Evan Hedges

CIS 3190: Software for Legacy Systems

Professor Michael Wirth

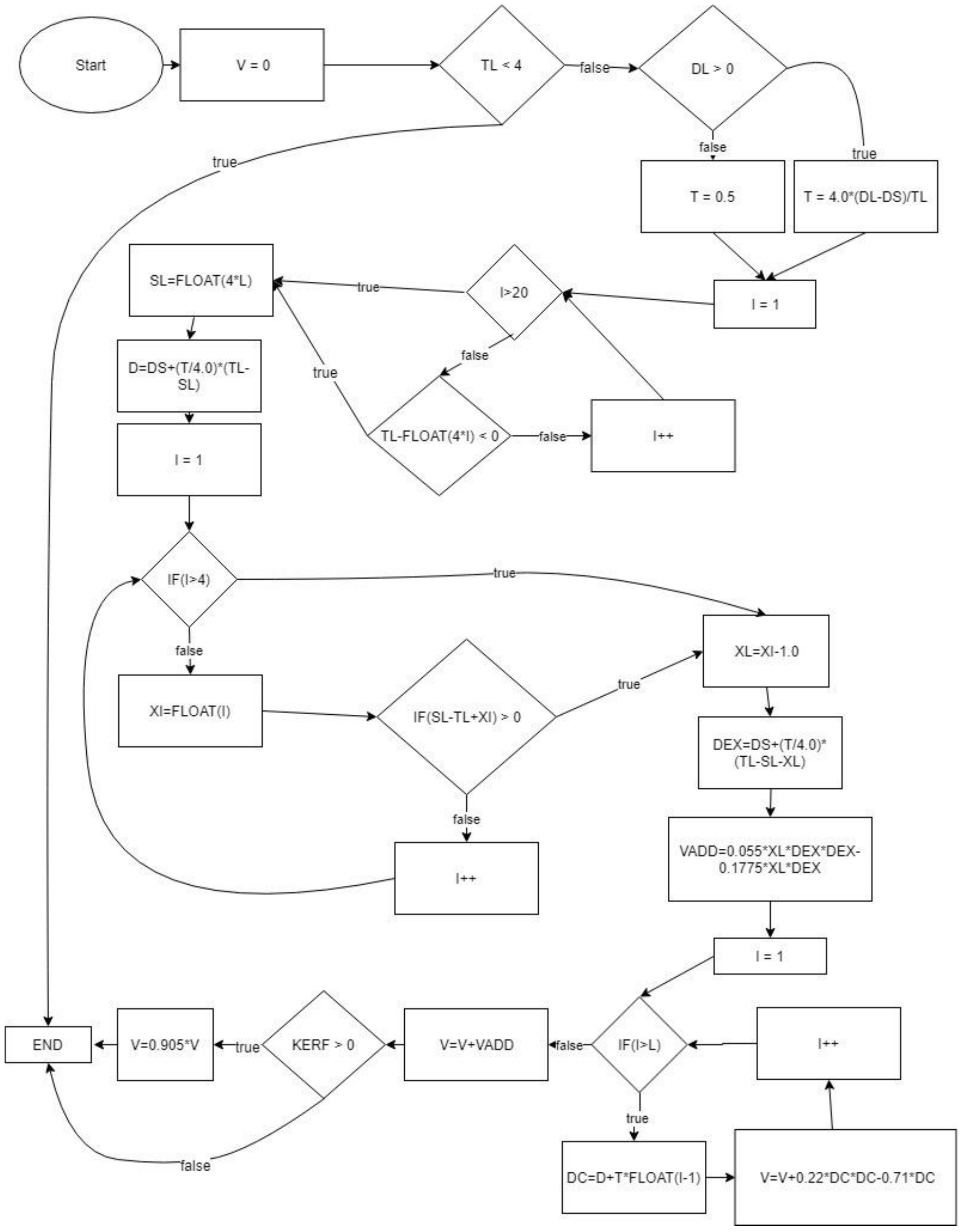
The ability to update outdated code is an important skill. This assignment allowed me to learn about how difficult it can be to understand outdated code. Despite the fact that there were a multitude of comments and descriptions of how the program ran, it was difficult to understand what certain variables were for and how parts of the program worked. It made me understand why proper documentation and variable names are important for allowing code to be updated. Certain variables such DEX did not readily reveal what their purpose was. By changing these variable names to something more descriptive it allows new programmers to understand the code, which results in an easier to maintain product. Another difficulty when updating the code was trying to understand how some of the loops ended. For example, the loop around line nine, I understood that the loop executed for L amount of times, however I where the loop ended. It lacked an END DO or an obvious end point to demonstrate what existed inside the block. Overall, understanding the overflow of the program was the most difficult part. As stated before, the variables name did not allow immediate understanding of their purpose, even with the comments. Furthermore, the certain parts of the control flow, such as some of the loops, were difficult to flow due to the lack of an obvious terminating statement. The end result was a very difficult piece of code to follow and understand.

