types in an arithmetic operation to allow it to evaluate that expression. Remember that in order for an operation to be evaluated the operands must be of the same type. 'Casting' means that, if the two operands are of different type,

one of the operands will temporarily converted to the operand of more general type in the operation. The order of casting is

INTEGER → REAL → COMPLEX

So if '`a`' is declared '`INTEGER` and '`b`' is declared '`REAL`' then the operation '`a/b`' would result in a '`REAL`'. This is because '`a`' would temporarily be cast to a '`REAL`' for the purpose of the operation. We can also force an expression or operand to result in a certain data type by using the intrinsic functions '`INT()`', '`REAL()`' and '`CMPLX()`', see the next section.

2 Intrinsic functions

Fortran, as with other programming languages, has a number of "intrinsic functions". Intrinsic functions are built into the Fortran language. The conversion between real and integer using '`REAL()`' or '`INT()`' are examples. One can take the sine of a real variable '`x`' and assign the answer to '`y`' by typing

```
y = SIN(x)
```

In this case '`x`' is called the 'argument' of the `SIN` function. Some of the functions you should be aware of are;

The Trigonometric Functions

`SIN(x)`	: Returns the sine of `x`
`COS(x)`	: Returns the cosine of `x`
`TAN(x)`	: Returns the tangent of `x`
`ASIN(x)`	: Returns the arcsine of `x`
`ACOS(x)`	: Returns the arccosine of `x`
`ATAN(x)`	: Returns the arctangent of `x`
`ATAN2(x,y)`	: Returns the arctangent of a complex number $x+iy$
`SINH(x)`	: Returns the hyperbolic sine of `x`
`COSH(x)`	: Returns the hyperbolic cosine of `x`
`TANH(x)`	: Returns the hyperbolic tangent of `x`

Other Mathematical Functions

`EXP(x)`	: Returns the e raised to the power of `x`
`LOG(x)`	: Returns the natural logarithm of `x`
`LOG10(x)`	: Returns the logarithm to the base 10 of `x`
`SQRT(x)`	: Returns the square root of `x`
`ABS(x)`	: Returns the absolute of `x`
`INT(x)`	: Returns an integer part of `x` by truncation
`REAL(x)`	: Returns the conversion of `x` to be data type `REAL`
`CMPLX(x)`	: Returns the conversion of `x` to data type `COMPLEX`
`AIMAG(x)`	: Returns the imaginary part of the complex data type `x`.
`NINT(x)`	: Returns the nearest integer to `x`
`MAX(x,..)`	: Returns the maximum of a comma separated list
`MIN(x,..)`	: Returns the minimum of a comma separated list

In functions such as `SIN(x), COS(x)` etc. the argument `x` is to be given in radians. Note that these functions return a value so will behave just like any other operand if used in an arithmetic operation ie '`x**2+sin(x)`'.

3 A Few Odds and Ends

3.1 Data Type Limitations

Fortran has to store all numbers internally in the computers memory. It is inevitable therefore that there will be restrictions in the numeric range of the various data types. We will not go into detail here but simply present the values to you. For the Linux machines you are working on

'`INTEGER`' : -2147483648 $\rightarrow$ 2147483647
'`REAL`' : $-10^{38} \rightarrow 10^{38}$

Any numerical calculations in your codes should not exceed these limits.

3.2 Complex arithmetic

To declare a complex data type in Fortran you give the real part and the imaginary part separated by a comma in brackets. ie.

```
COMPLEX ::  cmp1=(4,3)
```

There are a few intrinsic functions that are implemented specifically for complex numbers they are;

`AIMAG(c)`	: Returns the imaginary part of '`COMPLEX`' type '`c`'
`CONJG(c)`	: Returns the complex conjugate of '`COMPLEX`' type '`c`'
`CMPLX(x)`	: Returns the conversion of '`x`' to data type '`COMPLEX`'

To extract the real part from a complex number use the '`REAL`' function.

Exercise Two : Rewrite your Quadratic program making use of the data type '`COMPLEX`' so that you do not need to branch on the value of the discriminant for each of the three different types of root. Do this in the directory '`handout3/exercise2`'.

3.3 The true meaning of '=' in Fortran

In Fortran, the assignment operator ('`=`') instructs the compiler to generate code that evaluates the expression on the right and then assign the result to the variable on the left. A variable name is actually a pointer to a specific location in a computer's memory where the value of the variable's data type is stored. The operation '`m=m*n`' does the following,

1. Read the contents of locations '`m`' and '`n`' from memory.
2. Multiply '`m`' by '`n`'.
3. Assign the result of '`m*n`' to be stored in the memory location '`m`'. Similarly '`n=n+1`' will increase the value stored in '`n`' by 1.

