Brandon Chester

Re-engineering Fortran

Re-engineering process and strategy

The program I chose to re-engineer is a Fortran program that plays the game of dice. This program was quite old and brittle, and like many old programs, it did not follow the kind of structure that is expected of a modern piece of software. Even among Fortran programs it was quite poorly written, and used many features from the Fortran 77 standard that have since been deprecated or removed in the more modern Fortran 95 standard.

The first step of my re-engineering process was to simply understand what the program was doing. This was no small feat due to the usage of arithmetic if statements and many goto statements in the place of proper loops and control flow structures. Fortunately, I did have access to a high level overview of the program, which helped with identifying which areas would correspond to each aspect of the dice game.

As I tried to understand the current structure of the program, I found that the usage of arithmetic if statements was causing me a great deal of confusion. The non-descriptive naming of variables made this an even larger issue than it normally would be, as the statements essentially compared two variables of unknown meaning, and then jumped somewhere else in the code. Without a complete understanding of the program it would be impossible to identify the meaning of the variables, so I had to work backward.

My first step in tackling the problem was to rewrite the program in a pseudocode of sorts. This pseudocode still included GOTO statements, but they were made explicit rather than being embedded in arithmetic if statements. Consider the following code:

Before: `IF(I-L)1,2,2`

After: `if (I >= L) GOTO 2, else GOTO 1.`

By rewriting arithmetic if statements in this form, it is made more clear where the program moves to after the statement is evaluated, and it's also very explicit about what the condition being evaluated actually is. This step didn't eliminate the complexity of GOTO statements which could allow for arbitrary jumps throughout the code, but it did make it possible to track where the jumps were going in order to understand the flow of the program.

The next thing I had to do before I could truly begin re-architecting the program was to identify the behaviour of the label-based loops. These were the loops that had the following form:

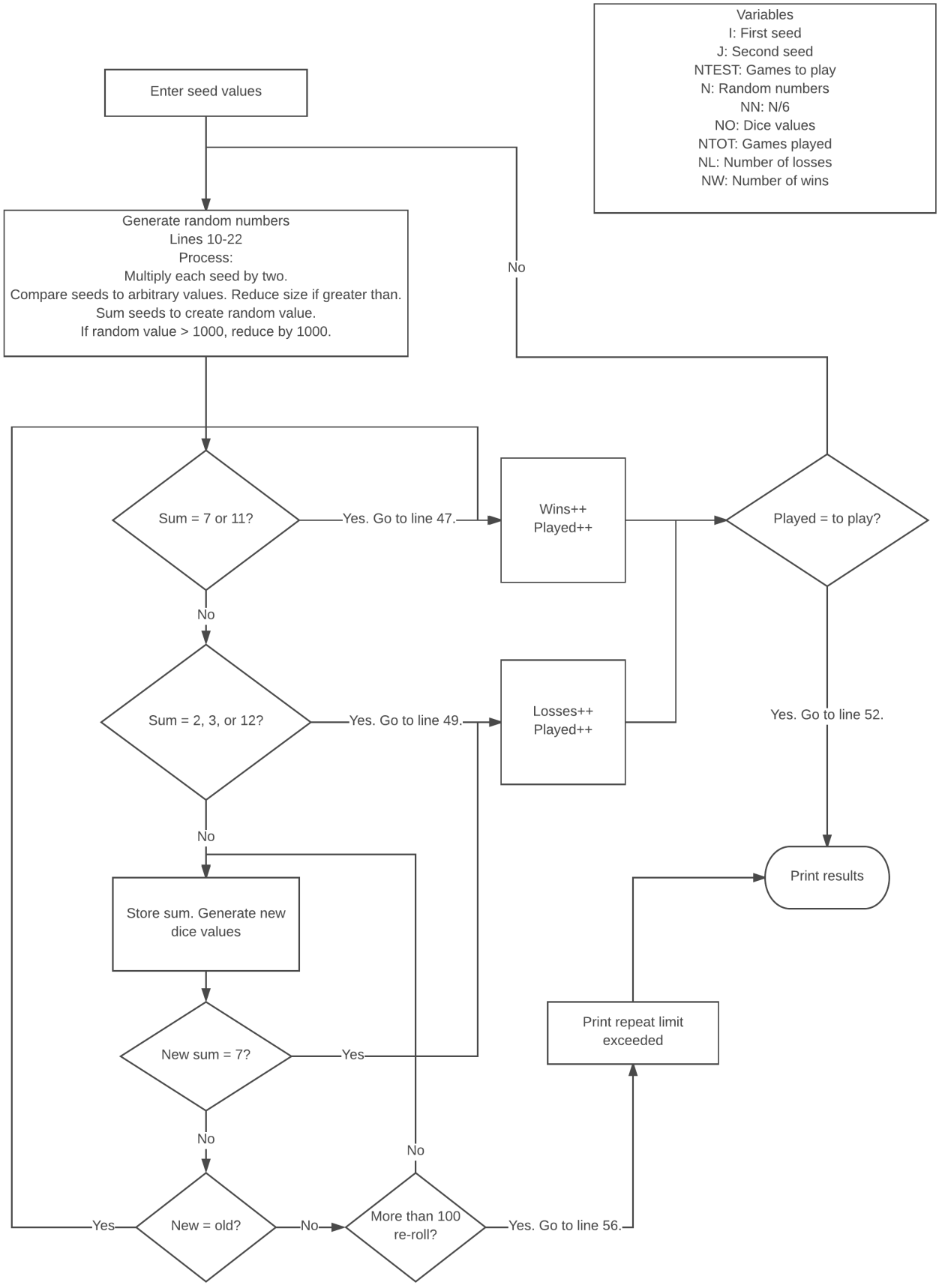
```
DO _LABEL_ K=1,2
```

In modern Fortran, the programmer is expected to begin their loops with a ‘do’ statement and end them with an ‘end do’ statement. The label-based loop in Fortran 77 performs the code specified by a label in the loop header, and doesn’t have an ending statement. This makes it particularly difficult to identify where the loop ends, because body of the loop encompasses every line of code up until a point where a line executes a GOTO statement that brings the execution back to the top of the loop. Fortunately, rewriting the arithmetic if statements before working on the loops made it easy to identify the point where the loop body is closed by a GOTO statement.

Once I had tackled the arithmetic if statements and the label-based loops, I was ready to get to work on re-architecting the program. With those two aspects gone, the task essentially boiled down to refactoring the program’s logic and migrating away from GOTO statements. Identifying the two major loops in the program allowed me to identify two blocks of code that could be moved into a function. The fact that the two blocks were essentially the same code also allowed me to identify the loops in the program as the method through which the program generated the pseudorandom numbers for the dice. Working backward even further allowed me to identify the meaning of most of the variables in the program based on how they were used in the algorithm.

After identifying the variables, the dice function, and the program flow, it was simple to migrate the program to Fortran 95. I created a fully fleshed out diagram which showed the flow of the program to help me with my re-engineering effort. I moved the dice number generation into a subroutine called diceRoll, and I migrated the GOTO statements to proper if statements and do loops. The resulting program looks very similar to something that you’d create in a language like C. It is longer than the original program, but the logic is simple to follow and understand, and only a few in-line comments are needed to explain the more complicated portions.

Brandon Chester



Greatest problems faced

The greatest problem I faced when doing my re-engineering was the use of the arithmetic if statement. I have no idea who thought it would be a good language feature, but it wasn't a good feature in 1957, and it isn't one now. Every time I looked at one I had to go through the thought process to see which variable had to be larger than the other for a certain GOTO statement to be triggered. Writing the if statements explicitly is cleaner and easy to understand.

The other big problem I encountered was the use of implicit variable types. Duck typing is generally not considered a good language feature, and I can't recall any languages that have been created after JavaScript that make use of it. Even in a fifty line Fortran program it makes it significantly more difficult to figure out what a variable is, what scope it can be used in, and what it's meant to do. Even if you give a descriptive name to a variable with an implicit type, it can be difficult to decipher what the type of the variable is unless you look through all the uses to confirm that it only ever holds integer values, or real values, or strings. If the variables in the code had been given explicit types, it would have been easier to figure out what their purpose was even with the cryptic names.

