Tyrone Xue
Principles of Programming Languages, CPSCI 220
Fortran 2018 - Term Project Part 3
6 May, 2024

Introduction

My name is Tyrone Xue. In this paper, I have chosen to write about the programming language Fortran, specifically all the way up to the Fortran 2018 iteration of it. I will be covering the background of Fortran which include its historical influence as the first implemented high-level programming language and its common uses within scientific computing. I then delve into the various aspects of Fortran such as its multi-paradigm features, implicit and static type characteristics, its syntax and semantics, common data types, and some interesting features that's included up to Fortran 2018. I will then proceed to explain additional information about Fortran such as a comparative analysis between Fortran and Ruby. Finally, I will describe the community support of Fortran and its development environment before concluding the report.

Fortran Background

Historical Influences

Fortran, short for mathematical FORmula TRANslation system, is a programming language proposed and created by John Backus in order for a computer to understand human intent [4]. The language was commercially released in 1957 alongside the IBM 704 Data Processing System which made computer programming more accessible. Prior to Fortran, programming was done in either machine language (gen 1), in which code is written in binary, or assembly language (gen 2) which uses mnemonics that stands for machine code. This allowed computer programming to be used by engineers and scientists because of how readable Fortran was as compared to the gen 1 and gen 2 programming languages which were only tailored for computer programming specialists [3].

In order for Fortran to be competitive with other programming languages, it has gone through numerous updates that shaped Fortran to what it is today [3]. Some notable releases are FORTRAN 66, FORTRAN 77, Fortran 90, and the different iterations of modern fortran up to Fortran 2023 (we will only cover features up to Fortran 2018 in this report).

FORTRAN 66 was the result of the American Standards Association's (now ANSI) decision to develop the American Standard Fortran committee. This led to the first standard version of Fortran which contained features like `DO` loops, `SUBROUTINE` and `FUNCTION` program units, data types like integers, reals, and complex values, and more [8].

FORTRAN 77 addressed some shortcomings with 66 and added `IF` statements and the ability to directly access files through I/O [8].

Fortran 90 provided significant changes such as inline comments, modules, recursive procedures and more without removing any existing features [8].

Modern Fortran, starting from 2003, introduced features that allow for object-oriented programming which could be done through modules, as opposed to classes in languages like C++ and Java [8].

Fortran 2008 included new features like submodules, coarrays alongside improvements to data usage, I/O, and execution control such as `EXIT`. Finally, Fortran 2018 involved a minor revision of Fortran 2008 which included further interoperability with C and additional parallel features like teams [9].

Fortran has inspired many programming languages such as C and Basic. Fortran was established as the first, implemented high level programming language. It introduced modular programming with subroutines and functions, allowing the reuse of code. And its concise mathematical notation influenced the syntax of subsequent programming languages.

Common uses

Today, Fortran is commonly used for scientific computing. I identified two main reasons for this. The first is Fortran's high performance computing capabilities, especially its support for arrays such as multidimensional arrays and array-based calculations [9]. To exemplify this, I created a program that calculates the reduced row echelon form (rref) of a matrix, a concept from linear algebra. The rref is the row echelon form (ref) of a matrix in which all pivots are leading ones (equal to 1) and are the only non-zero values in their column. The ref is a matrix in which there are only zeros below the pivots of the matrix, alongside other rules that define a ref of a matrix. Operations called row reductions are applied to the matrix to reach the ref. The ref is a matrix converted to a simplified form (but maintains its important properties).

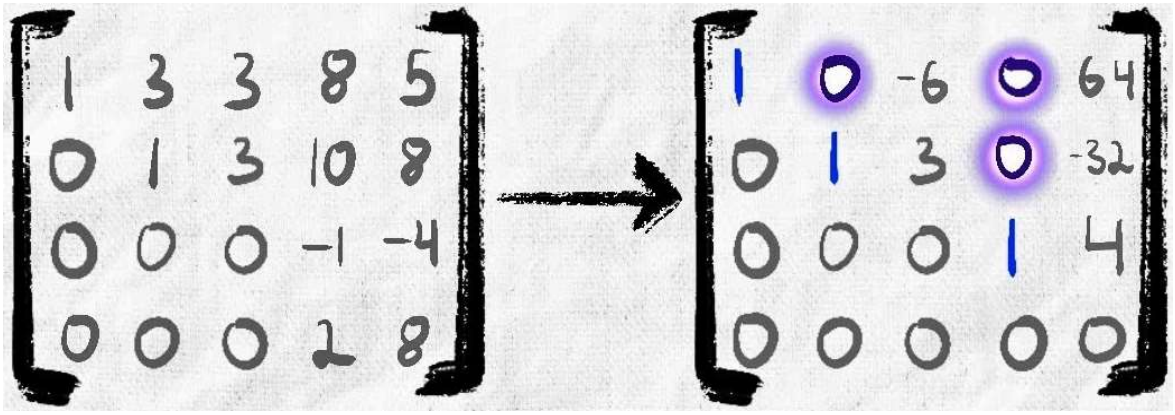

Fig. 1. A matrix turned into its reduced row echelon form. The blue values are pivots. The highlighted values indicate zero values above the pivot. Adapted from [5]

