FORTRAN 95

Connor Richardson
cbrichardson6@email.arizona.edu

Sean Callahan
seanpcallahan@email.arizona.adu

April 29, 2020

ABSTRACT

Fortran is one of the oldest living computer science languages, with a lifespan of over 50 years. With the capacity to perform computations with complex numbers and 7 dimensional matrices, as well as built in control structures to handle these unique types, Fortran proves itself to be an incredibly fast and versatile language. Although challenging to read, with less support for modularity, these are a small price to pay for the performance benefits of Fortran.

1 Introduction

Today, Fortran is known as an ancient and dead language, yet this is not entirely the case. Receiving development as recent as 2018, Fortran is still heavily used in the fields of research, where computing power and speed are key. With Fortran being as fast as C, but being able to do much more complex operations with complex numbers and Matrices spanning 7 dimensions, Fortran is an obvious choice for those in scientific research.

2 History

Fortran was initially developed by IBM to provide developers working on the IBM 704 Mainframe Computer with an alternative to assembly language. Because of hardware limitations at the time (1952-1956) Fortran prioritized performance with its the first Fortran compiler being an optimizing compiler, which brought its performance significantly closer to programs written with assembly language. Fortran's primary focus is scientific computing, primarily in use in scientific research and engineering.

Today, Fortran, although significantly less popular than in previous decades, is still a supported language, with revisions as recent as 2018. Its reach, however, has spread to encompass much more than its own development, but has also built a foundation for a number of dialects and entirely new languages, including BASIC.

3 Control Structures

Because of Fortran's age, the controls structures have shifted to accurately match common tropes in programming during each era. While the first versions supported what is known as the arithmetic if statement, which allows programmers to jump to one of three labels, with a conditional that determined which label by the sign of the operand. Modern versions of Fortran, don't encourage its use, with it finally losing full support in Fortran 2018. [3]

```
    IF ((x * 2) - 100) label1, label2, label3

label1 continue
    Print *, "Hello"

label2 continue
    Print *, "World"

label3 continue
```

```
    Print *, "!"
```

Fortran, and more specifically Fortran 95, supports various forms of the if conditional, first gaining support of the arithmetic if statement and in FORTRAN IV and Fortran 77, gaining an if conditional that is most similar to what we see in many other programming languages today. The if statement block works much like an if block in C or Java, with a conditional followed by code, but instead of the common curly braces separating the code from the conditional, this is done with the word "then" to signify the start of an if or else, and "end if" to end the contained code block. Closely related to the if statement, Fortran 95 also supports the switch case in a way very similar to that of Java. [2]

```
IF (x == 0) THEN
    Print *, "X is 0"
ELSE IF (x == 10) THEN
    Print *, "X is 10/10"
ELSE
    Print *, "X is none"
END IF
```

The primary control structure used in Fortran 95 is the do Loop, which is effectively a "while True" loop. This form of loop continuously repeats the loop until the exit operation is used, acting as a break. Curiously, however, the do loop also behaves very similarly to the for loop, allowing the creation of an iteration variable:

```
do i = 0, 100, 1
    print *, "Hello World"
end do
```

Much like list comprehensions, the do can be used to populate arrays by having what is essentially the beginning of a do loop contained in the array:

```
integer :: nums(5) =  [(i, i = 0,4)]
```

Perhaps the most interesting control structure in Fortran 95 is the where statement. Primarily used for modifying specific elements in an array, masking out elements that we don't wish to change. This simultaneously acts as a loop, while also acting as a conditional for each element in the array being iterated over.[3] An example of this would be a snippet of code that puts a min and max value on a set of numbers:

```
integer :: grades(10) = [90, -2, 100, 101, 85]
where (grades > 100)
    grades = 100
else where (grades < 0)
    grades = 0
end where
```

4 Data Types

For most of Fortran's data types they closely resemble that of C. The CHARACTER type in Fortran works as one would expect in many C variant languages, where a character is a single byte, and can be strung together into arrays in order to create strings. One key difference, however, is that the user can specify what kind of encoding is desired for the CHARACTER or string of characters. This could be ASCII, UNICODE, etc., although some compilers may not support all languages and encodings.