As for GOTO, I actually didn't think it was that difficult to work with one I explicitly wrote what the arithmetic if statements were doing. I've done a bit of work with low level programming, and it's not uncommon to see GOTO statements in code ported from an assembly language so I've gotten comfortable with tracing the flow of a program which uses them. GOTO statements tend to pose a greater issue when they're used to jump out of subprograms, which was thankfully not used in this situation.

What makes Fortran a good language?

Fortran is actually not a bad language at all, especially when you consider the modern incarnations. When looking at the source code for a modern Fortran program, the syntax is not unlike many modern languages, and you can clearly see how some newer languages have been influenced by it. Some aspects like the end statements for if statements and loops may be seen as antiquated due to the C-style brace syntax being used by most modern languages, but even they are not without their merits. Below are some of the aspects of Fortran that stood out to me as being good language features:

Differentiating between functions and subroutines
This is actually a major complaint I have about many programming languages. C and C-style languages treat functions, subroutines, and methods as the same thing, but from a mathematical perspective that is absolutely wrong. A function in a computer program should not have state. That is to say, a function should have a result, and it should be purely a function of its inputs. The same inputs should always produce the

same output, and the function should not affect the state of the program; it should not have side-effects.

Fortran differentiates subroutines and functions. Unfortunately, it doesn't do a perfect job of enforcing the definition of a function because you can still allow parameters that are passed in to be modified, but providing two constructs for encapsulating program logic allows the programmer to enforce this themselves if they'd like to go for a functional programming approach.

Explicit parameter intent in subroutines and functions
This is another feature that C and C-style languages could benefit from. In Fortran, the programmer states the intent for parameters passed to a subroutine or a function. This makes it clear to the caller whether the subroutine can modify the values of the variables that have been passed in. In most other languages, the programmer has to infer this based on whether the method takes reference-type or value-type parameters, which leads to a whole mountain of confusion about reference and value semantics, particularly in the case of object-oriented languages like Java which do not have explicit pointers.

Fortran could benefit from moving these labels into the subroutine definition. The Swift programming language actually has the same concept of an inout parameter as Fortran, and it places them within the method name which makes it clear which parameters can be modified.

End statements for do and if statements
In general, I'm heavily in favour of the Swift programming language's solution to closing loops and control flow statements, which is forcing the use of brace brackets. However, Fortran's solution of end statements for these structures accomplishes the same goal of making it clear where these structures begin and end, removing any ambiguity about scope or program structure.

Ease of implementation compared to C

I actually feel that modern renditions of Fortran resemble modern programming languages very closely. I don't believe it would be easier to implement this program in a language like C, and in fact it was not easier to do so. It may be the case that C has slightly less overhead due to a more compact syntax, as well as not needing to re-declare variables within the scope of a function/subroutine. However, Fortran's inout intent feature for variables passed to a subroutine makes it much easier to modify parameters inside of a subroutine. In C, doing this for value types requires passing the address to the function so you can work with a pointer to that variable, which then requires an explicit dereferencing operation in order to access and modify the values. Fortran allows you to define the behaviour in the variable declaration, and doesn't require the use of explicit pointers.

Difficulty of learning Fortran

As I mentioned earlier, modern Fortran resembles many other modern programming languages. With my experience programming in many languages, learning Fortran was not difficult. The real difficulty lies not with the language, but with poorly written programs that were created using older versions of the language that did not try to enforce good design patterns.

Re-engineering larger Fortran programs

If I were tasked with re-engineering a Fortran program that was 10,000 lines long, I don't believe I would be able to complete the task if the code was as opaque and confusing as the original dice game program was. Tracing the flow of the 50 line program was difficult enough, and that's after considering that it came with a high level flow diagram. I definitely feel comfortable writing some Fortran code from scratch in a modern way, but trying to re-engineer a large amount of old code is not something that I believe I could do successfully.