This snippet performs the row reduction of swapping two rows.

```
do jj = j, size(matrix, 2)
  tempMatrix(1, jj) = rref(lead, jj)
end do
```

```
rref(lead, :) = rref(i, :)
do jj = j, size(matrix, 2)
  rref(i, jj) = tempMatrix(1, jj)
end do
```

This snippet performs the row reduction of adding a multiple of a row to another.

```
do ii = 1, size(matrix, 1)
  if ((rref(ii, j) /= 0) .and. (lead /= ii)) then
    div = rref(ii, j) / rref(lead, j)
    rref(ii, :) = rref(ii, :) - (rref(lead, :) * div)
  end if
end do
```

This snippet performs the row reduction of multiplying the row by the inverse of the pivot.

```
div = rref(lead, j)
do jj = j, size(matrix, 2)
  rref(lead, jj) = rref(lead, jj) / div
end do
```

Because a matrix is inherently 2 dimensional (one dimension for rows, one dimension for columns), I used the dimension statement in Fortran to create a multidimensional array that would represent a matrix:

```
real(8), dimension(3,3) :: A
```

In the above snippet, I created an array A with 3 columns and 3 rows (3x3 matrix) composed of double precision values. This allows me to perform any calculations within the multidimensional array as you would with a normal array. Thus, this program illustrates one of the common uses of Fortran as it allows for intuitive programming of mathematical concepts.

The second reason I identified was the longevity for Fortran code and its backwards compatibility. By allowing newer iterations of Fortran to be backwards compatible with older versions, scientists are still able to use legacy code which is largely preferable over having to write new code even if the legacy code is harder to read and not as efficient. Especially when many of our programs for weather prediction, traffic monitoring, and etc originated in the 80s when Fortran was quite popular, Fortran today is still of great value [10].

Fortran Aspects

Language Paradigms

Fortran is a multi-paradigm programming language which can be tailored to the specific problem the program is trying to solve. Fortran as an imperative programming language is capable of both object-oriented and procedural programming [7]. You can create objects and polymorphism through derived types and modules starting from Fortran 2003, allowing for object-oriented programming [11]. For example, inheritance can be achieved by utilizing a submodule within a module. It's also imperative as it runs a sequence of instructions that can be issuing commands, controlling program flow in the form of conditionals and loops, defining

variables, and creating functions. It's procedural as you can use program units like `subroutine` and `function` to perform step by step computation [7].

I wrote a program that calculates the greatest common divisor between two positive integers. This program borrows inspiration from the Euclidean Algorithm which divides the larger number by a smaller number. Recursively perform the same calculations between the smaller number and the remainder of the division until there is a remainder of 0. With a focus on exploring Fortran's procedural programming aspect, this program explores functions especially by utilizing `recursive` in front of the function block. This indicates to Fortran that the function is a recursive function which is able to call itself within the function.

```
recursive function GCD(a, b) result(x)
implicit none
  real, intent(in) :: a, b
  real :: x, div, mul
  integer :: i
  div = a / b
  if (div /= int(div)) then
    i = 0
    mul = 0
    do while (mul < a)
      i = i + 1
      mul = b * i
    end do
    mul = b * (i - 1)
    x = GCD(b, a - mul)
  else
    x = b
  end if
end function GCD
```

Other than the imperative programming paradigms, Fortran is also capable of array-oriented programming, functional programming, and parallel programming [7]. In the previous section on common uses, I heavily demonstrated Fortran's capabilities for array-oriented programming by creating a rref calculator utilizing multidimensional arrays as matrices. This array-oriented programming allows Fortran to be useful for computations of large data sets, making it ideal for scientific computing. Because of Fortran's historical background in mathematics and science, it was previously a functional programming language. And even with Modern Fortran, functional programming is still possible. What makes Fortran capable of functional programming is that you're able to create pure functions in which the parameters within the function can't be changed and while mapping inputs with output. Since Fortran 2008, the language has also been capable of parallel programming in which code could be run on a single CPU or shared-memory by utilizing coarrays, teams, etc [11].