Much like the CHARACTER type, Fortran also supports LOGICALs, which are roughly equivalent to Booleans in Java, holding a true or false value and are declared as:

```
.TRUE.
```

Fortran also has support for data types like INTEGER and REAL, which, once again, function similarly to ints and floats that we already know. Much like C, we can specify how many bytes we want an integer to be by using the kind parameter. This also allows the programmer to specify a variable to be of double precision instead of the standard floating point precision. In order to do this, you add the statement:

```
kind = 8 /*where 8 would be the number of bytes you desire*/
```

So, to specify to initialize a variable with a C equivalent of a long long int on a 32 bit machine, we can simply say

```
integer(kind=8):: c_long_long_int
```

Where Fortran really starts to differ from C and Java is in its support of complex numbers. Storing the complex number in what is essentially a tuple, such that (a, b) is equivalent to the expression a - bi.

Fortran, since it is a mathematically focused language, does support arrays, in up to seven dimensions. These can be declared using the attribute dimension directly following the type of the array.

```
integer, dimension(10,10) :: grades_and_ids
```

5 Subprograms

Subroutines and functions in Fortran function very similarly to functions in a programming language like Python, C, or Java, with a handful of key differences. Functions are the most familiar to programmers of C and Java. They all begin with a function declaration, which names the function, as well as any parameters, which Fortran calls "dummy arguments." Every function begins with the keyword "Function," followed by the function name, and the dummy argument enclosed in parenthesis. Several key characteristics of functions are that they cannot return more than a single value or type of value, and that each return value must be of the same name. Unlike C or Java, the arguments are not typed in the declaration, but are not loosely typed as in Python either. Instead, in the first lines of the function, we declare the types of both the return value and the dummy arguments.[1] Then, functions proceed as normal, returning whatever value is stored in the retval when we reach the end of the function, denoted with "end function function_name." An example function, which simply doubles an input integer is given as :

```
function double_num (x)

    implicit none

    integer :: double_num

    integer :: x

    double_num = 2 * x

end function double_num
```

It is possible to specify return values that differ from the function name, however, this requires an additional keyword added in the declaration: "result." By adding "result" with a new return value name encapsulated in parenthesis immediately after, the returned value is whatever directly follows the result keyword. When added, the double_num program previously shown becomes:

```
function double_num (x) result (retval)

    implicit none

    integer :: retval

    integer :: x

    retval = 2 * x

end function double_num
```

Subroutines, function in a slightly different sense. Instead of returning only a single value specified to be that with the same name as the function, or alternatively named with the result keyword, subroutines can return multiple values. The primary differences are that when we declare our dummy arguments, we specify their intent: in, out, or inout. [1] If the arguments are in only, we shouldn't modify them, but if they are out only, there is no guarantee that the values will contain information that we expect when passed in. If one seeks to both take in and overwrite an argument, they should specify the intent as inout. Subroutines can be defined to be recursive by inserting the "recursive" keyword before the

beginning of the subroutine definition, which begins with "subroutine" instead of "function." A subroutine version of the double_num function can be written as :

```
subroutine double_num (x)

    implicit none

    integer, intent(inout) :: x

    x = 2 * x

end subroutine double_num
```

Fortran also supports, and sometimes requires, the use of interfaces, which explicitly define subroutine dummy arguments. Because while linking external libraries to our main program, the program does not know the contents of these libraries, interfaces prevent compile time errors by declaring these subroutines and their dummy arguments, without fully defining them. An interface for our double_num subroutine would be:

```
interface

    subroutine double_num (x)

        integer, intent (inout) :: x

    end subroutine double_num

end interface
```

6 Summary

Fortran clearly lives on today, despite falling out of use from everyday programming. And although its control structures are rather prone to spaghetti code, it counteracts its unwieldy and verbose language with its vast amount of built in structures, that are optimized for mathematical calculation.

References

[1] Fortran manual. http://physik.uibk.ac.at/hephy/praktikum/fortran_manual.pdf. Accessed: 2020-04-29.

[2] John W. Backus. The history of fortran i, ii, and III. https://doi.org/10.1109/85.728232, 1998. Accessed: Sun, 02 Jun 2019 21:01:39 +0200.

[3] Tanja van Mourik. Fortran95 manual (fith revision). http://www-eio.upc.edu/lceio/manuals/Fortran95-manual.pdf, 2005. Accessed: 2020-04-29.