Lecture 1: Nostalgic Programming (**Fortran**)

We start off the semester near the start of PL history, with **Fortran**. Why, you ask, when there are so many nicer languages in the world? Fortran can claim a feat that few languages can: It has survived over 50 years without losing its relevance. While you may have never written Fortran, it's still heavily used in scientific and numeric computing. If you've ever used a major numeric library (e.g. LAPACK or BLAS for linear algebra) it was probably written in Fortran.

It's hard for a language to go 50 years without seeing a lot of change. Many of the language features we take for granted today have made their way into Fortran, but not for a long time. Likewise, old versions of the language have features that have long since fallen out of use. Perhaps more interesting, new versions of Fortran have not just back-ported features from more modern languages, but innovated within Fortran's specialty of high performance code.

Before we get started, a few fun facts:

- Fortran is older than my dad (1957 vs. 1958)
- Intel's Fortran compiler beats *every* other implementation of *every* language on the Programming Language Shootout benchmark.
- Early Fortran compilers fit in 4K words of memory

Timeline of Fortran features:

Year	Feature(s)
1957	I/O, DO loops, GOTO's, IF statements (all fixed-layout)
1958	Functions
1966	Booleans, portability
1977	Block if/else, strings, better I/O
1990	Free-form input, SIMD parallelism, recursion, memory allocation, modules
1995	Better SIMD parallelism
2003	OOP, function pointers
2008	More better parallelism (SIMD and MIMD)

You'll notice there are lots of versions of Fortran. There's no way we're going to cover 7 versions in one lecture, but it IS important to see how the language has evolved over time, so we're going to learn Fortran '57 first, then Fortran '08.

1 Fortran '57

There's nothing like learning by example, so we're going to start out with some example Fortran programs (don't worry, we're going to take our time reading them). Let's start with the very first program given in the original Fortran manual:

(Note to self: Draw this on a grid on the board so people can see the alignment)

```
C      PROGRAM FOR FINDING THE LARGEST VALUE
C    X    ATTAINED BY A SET OF NUMBERS
       BIGA = A(1)
       DO 20 I = 2, N
       IF (BIGA - A(I)) 10, 20, 20
```

```
10  BIGA = A(I)
20  CONTINUE
```

We can already see a lot of weird things happening in even this short code snippet. You'll notice a lot of interestingly-placed whitespace. This is because up until 1990, Fortran was a *fixed-format* language, meaning whitespace matters, a lot. The format is as follows:

- All text is uppercase.
- Column 1 is the comment column. If this position contains a C, the rest of the line is ignored.
- Columns 1-5 (notice column 1 serves double-duty) are the statement number. Statement numbers are needed so you can tell one statement how to jump to another (control flow). More on this later. Note that you don't need a number on every line, only the ones you want to refer to from somewhere else. Also, statement numbers need not be consecutive or even increasing. However, it's often easier to read code where the numbers are increasing, and programmers traditionally space out the sequence numbers enough that they can insert more code without renumbering the rest.
- Column 6 is the continuation column. Remember that Fortran had to be punched onto cards, which have finite length. If you needed to write a really long line, you would spread it out over multiple cards. The way you tell the compiler that two cards are the same line is by putting an X in the 6'th column of the second line. (You can chain multiple cards in this fashion if the line is really long). Note that our comment doesn't actually have to be a continuation line, we could just have two comment lines in a row.
- Columns 7-72 contain the actual program text.
- Columns 73-80 are ignored. Back in the day, these would be used to number the cards in case some stupid intern dropped the deck and you had to put them back in order. These days it's mostly used to write love letters to your girlfriend without the compiler barfing on your program.

Before any of you get too confused, I'd like to point out that the variables A and N aren't defined anywhere. This isn't a magical Fortran feature that invents variables for you. This is just an incomplete code snippet. We're assuming we've already defined these variables somewhere, and that A is an array and N is an integer.

How do we know the types? You'll see how to define an array later, and N is an integer for a funny reason - any variable starting with letters I through N is considered an integer. All other variables are floating point or arrays. This might seem really stupid, but nobody had done types before. Noone had developed the idea of declaring a variable's type along with its name, so Fortran mixed the two together.