Type Characteristics

Fortran is an ahead-of-time (AOT) compiled programming language which means that compilation is done before it's run. And as the program is being compiled, the type of each variable is fixed and can't be changed [1]. By this definition of static typing, it makes Fortran a statically typed programming language [7]. But this is only possible because of `implicit none`, an additional statement written at the start of a program which signals to the compiler that everything will be explicitly typed, as opposed to implicit typing [11]. Variables are declared by through the following format:

```
<data-type> :: <name>
```

The example below creates a variable `x` with the real data type.

```
real :: x
```

Because implicit typing is considered bad practice, you wouldn't usually find a Fortran program without `implicit none`. But Fortran is still capable of implicit, otherwise known as weak, typing as long as `implicit none` isn't included within the program/function unit. When utilizing implicit typing in Fortran, variable names that start with the characters "i" through "n" in the alphabet will be considered integers unless specified. All other variable names default into real values [2]. Or you can make your own user-defined implicit typing utilizing the `implicit` statement which can set variables that start with a certain character a default type you want to give it. However, implicit typing can lead to fatal program errors such as when a variable name is misspelled. Thus, it has been considered a deprecated feature by many and should not be used [11]. Keeping this feature, nevertheless, does ensure the backwards compatibility of Fortran with legacy code.

Syntax & Semantics

In Fortran, a program unit is a sequence which can define the environment and run any commands before it is terminated by an `end` statement. In Fortran, it can either be a main program, an external subprogram, a module, or a block data program unit. All Fortran programs start with the `program` statement and are terminated with the end `program`. An external subprogram (or procedure) can be a function or subroutine that's outside of another program unit such as the main program, a module, etc. A module contains definitions that can be used in other program units through the `use` statement. As previously mentioned, modules correspond to classes in languages like C++ for object-oriented programming. Finally, a block data program unit specifies initial values for objects in named common blocks. Common blocks provide a way to pass information between subroutines. However, the block data program unit can be replaced with a module in Modern Fortran. I would like to note that common blocks are different from the `Block` statement introduced in Fortran 2008 which is used to contain declarations [11].

In Fortran, global scopes do not exist. Variables have a local scope that are only accessible in the function with which they are defined. Variables can't be passed by reference between program units. Utilizing `intent(in)` or `intent(out)`, variables can be given as an input or output, but they can't be changed [11].

When a program unit wants to use the variables from an external subprogram, you will have to use an `interface` block. The `interface` block tells how an external subprogram is to be used alongside the subprogram's arguments and types [11].

This snippet is an example of an interface used in the rref program I mentioned previously so that I can access the variables from the `rref` external subprogram.

```
interface matrixManipulation
             function rref(matrix)
                real(8), intent(in) :: matrix(:,:)
                real(8) rref(size(matrix,1), size(matrix,2))
             end function
end interface
```

Comments in Fortran can be started with the escalation mark character "!" which ignores any characters after it. The only special case is when it's used in a character string [11].

Commands such as `print*` and `write(*,*)` displays data as output while `read(*,*)` takes in user input. These I/O commands can be formatted using format edit descriptors to explicitly statement how you want the input/output data to be processed or expressed [11].

This snippet of code presents an example of I/O commands in which the `write` statement would first write out a single space (formatted by `1x`) and then the string `"Hello World!"` (formatted by `a`). Since `advance='no'`, when `read` is reading from input, it will be typed inline instead of a newline.

```
write(*, '(1x, a)', advance='no') "Hello World!"
read(*,*) X
Initial output => " Hello World!"
Input => " Goodbye school!"
Displayed output => " Hello World! Goodbye school!"
```

Identifiers are names for variables, programs, subprograms, etc that gives the user the ability to identify specific sequences of code. In Fortran, identifiers can't be longer than 31 characters that are either alphanumeric or underscores. But, the first character of identifiers has to be a letter. Finally, identifier names are case sensitive which makes implicit typing all the more dangerous. Words used as keywords cannot be used as identifiers. These are words inherent to the Fortran language such as `if`, `intent`, `interface`, `module`, `exit`, etc [2].

The operators in Fortran can be categorized into three different types: arithmetic, relational, and logical operators. Arithmetic operators has addition `+`, subtraction `-`, multiplication `*`, division `/`, and exponentiation `**`. Relational operators can be written in two different ways based on preferences. For example, the == operator and `.eq.` both check whether two values are equal. In contrast, `/=` and `.ne.` both check if two values are not equal. Lastly, there are relational operators like `.and.` or `.or.` which provides a boolean value based on the logic provided [2]. There are also logical operators that check the equivalence between two operands through `.eqv.` or `.neqv.` based on whether or not they share the same memory

[11]. Operators also have precedence in Fortran which would determine how an expression is broken down [2].