There are a variety of features that makes Fortran a good language. The ability to use a subroutine and edit the variables is particularly useful. In C, for example, you may need to edit multiple values in a function. The result is either having to create multiple functions to change the values or use memory management to return the values. The best example of this would be user input. In this assignment, it requested that the user input be in a separate function from the min. In C, this would require that the variables either have memory, or create multiple functions to read in the values. The first solution leads could result in leaky memory, meanwhile the second solution is tedious. In Fortran, however, it requires only one subroutine. Another benefit of Fortran is that it does not require me to manage memory. Allocating memory on the heap is useful, however, it can be difficult to catch all the leaks. Therefore, it will require additional tools to catch any mistakes. Although not important to this assignment, Fortran has scientific notation built into the system. This means that there is no external library required to accurately represent large numbers.

I think that C may have been the better language for this program. The main reasoning behind this is that the compiler does not provide the most accurate calculations. There was a slight variation between my values and the test values provided. Although, this does not matter for this project, it may matter on production code that require precision. For example, if someone did not know that the numbers could be imprecise, they may miss that the calculations could be off. If the program sent data to a device, such as a satellite, it may be an expensive mistake. Another benefit would have been the math library. Near the beginning, the program required calculating four foots sections. The original creator used multiple loops to calculate the correct answer In C, I could have used the mod and modulus to find the answer. In Fortran, both the mod and modulo functions require complex numbers. Using the complex numbers did not seem intuitive when reengineering the problem and seemed to add addition levels of unneeded complexity. I could have switched the variables to be a complex number, but this requires a real integer and imaginary pair. As such, I left the way the original formulas because it seemed to be the most intuitive way to understand the program. However, I must recognize that I am more familiar with C, and as such, would have a bias towards that language being easier.

When I began the re-engineering process, I started by doing the basic changes. These included updating comments, adding white space, and changing variable names. The goals behind there changes were to increase the readability and understand the purpose behind each section of code. The next major step was to update the loops in the format. The reason behind this was twofold, it would separate the program in to blocks and it did not affect the logic of the code. If I engineered something incorrectly, the result would have been an obvious error. The output would have been either an infinite loop, very large or very small answer. Next I updated the arithmetic if statements. The if statements were hard to follow, due to some of them being in loops, other standalone statements. These changes were left so long in order to familiarize myself with how the program flowed. Some of the ifs were difficult to understand how they worked and how they affected the calculations. However, by leaving the ifs as the last step, it ensured that the proper flow would happen. Once I started changing the Ifs, it was a tedious testing to ensure that each of the conditions proved to be proper result. Overall, the most important step of reengineering did not lay on how I changed anything, but on using the old code to validate that I was correct. By getting user input, it allowed me to take the old, unedited code, and feed test cases into the program. Therefore, I knew that the output from the original code would always yield the correct values. This allowed me to constantly change the code and ensured that at all times, the reengineered code always output the correct answer.

I did add some embellishments to the program. I realized that if someone is running this program, they may need to calculate multiple logs. Therefore, there is a basic menu that allows the user to enter the values of multiple logs or leave the program. Furthermore, I added some basic error checking, such as restricting the user from input 0 as the small diameter, and preventing negative values from being entered. The reasoning behind this change was to limit the invalid data the user could enter. The most important improvement was the addition of whitespace to increase readability. The original program lacked whitespace, which made reading the program quite difficult. With the addition of white space, it’s easier to understand how the program works.


Start
V = 0
TL < 4
false
DL > 0
false
true
T = 0.5
T = 4.0*(DL-DS)/TL
true
I = 1
I>20
true
false
SL=FLOAT(4*L)
true
TL-FLOAT(4*I) < 0
false
I++
D=DS+(T/4.0)*(TL-SL)
I = 1
IF(I>4)
true
false
XI=FLOAT(I)
IF(SL-TL+XI) > 0
true
false
I++
XL=XI-1.0
DEX=DS+(T/4.0)*(TL-SL-XL)
VADD=0.055*XL*DEX*DEX-0.1775*XL*DEX
I = 1
IF(I>L)
false
true
I++
DC=D+T*FLOAT(I-1)
V=V+0.22*DC*DC-0.71*DC
V=V+VADD
KERF > 0
true
false
V=0.905*V
END