Chapter 2: The Program

Let's jump right into Fortran programming. We start with an introductory example to learn program structure, some basic Fortran syntax, and how to compile/run a Fortran program.

2.1 Example: Adding Two Integers

Below we see the pseudocode for a simple program that adds two fixed integers.

Algorithm 1: *A First Example*: Add two integers and print

Procedure A First Example
$i \leftarrow 3$
$j \leftarrow 7$
$k \leftarrow i + j$
Print k
End Procedure A First Example

Now we translate this pseudocode into workable code, which we save as the program "firstExample.f90'

firstExample.f90

```
PROGRAM firstExample
   IMPLICIT NONE
   INTEGER :: i,j,k

   i = 3
   j = 7
   k = i + j
   WRITE(*,*)k
END PROGRAM firstExample
```

A few quick notes on this simple example:

1. An executable source file will start with PROGRAM PROGRAM_NAME and end with END PROGRAM PROGRAM_NAME
2. Fortran is a sequential language, thus we always declare variables, like INTEGER at the top of the program
3. The variable declaration will always indicate the type followed by two colons and then the variable name, e.g., INTEGER :: `i,j,k`
4. We'll look at more on I/O in Chap. 3, but for now know that in the WRITE statement (1) the first '*' means 'print to terminal' and (2) the second '*' means "list-directed", where the formatting is chosen by the compiler. The PRINT* command is another way to produce a list-directed write to the terminal screen.
5. Again, it is very important to be aware that Fortran is ***not case sensitive***. So the output for firstExample.f90 would be the same, for example, if we had written $J = 7$.

It is important to note that just after the program is initialized we will ***always include the statement*** IMPLICIT NONE.

```
PROGRAM function˙Name
   IMPLICIT NONE
   ...
```

This statement makes the use of undeclared variables illegal, and so a compiler error will result if we forget to declare a variable (what we want to happen). In Fortran, any undeclared variables is of the type single precision.

Also, the Fortran compiler will implicitly assign data types depending on what letter a variable begins with (a-h & o-z are `real` and i-n are `integer` types). This can lead to bugs and lots of headaches, so we always turn off this feature by including IMPLICIT NONE.

Once we save our simple program as firstExample.f90, we are ready to run. We open a terminal, move to the correct folder where our source file is located, and then type on the command line:

firstExample.f90 - Commands and Output

```
gfortran firstExample.f90 -o firstExample
./firstExample
        10
```

We decode what was just done to gather some general information on how to run Fortran programs:

1. When we run the compiler (the `gfortran` command above) it creates an executable file
2. The compiler flag `-o` lets us name the executable, in this case 'firstExample'. If you ***don't*** use this flag the executable has the default name 'a.out'
3. To run the executable from the command line we use the ./command

2.2 Example: Adding Two Real Numbers (in Double Precision)

Let's do another quick example to reinforce the Fortran syntax we just learned. Also, we will introduce the common Fortran syntax for manipulating real numbers. As such, we have the pesudocode for a routine that adds two (double precision) real numbers:

Algorithm 2: *Real Numbers*: Add two real numbers and print

Procedure Real Numbers
$x \leftarrow \pi$
$y \leftarrow 2.5$
$z \leftarrow x + y$
Print z
End Procedure Real Numbers

Now we translate the second set of pseudocode into workable a program. We save this program as "realAdd.f90'

realAdd.f90

```
PROGRAM realAdd
   IMPLICIT NONE
   INTEGER,PARAMETER       :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)           :: x,y,z
   REAL(KIND=RP),PARAMETER :: pi = 4.0_RP*ATAN(1.0_RP)
!  Let's add some numbers
   x = pi
   y = 2.5_RP
   z = x + y
   PRINT*,z
END PROGRAM realAdd
```

There are a few important things happening in this implementation of adding two real numbers that require in depth discussion.

1. We define the number of digits of accuracy on our real numbers in a rather unusual way. (*Note, if some of this doesn't make sense don't fret, finite precision will be covered by Dr. Gallivan's FCM I class.*) The highlight is that a double precision number has sixteen digits of accuracy, and a single precision number has seven digits of accuracy. These are the only two precisions a floating point number may be stored in most computers. The argument of the function SELECTED_REAL_KIND is the *minimum precision* we are requiring of our real numbers, so we input 15 as the argument because we want to guarantee *at least* 15 digits of accuracy. Since the hardware can only interpret single or double precision, it will interpret all real numbers defined with the

KIND=RP parameter to have sixteen digits of accuracy, or double precision. In this way, we define a parameter to indicate what is meant by "real precision" called RP. Then we specify the accuracy of the REAL data type by including KIND=RP.

There is a DOUBLE PRECISION variable type in Fortran, but ***don't use it. It is deprecated.*** The reason is because double precision means different things on different machines. However, if you specify 16 digits with the intrinsic function SELECTED_REAL_KIND, you know that no matter what machine you use that you'll always have that many digits of accuracy. Thus, it makes your code portable and easy to change.

2. What does it mean for a variable to be a PARAMETER? It means, we define a number, in this case π, that will remain constant throughout run of the program. One may be tempted to hard-code π to a certain number of digits, but on different machines with different precisions you may lose accuracy. To maintain portability of code ***try not hard code your constants***. The constant π is really the only one I use, but sometimes it may be unavoidable like with Euler's constant γ.

3. Inside the program instructions, how do we tell the compiler to treat a number as a real? If we want to the compiler to treat the number one, say, as a double precision number we must write `1.0_RP`.If we only type 1 in our program, then the compiler will treat 1 as an integer and can introduce errors.

4. What's with all the exclamation points? The exclamation point, !, acts as the comment character in Fortran. Thus, anything that appears after an exclamation point is ignored by the compiler. Comments are there to make the code more readable for the author as well as for other people. Always remember, codes can never have too many comments so be as liberal with them as you like. It will only make your life easier when you go back to read code and remember what exactly it does.

Finally, we compile and run the second example of adding two real numbers.

realAdd.f90 - Commands and Output

```
gfortran realAdd.f90 -o realAdd
./realAdd
        5.6415926535897931
```