This snippet below provides a quick example of operator precedence in which it would return 24 instead of 0 because the `*` has a higher precedence than `+`.

```
Answer = 24 + 1 * 0
Output => 24
```

There are also some interesting commands to handle execution flow in Fortran. Two of which are `cycle`, `exit` and `goto`. Fortran's `cycle` and `exit` are equivalent to Python's `continue` and `break`, respectively. In a `do` loop, `cycle` starts the next iteration of the `do` loop while `exit` would terminate the `do` loop entirely [11].

The following snippet is a `do while` loop that would print out all odd numbers from 1-50 because the `do` loop will move on to the next iteration when it is even using `cycle`, and it will `exit` once variable `i` reaches 51.

```
i = 0
do while (i /= 100)
  i = i + 1
  if (mod(i,2) == 0) then
    cycle
  else if (i == 51) then
    exit
  end if
  print*, i
end do
```

The `goto` statement in Fortran is reminiscent of the `goto` statement in C. By creating a label, you can use the goto statement to, in a sense, jump to the line of code that has the label and transfers control of the program to the label [11].

The following snippet of code I adapted from the GCD program takes in two integers into variables a and b from user input. Because of how my GCD program works, those two integer variables have to be positive for it to run properly. By creating the label 100, I was able to use goto as a way to handle errors by jumping to the code with the label 100 and repeating the steps until the user provides two positive integers.

```
100     print*, "Please provide positive integers only"
        write(*, '(1x, a)', advance='no'), "First number: "
        read(*, *) a
        write(*, '(1x, a)', advance='no'), "Second number: "
        read(*, *) b

        <code body here>

        if ((a <= 0) .or. (b <= 0)) then
            print*, "Error: (var <= 0) need (var > 0)"
```

```
        print*
        goto 100
    end if
```

Common Data Types & Structures

Fortran has integer, real, logical, character, and complex intrinsic data types. Integers, like in most programming languages, store whole number values. Reals store floating point values which can be changed based on how precise you want them to be, with `real(8)` providing the highest 64-bit value precision (double precision). Logical data types are the booleans in Fortran that will either be `.true.` or `.false.`. Character data types store strings and have to be given a length using the `len` specifier. Finally, complex data types are made of two floating point values in which one acts as the real part and the other as the imaginary part [2].

To explore the complex data type even further, I wrote a program that essentially asks the user to provide the values for a quadratic equation and then finds the solution for that quadratic equation utilizing the quadratic formula. The complex data type allows calculations of a value with both a real part and an imaginary part (a + bi), a complex number. Because we're able to calculate with complex numbers in Fortran, we're able to do things such as calculate the complex solutions that result from the quadratic formula. These complex solutions stem from square rooting a negative discriminant value which doesn't exist within the set of real numbers.

This snippet computes the discriminant of the quadratic formula in which the complex variable `quadraticFormula` can be square rooted despite the discriminant being a negative number such that it will result in an actual complex value with a real and imaginary part. Both the real and imaginary parts of a complex variable x can be accessed separately as well using `real(x)` and `aimag(x)` [11].

```
quadraticFormula = B**2 - 4*A*C
quadraticFormula = sqrt(quadraticFormula)
```

There are also arrays in which you can create a sequential collection of elements of the same type. Arrays can be both one-dimensional or multidimensional (as in the rref example). With Fortran 2008 and forward, the maximum number of dimensions, otherwise known as the rank, of an array is 15 [11].

Finally, I also briefly mentioned derived types when talking about Fortran as an object-oriented programming language. Derived types are user-defined data types which are made of Fortran's intrinsic data types. A derived type can contain program units that are bound to it using the `contain` statement, thus establishing what is similar to a class in other object-oriented programming languages [7].

An example of a simple derived type that is able to declare a variable as a `person` which is composed of a `name` string and an `age` integer.

```
type :: person
  character(len = 10) :: name
  integer :: age
end type
```

Interesting Features

Some interesting features of Fortran include procedure pointers and coarrays. A feature added to Fortran 2003, procedure pointers which is a derived type that, as its name implies, points to a procedure. The procedure pointer can be associated with any subprogram that isn't a procedure pointer. Procedure pointers can help with increasing the performance of the Fortran program and can represent pointers used in C, allowing Fortran to be interoperable with C [11].

