Tic-Tac-Toe Re-engineered

aka: Pride and Prejudice and Fortran

by Matt Breckon

“Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.” - Michael C. Feathers

At an initial read, this single quote left little to be desired about Fortran. The key distinction to make is Fortran 95 and Fortran IV are completely different beasts. One of which I’ve discovered is actually quite pleasant.

Process

Migration: Fortran IV to Fortran 95

Converting the original Tic-Tac-Toe program from IV to 95 proved to actually be fairly straightforward. I managed the entire process with git to allow myself to make small, isolated changes to the code, and to keep track of what exactly was changed with each commit. The following document will cover overall major changes in reference to related git commits.

At First Glance

Initially, the original code was rather intimidating to look at, and it didn’t make much sense to dive deep into understanding the logic behind all the go to statements until I could sanely read and understand the code as a whole. So, I started simple. My first commit consisted of modernizing the comments from the old format prefixed with C’s to the modern prefix with !’s.

The second step I took was removing implicit variables in the code by introducing “implicit none” at the beginning of the program and at the beginning of each subroutine. This proved to break the program as the index variables I and J used throughout the program and subroutines needed to be declared, and so I did just that.

The next task involved modernizing variable declarations with a simple addition of :: between the datatype and the variable name(s). However, an additional step that took a little more research to understand was modernizing array declarations. Initially, 2D arrays of characters of “kind” 1 were introduced with, for example, “CHARACTER * 1

TICTAC(3,3)". The asterisks appear to represent "kind" so I decided it was best to use the modern representation, along with declaring the array size with the modern dimension attribute.

```
-  character(1) :: tictac(3,3), winner, replay
+  character(1), dimension(3,3) :: tictac
+  character(1) :: winner, replay
```

At this point I realized I could not stand constantly reading code in all-caps. I immediately stopped what I was doing and converted everything to lowercase, modified indentation, and converted the file format to .f95 to support the modified indentation.

The next step I took before moving onto structural re-engineering of the code was replacing ".eq.", ".gt", and ".lt" relational operators with modern substitutes such as "==", ">", and "<".

Afterwards, I jumped to modifying the structure of the program and later returned to a few aspects I missed. So, these are commits later on in the process but apply to the modernization of the look and feel of the program. The first thing I overlooked was adding the parameter attribute to constant variables. The second thing I overlooked was replacing the Hollerith constants used when the board was printed. Only one constant was used as "2x" or "1x" in place of " ". This proved to be a simple modernization of the code. The third and final thing I overlooked until further into the assignment was array initialization. In place of initialization with the statement data, I was able to initialize arrays in one line using reshape.

```
-  integer :: paths(3,8), pathsum(8)
-  data paths/1,2,3,4,5,6,7,8,9,1,4,7,2,5,8,3,6,9,1,5,9,3,5,7/
-  integer :: board(9,2), k, x, y, randpos
-  data board / 1,1,1,2,2,2,3,3,3,1,2,3,1,2,3,1,2,3 /
+  integer :: pathsum(8), k, x, y, randpos
+  integer, dimension(3,8) :: paths = reshape((/1,2,3,4,5,6,7,8,9,1,4,7,2,5,8,3,6,9,1,5,9,3,5,7/), shape(paths))
+  integer, dimension(9,2) :: board = reshape((/ 1,1,1,2,2,2,3,3,3,1,2,3,1,2,3,1,2,3 /), shape(board))
```

Design Structure

Re-engineering the program consisted solely of hunting down go to statements and their corresponding labels. This was definitely the most difficult part of the modernization process due to deciphering the logic behind all of the routing.

Before I tackled replacing use of go to's, I noticed a use of the stop command before the end of the program. The use of stop is obsolete and the program only requires one

entrance and one exit. So I simply removed this line and the program maintained functionality.

From here, I made several commits separately tackling each go to statement block I could find. First, a do loop was used to replace go to's for user move selection. Secondly, a do loop was used to replace the go to for every turn taken by the player and computer. This part took a little more thought as the go to's conditionally looped to different portions of the code to check for winning moves, allow the computer to play, and print both the player and computer's move.

The next step was modernizing the do loops and modifying the go to statements found within the subroutines and functions. Modernizing all of the previously used do loops was trivial as it was a matter of removing the "[label] continue" statement and replacing it with "end do". The rest of the go to statements consisted of replacing the function chkplay's go to branching with a "select case", identical to a switch in C. Subroutine compmove's go to loop for randomly selecting a move was replaced with a do loop. Function same's go to branching for returning true or false was replaced with if-else branching statements.

Going Beyond

Once everything had been clearly understood and modified into working if-else statements and do loops, it was trivial to add additional functionality to the game. A simple prompt post-game if the user wants to replay was added through the use of two do loops. A character variable was added called replay representing "y" or "n" to replay the game. The entire game was wrapped in a loop checking this variable to exit or not, and the post-game prompt was wrapped in a loop in case the user typed anything but "y" or "n".

Discussion

Re-engineering vs. Re-implementing

"Would it have been easier to translate the program into a language such as C?". The answer to this question weighs heavily on what the program is. Most of the time, developers would rather spend the time re-implementing a program in a language they have experience with rather than deal with tackling some legacy code.

In the case of the Tic-Tac-Toe program, the transition from Fortran IV to Fortran 95 provided to be a small enough task that I could tackle in small bursts of free time over the past few days. This small program doesn't depend on external libraries or other dependencies with development overhead, proving re-engineering to be much less work than starting from scratch in a different language such as C.

Modernization Alternatives

As far as the process went, I don't think there is a better way of modernizing Fortran IV to Fortran 95. However, if your goal is generally more modern code, then I would have opted for a more accessible language such as Python or JavaScript. These choices, however, really come down to preference and are not necessarily "better" for this particular Tic-Tac-Toe implementation as Fortran 95 contained everything needed for such a simple application.

Learning Curve

Working in both Fortran IV and Fortran 95 reminded me heavily of working in, respectively, an x86 assembly language and C. Having not touched any assembly in over a year made me initially intimated when I saw the go to statements and labels. But as far as Fortran being easy or hard to learn, it was definitely easy based on my past knowledge. A few things weren't overly clear such as "write(*,*)" and "read(*,*)" dealing with stdin and stdout, reshaping arrays on initialization, and array "kinds". However, these small issues were easily overcome with a bit of research leaving my impression of Fortran as a language easy to learn with a background in x86 assembly and C.

Structures

Working with this rather simple application of Tic-Tac-Toe in Fortran there was not really any distinguishable structures found in Fortran used for this program that were not readily available within C. However, one notable exception I noticed was use of rand() to generate a value without the need for an initialized RNG like you would in C. Also not immediately obvious, but I like how Fortran contains reshape() to allow dynamic arrangement of arrays.