The actual code starts on line 3:

```
BIGA = A(1)
```

The first statement defines a new variable BIGA and sets it to the first element of A. We already see two interesting things here: There's no special syntax for declaring a variable vs. assigning it. If the variable exists, it gets modified, otherwise it gets defined. This is common among scripting languages, but somewhat uncommon for a compiled language like Fortran.

Secondly, note that arrays are *1-indexed*. A(1) refers to the first element of the array, not the second like it would in many programming languages. Given the mathematical background to Fortran, and the lack of any established convention, this is not an unreasonable decision. Somewhat less interesting: The syntax $A(I)$ for array access is also pretty uncommon today, in favor of $A[I]$. Again, there was little convention to go on. The only downside of this syntax is that function calls and array accesses look awfully similar (aka the same).

```
      DO 20 I = 2, N
```

The next statement is the start of a DO loop. This is awfully similar to the for loops you know and love, but with a critical difference: Fortran doesn't have any sort of curly-brace-imposed block structure. Every line in some sense stands on its own. Instead of using braces to delimit the end of the loop, we explicitly write out a statement number for the end of the loop. The 20 in `DO 20 I = 2, N` means that after executing statement number 20, we come back to the top of the loop. The `I = 2, N` part is more standard: it means we make a variable `I` whose initial value is 2 and goes up to N(inclusive).

```
      IF (BIGA - A(I)) 10, 20, 20
```

This is called an "arithmetic if". Later versions of Fortran have a more conventional "logical if", but this is the only kind of "if" statement in the original Fortran. To evaluate it, first we compute `BIGA - A(I)`. We then look at its value. We jump to the first statement number if the value is less than 0, the second if exactly 0, the third if greater than 0.

You're probably not used to programming this way, are you? What this line is really trying to compute is "`IF BIGA < A(I) THEN 10 ELSE 20`", but we can't express comparisons directly in fortran, we need to subtract and then compare the value with 0. You're also probably not used to programming with jumps (goto's) either. Early Fortran is a very unstructured language - there are no block statements at all - whenever you want to go somewhere, you have to specify its statement number (except calling a function, which you can thankfully do by name).

```
10    BIGA = A(I)
```

This statement is number 10. If you look at the previous line you'll see this is the `BIGA < A(I)` case, which means we need to update the maximum. That's exactly what we do - this line just changes `BIGA` to be the current element of the array.

```
20    CONTINUE
```

This statement is kind of boring. A `CONTINUE` just returns to the top of a loop. But there is actually something interesting happening here. Not only do we execute this line in the `BIGA >= A(I)` case of the "if" statement, we also execute it in the opposite case, after first executing statement 10. This kind of control flow would be very unnatural in a structured programming language like C - having the else branch start half-way through the then branch, but this style is easy and common in Fortran. Granted, much of the world's worst code has been written by taking this unstructured style to its logical extreme, leading to spaghetti code.

1.1 A more complete example

```
C      PROGRAM FOR FINDING THE LARGEST VALUE
C    X    ATTAINED BY A SET OF NUMBERS
       DIMENSION A(999)
       READ 1 N, (A(I), I = 1,N)
    1  FORMAT (I3/(12F6.2))
       BIGA = A(1)
       DO 20 I = 2, N
       IF (BIGA - A(I)) 10, 20, 20
   10  BIGA = A(I)
   20  CONTINUE
       PRINT 2 N, BIGA
    2  FORMAT (21HTHE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
       STOP 0
```

The `DIMENSION` statement says A is an array with 999 elements. From the name A, we know it's an array of floats.

The next line is a read statement. The first argument `1` is the number of the `FORMAT` statement that specifies the input format. The rest of the line says where to store the inputs. The syntax `N, (A(I), I = 1,N)` means we put the first input in `N`, then put the next `N` inputs in consecutive elements of `A`.