3.4 Blank spaces and continuation lines

Notice that there are spaces between the operators and their operands in the statement '`m = m * n`' yet no extra spaces in '`m=m*n`'. You are at liberty to use as much 'spacing' as you think is necessary to make

the line easier to read. The Fortran compiler ignores these spaces. Like the indentation of new lines they are for the benefit of the programmer. In all programming languages it is considered good practice to keep codes well structured and well set out by making use of consistent spaces and indentations.
If a line of Fortran code gets so long that it doesn't fit on a line in the '*nano*' editor then you need to use the 'line continuation character'. This involves ending the line you are typing with the '`&`' symbol. The next line of text is then taken as a continuation of the previous line e.g. the two lines

```
a = a + b + c + d + e  &
  + f + g
```

are treated by Fortran 90 as though they are the single line

```
a = a + b + c + d + e + f + g
```

Notice that all of the spaces at the start of the second line have also been included in the new extended line. Since F90 ignores these spaces this isn't a problem here. This would be a problem if the line you are continuing is part of a character string. This is one of the few places where F90 doesn't ignore spaces. In this case you *must* place an `&` before the text on the second line as well as at the end of the line you wish to continue e.g.

```
PRINT*,"If you want to split a sentence onto more than &
        &one line then you should use the line continuation &
        &character like this."
```

This forces the compiler to ignore the extra spaces. If you have long strings then you must remember to always have the second '`&`' otherwise your program will not compile.

3.5 Simple Input and Output

In their most simple form the '`READ*`' and '`PRINT*`' statements allow us to read in data from the default input device, normally the keyboard, and output data to the default output device, normally the display.

3.5.1 The list directed 'PRINT*' command

The 'PRINT*' command has the simplified form,

PRINT*,<*expr1*>[,<*expr2*>,<*expr3*>....]

some examples of the 'PRINT*' command are;

```
PROGRAM example
! ** Demonstrate some PRINT statements to the screen

  IMPLICIT NONE

  INTEGER :: a=3,b=7,mult,add,minus
  REAL    :: divide

  mult=a*b
  add=a+b
  minus=a-b
  divide=REAL(a)/b !** Note the forced cast of INTEGER to REAL

  PRINT*,"3*7=",mult
  PRINT*,"3+7=",add
  PRINT*,"3-7=",minus
  PRINT*,"3/7=",divide
  PRINT*,"3^2=",a**2 !** Expressions can be evaluated in a PRINT statement
  PRINT*,"7^2=",b**2

END PROGRAM example
```

From the example code you can see that strings (the text inside quotes) can be output straight to the screen along with numeric variables or expressions. Note the use of the 'REAL' function to convert one of the integers in the operation to a real value so the result returned will be the correct fractional value.

3.5.2 The list directed 'READ*' command

The list directed 'READ*' command has the simplified form;

READ*,<*var1*>[,<*var2*>,<*var3*>....]

It is used for 'reading' in information from the keyboard to the program. The following small piece of

code demonstrates how the command works.

```
PROGRAM test

  IMPLICIT NONE

  INTEGER :: a,b,c

  PRINT*,"Please input a"
  READ*,a
  PRINT*,"Please input b"
  READ*,b
  c=a*b
  PRINT*,"a*b=",c
END PROGRAM test
```

Exercise Three: A radioactive isotope of an element is a form of the element that is unstable. It spontaneously decays into another element over a period of time. This process of decay is exponential. If 'Q_0' is the initial amount of radioactive substance at time '$t = 0$' then the amount of radioactive substance at any time 't' in the future is given by,

$$Q(t) = Q_0 e^{(-\lambda t)}$$

Where 't' is the time in years and λ is the decay constant for carbon-14 : 0.00012097 per year. When a living being dies the amount of carbon-14 in the body decays from its constant value during life. We can measure the amount that has decayed and hence estimate the time since death. To do this we assume we know the normal amount in such a living being and we can measure the amount in the current sample and so calculate the fraction or percentage left from the time of death. The above equation can be rearranged to give,

$$t = -\frac{1}{\lambda} \log \left(\frac{Q}{Q_0}\right)$$

Write a computer code to calculate the age of a sample of material. Input from the keyboard the amount of radioactive material left in the sample as a percentage of the original and use this to output its age.

.

Exercise Four: The Maclaurin (or Taylor) expansion for $\sin(x)$ about zero gives the series

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - \ ....$$

Write a program to calculate the approximation to '$\sin(x)$' given by the above series expansion. Use the '`READ*`' command to input the value of '`x`' to approximate and the number of terms 'n' you would like to use for the approximation. Use an indexed '`DO`' construct that starts looping at '`3`' with a 'stride' of '2' to give you the odd powers of 'x'. Build up the factorial inside the '`DO`' loop structure as you iterate over the expansion with a statement '`fact=fact*(???)`'. Replace the '`???`' as appropriate. At the end of your code print out your results as follows

```
PRINT*,'The approximation is for x=',x
PRINT*,'Number of terms in the approximation = ',n
PRINT*,'The approximation =',sinx
PRINT*,'The true value is =',SIN(x)
```

where the variable '`sinx`' is your approximation. Experiment with various values of 'x' and the number of terms 'n' in the approximation however use no more than seventeen terms.

PART TWO : Can you identify a potential problem with the way in which the factorial is being built up and hence stored as the loop iterates. Can you think of a way to alter the program such that the factorial is not explicitly calculated but its value is implicitly factored into in the summation.