After Fortran 2008, coarrays were added to the Fortran standard. Coarrays is a model which allows parallel programming to be done directly using only a Fortran compiler instead of having to use an external API like Message Passaging Interface (MPI) [12]. Coarrays allow for work distribution and data distribution. Work can be distributed as a single program can be replicated such that each replication, called images, has its own data. Data can be distributed as each image can access objects on any other image. In Fortran 2018, coarrays were given enhancements in the form of teams which were designed to allow for the independent execution of image sets. As one of the many enhancements to coarray programming, Fortran 2018 also introduced image failure handling such that when an image would fail given a large number of images, the team should continue to execute [11].

Coarrays can be created utilizing the `codimension` statement. The following declares a scalar coarray `X`.

```
real, codimension[*] :: X
```

Additional Information

Comparative Analysis

Of the languages we learned in the Principles of Programming Languages course, Fortran shares many similarities and differences with them. One such language is Ruby which is an object-oriented programming language. Ruby is almost entirely object-oriented as nearly everything can be considered an object. This is unlike the multiparadigm Fortran which doesn't necessarily need everything as an object based on the problem to be solved. But Fortran's object-oriented features allows you to create derived types and modules that are of resemblance to the objects in Ruby. Ruby uses just-in-time (JIT) compilation instead of AOT compilation, which means that Ruby is compiled as it is being run. This allows Ruby to be dynamically typed as its interpreter can resolve code as it's being compiled. On the other hand, Fortran's AOT compilation requires it to be statically typed as it's being compiled before the program is run. The methods in Ruby are analogous to subroutines and functions in Fortran in which a section of code can be given an identifier. Ruby also has blocks which, in a sense, can be compared to `do`

loops in Fortran as they both iterate until a given expression is achieved. Although both languages can technically be used for general purpose programming, Ruby is mainly used for web development while Fortran, as mentioned in the common uses section, is used for scientific computing.

Community Support

Fortran has a fairly active Discourse community. It's a forum in which you're able to get help, discuss, and announce things related to Fortran. The community seems to be growing as according to the TIOBE Index for May 2024, Fortran is within the top 10 programming languages [6].

Development environment

In terms of Fortran 2018, when utilizing Intel's Fortran Compiler, the best (and perhaps only) development environment for Windows is Visual Studio (VS). While for linux, there are more options, one of which is Geany. For other, older versions of Fortrans in which compilers are more readily accessible, you can easily implement it into VS Code (different from VS) or your favorite text editor as opposed to forcing you into an IDE [7].

Conclusion

In conclusion, Fortran is a multiparadigm programming language that's mainly useful for scientific computing. Fortran is a statically typed language in which you're able to switch between implicit and explicit typing utilizing the `implicit` statement. It's capable of creating multidimensional arrays and has a syntax in which variables can only be locally defined. It has some unique data types like complex in which it is capable of complex value calculations. It also has some interesting features that allow for interoperability with programming language C such as procedure pointers and features for parallel programming like coarrays and teams. With each new iteration of Fortran (the most recent being Fortran 2023), it has become increasingly popular with a growing community. Fortran ties us to valuable scientific programs created in the past with its backward compatibility and is a vital language to the scientific community.

References

[1] “Ahead of time compilation,” Intel, https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2024-1/ahead-of-time-compilation.html (accessed May 6, 2024).

[2] “Fortran tutorial,” Tutorialspoint, https://www.tutorialspoint.com/fortran/index.htm (accessed May 6, 2024).

[3] “Fortran,” Encyclopædia Britannica, https://www.britannica.com/technology/FORTRAN (accessed May 6, 2024).

[4] “Fortran,” IBM, https://www.ibm.com/history/fortran (accessed May 6, 2024).

[5] “Reduced row echelon form of the Matrix explained: Linear Algebra,” YouTube, https://youtu.be/90YQb3Gajao?si=B6p9r7bzSw-mjrDB (accessed May 6, 2024).

[6] “Tiobe index,” TIOBE, https://www.tiobe.com/tiobe-index/ (accessed May 6, 2024).

[7] “The Fortran programming language,” Fortran Programming Language, https://fortran-lang.org/ (accessed May 6, 2024).

[8] Abby, “Fortran programming language: History, origin, and more,” History, https://history-computer.com/fortran-guide/ (accessed May 6, 2024).

[9] I. D. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*. Cham, Switzerland: Springer Nature Switzerland AG, 2021.

[10] I. Slik, “Fortran is alive!,” VORtech, https://www.vortech.nl/en/fortran-is-alive/ (accessed May 6, 2024).

[11] M. Metcalf, J. Reid, and M. Cohen, *Modern Fortran Explained: Incorporating FORTRAN 2018*. Oxford: Oxford University Press, 2018.

[12] Nersc, “Fortran coarrays,” Coarrays - NERSC Documentation, https://docs.nersc.gov/development/programming-models/coarrays/ (accessed May 6, 2024).