Here are some examples for the FORMAT statement (used both for reading and writing):

I3: A 3-digit integer
H5HELLO: The 5-character string "HELLO"
F3.2: A 3-digit float with two digits after the decimal
I3 / F3.2: Each line alternates between a 3-digit integer and 3-digit float with two digits after the decimal
I3 / I2 / (I1): The first line has a 3-digit integer, the second has 2-digit, and all the rest have 1-digit.
2I3: Each line has 2 3-digit integers

Note the last two formats (and especially the last one) are strange. We can use slashes to separate multiple patterns to say they alternate. If we want to special-case the first few rows and make the rest use the same pattern, we separate with slashes, then put the last pattern in parentheses.

The `STOP` instruction is simple - it just terminates the program and returns the given status code.

1.2 A more complete non-example

At this point you basically know Fortran '57 (it's a pretty simple language), but here are a few more features you may want to know about and one or two you might not.

1.2.1 Standard Library:

The standard library is, umm...small. It contains 5 functions with several variations. The naming conventions are as such:

- All function names end in F.
- Functions that return integers are prefixed with X
- If functions can accept both integers and floating points, the integer version ends in 0F, and the floating point version ends in 1F.

Without further ado, here's the library:

Name	Operation	"Type"
ABSF	Absolute value	`float -> float`
XABSF		`int -> int`
INTF	Round toward 0	`float -> float`
XINTF		`float -> int`
MODF	Compute X mod Y	`float * float -> float`
XMODF		`int * int -> int`
MAX0F	Max of arbitrarily many values	`int * int * ... -> float`
MAX1F		`float * float * ... -> float`
XMAX0F		`int * int * ... -> int`
XMAX1F		`float * float * ... -> int`
MIN0F	Min of arbitrarily many values	`int * int * ... -> float`
MIN1F		`float * float * ... -> float`
XMIN0F		`int * int * ... -> int`
XMIN1F		`float * float * ... -> int`

1.2.2 Arithmetic Operators

You saw the operator `-` in our example above. Fortran supports the four basic arithmetic operators, as well as `**` for exponentiation. Also note that Fortran lacks the logical and bitwise operators that may be familiar to C users.

1.2.3 DO loops

In our earlier example, the DO loop takes three arguments: the number of the loop's last statement, a minimum value and maximum value. DO loops also support a more general case that takes a four argument: an increment. So while the 3-argument do loop

```
      DO 300, I = 1, 100
      ... <STUFF>
  300 CONTINUE
```

is equivalent to the C code:

```
for(int i=1; i <= 100; i++) {
<do stuff>
}
```

you can also a DO loop like

```
      DO 300, I = 1, 5
...      <STUFF>
 300 CONTINUE
```

which is equivalent to the C code:

```
for(int i = 1; i <= 100; i +=5) {
 <do stuff>
}
```

Note also that in Fortran it is illegal to change the loop index or upper bound during the body of a loop, unlike in C.

1.2.4 DIMENSION statements

Earlier we mentioned the `DIMENSION` statement used to define arrays. What we didn't mention is that you can define arrays of up to (but not beyond) 3 dimensions by specifying multiple parameters to the `DIMENSION` clause. So `DIMENSION FOO(2,4,8)` makes `FOO` a $2 \times 4 \times 8$ matrix. Note on memory representation: The first argument varies most rapidly - that is Fortran uses *column-major* order, which is the opposite of C's *row-major* order. I've heard smart math people[1] complain that column-major is faster (better cache performance) for some important matrix algorithms.

1.2.5 GO TO statements

We've somehow managed to skip the most fun statements in Fortran - GO TO! This is possibly the most hated construct in any programming language in the history of mankind, inviting the wrath of such über-nerds as Edsger Dijkstra and Niklaus Wirth. The GO TO statement allows you to write more bad code faster. It appears in three forms:

Unconditional GO TO This literally means "go to this statement, and continue from there". In the absense of block statements, you will use GO TO's a lot. While you don't need GO TO's to implement a loop, you do need them to implement a traditional IF block. Say you want to do something based on a 3-valued compare of X, then do a bunch of the same stuff in every case. You would write that like so:

```
      IF (X) 1, 2, 3
    1 <LESS CASE>
      ...
      GO TO 4
    2 <EQUAL CASE>
      ...
      GO TO 4
    3 <GREATER CASE>
       ...
    4 <REST OF FUNCTION>
```

[1] Earlin Lutz, Bentley Systems, Inc.

Computed GO TO Computed GO TO's have the following syntax

```
GO TO (n_1, ..., n_m) I
```

Here each n_1 is a statement number. I is used as an index into the list of statement numbers, then execution continues at the corresponding statement. This is somewhat analagous to C's switch statement, which each branch of the switch is a different entry in the jump table. This statement can be used to implement efficient multi-way branches, but is hated even more than regular GO TO.

Assigned GO TO This is a variant on computed GO TO. The syntax is almost the same, except the variable comes before the statement numbers:

```
GO TO L (n_1, ..., n_m)
```

Here L is a variable pointing to a statement, and that statement must be one of the entries in the list `(n_1, ..., n_m)`. What does "point to" mean in this case? You can't just set the variable to a statement number with a normal assignment operation, you must use the following:

```
ASSIGN i TO L
```

This instruction means that if later use L in an assigned GOTO, we end up at statement i. Why do we have this separate assign instruction? There's actually a good reason: We want be able to check at compile time that we're jumping to a valid statement (one specified in our list). This is easy to do with the `ASSIGN` statement because i must always be a literal integer. Without `ASSIGN`, it would be undecidable because i could contain any variable depending on runtime behavior.

1.3 Language Evaluation

So now you're all world-class experts in Fortran, so let's take a look at the language design and decide how well the designers did. Let's make a list of the good and bad parts of Fortran:

Good	Bad
Variables, not registers	No functions?
Arithmetic expressions	Microscopic library
Loops	Unstructured
I/O	Fixed-format
	Types?

Looking at the positives, one thing you should take away is that, while quaint compared to today's languages, Fortran '57 represented a massive boost in programmer productivity - early papers estimated a factor of 5 productivity improvement. Having any real programming language, no matter how meager, freed programmers from a lot of really stupid tasks like managing registers or flattening a complicated formula into all its component instructions. Few languages can claim such a great improvement over their predecessors.

Let's tackle the negatives:

No functions So it seems somewhat lacking to create a programming language that doesn't allow you to write functions. It's not like nobody thought of the idea before - Fortran provides a few functions built-in and assembly programmers had been structuring code into functions since time immemorial. The Fortran manual even alludes to simulating functions with GOTO's.

Frankly, not having functions is kind of an inexcusable failure in a language, except maybe for the fact that every other aspect of Fortran was a huge improvement. The real excuse, of course, is that the language added function support within about a year of the original release. One could imagine they were sufficiently excited by the rest of the language that they didn't want to wait to implement functions before the first release.

Standard Library By most standards, 5 functions make for a poor standard library - even SML can do better than that! To anyone who complains, I'd like to point out that early Fortran compilers ran on machines with 4K of memory. Providing users with a large body of code is not very practical when resources are so limited (though many installations did have a few more functions than this library suggests, such as `SQRT` and `SIN`).

Unstructured A major inconvenience in Fortran is the lack of block-structured control flow (this one problem alone was enough to make many people want the language dead), but the fact is this idea didn't really catch on in other languages until the late 60's - early 70's, meaning that Fortran was not too late to the party adding it in '77.

Fixed-format Considering that everybody wrote code on punch cards back in the day, it's totally reasonable that Fortran used a fixed format in the 50's. Of course what's not reasonable is that it maintained this format until 1990. Most programming languages could connect to the Internet before Fortran could compile without spaces at the beginning of each line.

Types Fortran doesn't have many types: int, float and array. Fortran was designed for machines with extremely limited resources, so this is ok. Sure, you can't define a linked list in Fortran '57, but you probably couldn't run any program that needed them in a reasonable length of time either.

What's more sad is the way types are notated. Distinguishing types based on the first character a name is considered, well, unstylish today. This is actually not to far from the idea of Hungarian notation which was popular for a while and still used in some circles today, but taken to an illogical extreme. Fortran already has declarations for arrays - clearly there was nothing stopping it from declaring types at the same time. This is yet another matter of "It was 1957. You can't expect them to think of everything".

2 Fortran '08

So I didn't really have any experience with Fortran '08 or Fortran '57. And I couldn't even find a compiler for '57. Clearly the solution was to use '08 to write a compiler for '57. This is in fact what I did, and you get to use that compiler when writing your homework. What follows is the knowledge I gained from writing that compiler. Modern Fortran has become a fairly big language by comparison, but this should give you a feel for it.

2.1 Implementation Detail

When working with gfortran, it uses file extension to decide what language version you're working with. If you want all the nice features of modern Fortran (like free-form syntax), make your file

names end with `.f08`

2.2 Hello World

We start off again with a simple program:

```
program Hello
print *, "Hello, World!"
end program Hello
```

We get to see our first block statement in Fortran: the `program` declaration. This is in some sense just boilerplate - pick whatever name you want for your program. I have yet to use the name in any meaningful way.

The `print` statement has gotten easier to use. It doesn't require a format string, you just specify an output device (`*` means standard output) and pass it a comma-separated list of arguments (usually but not necessarily strings). If you want to use sophisticated formatting you use the more general `write` statement, which has an embedded format string in a format analogous (not the same) to Fortran '57:

```
program Hello
write (*,"(a)") "Hello, World!"
end program Hello
```

In a format string, "a" refers to a string, and "I" and "F" have the same meanings as they used to. Format specifiers are comma-separated, as the inputs/outputs. This is roughly the same syntax as '57, but now there's no need for a separate format statement and the resulting code is cleaner.

If you are more observant than I am, you may have noticed that there are no functions defined anywhere in this program, yet I said modern Fortran has functions! Essentially the `program` declaration is another function declaration, especially for the `main` function. I can see the appeal of not requiring a `main` function if you're going to write short programs where you don't want extra boilerplate, but in the end having extra syntax seems confusing. I like to follow this template:

```
program Hello
call main
contains
subroutine main()
 ...
end subroutine main
... more functions ....

end program Hello
```

The `contains` keyword is straightforward: it separates the main program from the rest of your functions. The `call` statement is used to call a subroutine, as you might expect. You might not be used to the term subroutine. This is roughly another word for function, and in the context of Fortran it means a function that doesn't return anything (equivalent of a `void`

function in C). Fortran reserves the word `function` to mean specifically things that return stuff (not subroutines). For fun, let's rewrite our '57 example in '08 (though due to the great power of functions, we can leave out the I/O from our example)

```
integer function max(len, a)
integer, intent(in) :: len
integer, dimension(:), intent(in) :: a
integer :: biga = a(1)
do i=1, len
  if (a(i) > biga) then
    biga = a(i)
  end if
end do
end function max
```

The syntax

```
integer function max(len, a)
end function max
```

says `max` takes two arguments `len,a` and returns an integer. Next are the type declarations (both locals and arguments). `intent(in)` marks an input (so it can't be modified), while `intent(out)` marks an output argument and `intent(inout)` means both. Local variables don't have an intent. `dimension(:)` means the variable is an array and the length is unspecified.

Note the "implicit typing" feature still exists and can be used instead of type declarations, though you should disable it, by putting an `implicit none` declaration at the start of your program.

The syntax for `do` and `if` is mostly self-explanatory: We no longer need to specify statement numbers since we have a structured syntax. Instead of '57's arithmetic if, we have logical if, which takes a boolean argument. Speaking of which, here are some new types:

- `logical` is the type more commonly known as `boolean` in most languages. It has two values named `.true.` and `.false.` (Fortran has a trend of putting certain keywords in dots). Logical values are introduced with the comparison operators `==, <, >, <=, >=, .ne.` (the last meaning "not equal"). There are also logical operators `.and., .or., .not.`.
- `string` is not its own type, rather an array of characters. Strings are always fixed-length (the length can be a variable, but once set, the same string is always the same length), and any leftover space is padded with spaces. Use the built-in `trim` function to remove spaces, and `//` as the concat operator. You can easily parse/generate string representations of integers and floats by using the `write` and `read` statements with a string argument instead of an I/O device.
- Pointers are also not quite their own type. If you want a dynamically-sized array, you use the `allocatable` attribute on the variable declaration. The `allocate` statement allocates space for the array and `deallocate` frees memory.

  The following example declares, allocates and frees an array.

  ```
  integer :: length = 10
  ```

```
integer, dimension(:), allocatable :: my_array
allocate my_array(length)
deallocate my_array
```