CIS*3190 A1 Design Report

Re-Engineering a Reverse Polish Translator from Fortran IV to Fortran 95

Michael Tran

0524704

Synopsis

For this assignment we were tasked with re-engineering a reverse polish translator from Fortran IV to Fortran 90/95. The reverse polish translator's job is to convert an algebraic equation/notation from infix notation to postfix notation. Infix notation is when the arithmetic operator appears between the two operands to which it is being applied. This may require parentheses to specify the order of operations. Whereas, postfix notation, eliminates the need for parentheses because the operator is placed directly after the two operands to which it applies. This form of notation is sometimes called Reverse Polish Notation.

Before attempting to re-engineer the program, I first had to learn the basics of Fortran which would allow me to understand the program and to change the legacy Fortran program into modern Fortran. After learning much of the basics of modern Fortran, I spent time looking at the flow chart that was given to gain a better understanding of what is happening in the program. Afterwards, I examined the legacy code line by line to locate where I would have to modify legacy features in the program to modern Fortran. While locating all the legacy features in the program I grouped parts of the code where it coordinated with the flow chart to make it easier to understand and convert to modern Fortran. After grouping the code together that's when I began to convert the legacy code to modern Fortran.

The first group included the variable declarations and initializing arrays. I started with the constants that were being used for this program. I decided that it would be better to use a module to define the constants that were used because it would keep all the constants in one place as well as future constants can be added to this module. Another reason for this was to make the code look much more structured. Next, I decided to make a subprogram for initializing the arrays. This would add more structure to the program as well as greater readability. In the legacy code it contained a label for the do loop which had to be modified. As for the variable declarations, I changed the constants to character types and used the parameter attribute. As for the other variables, I changed the source, operatorStack and polishString to character type variables while keeping the other variable as integer type and used the dimension attribute to define them as arrays. After doing all of this, I made sure that the program still worked before moving on to the next group of code.

The second group included reading the user input and deciding if it was valid or not. If the user input was good, the program would go through the string and assign hierarchy values depending if it was an operand or operator. I decided to keep this portion of the code part of the main program because I found it easier to check whether the user wanted to exit the program or inputted to many characters which would force the program to start back at the top. For this portion of the code, this is where I first encountered a go to statement I had to remove from the legacy code. Along with the go to statements, they included the use of legacy relational operator's .EQ. and .GE. as well as a stop statement that had to be modified. This group was occurring all in a do loop that had to be modified to modern Fortran due to a label. After implementing all of this, I once again checked if it still worked before moving on to the last group of code.

The final group – most important group – contained the main part of the program, converting the algebraic notation to reverse polish notation. For this I decided to break it into two subprograms. The first subprogram dealt with translating the user input into the polish translation; however, it needed a second smaller subprogram to reduce redundant code. This portion of the program had the majority of the go to statements where I had to modify. I made use of if statements to remove all of the go to statements. I also had to make use of a do loop to convert the entire algebraic string to RPN. The second subprogram was to move the operator to the polish string. This subprogram would be called after removing parentheses and moving the operand to the polish string.

Finally, for the main program, I decided to use a do while loop. The reason I used a do while loop was because I believe it was easier to control when the program would exit. Inside the do while loop is where I implemented all the groups to get the finished product. After testing the program to make sure that the legacy code was successfully re-engineered to modern Fortran, I decided to implement the exponential operator. The only things I had to do were add the exponential operator to the constants module and add an if statement in the second group of code to give the operator the highest hierarchy value.

<u>Identify Legacy Features</u>

Legacy Code	Re-Engineered Code	Comments
INTEGER*1 SOURCE(4), etc	integer(1), dimension(40) :: source, etc	• Occurs at line 20 • Legacy defined length of array • Modernized it by using dimension attribute to define length of array and used the F90+ standard of defining variables
INTEGER*1 BLANK, etc	character(1) :: BLANK	• Occurs at line 21 • Legacy declaration of variables • Declaration of variables from legacy to F90+ standard

DATA BLANK/1H /, LPAREN/1H(/, etc	Character(1), parameter :: BLANK = ‘ ‘, LPAREN = ‘(‘, etc	• Occurs at line 22-23 • Legacy Hollerith constants used • Used the parameter attribute to define the constants and used character type variable

DO 20 L = 1, 40
POLISH(L) = BLANK
20 CONTINUE
	do i = 1, 40
	polish(i)
	end do
		• Occurs at line 26-31 and 41-52
		• Legacy do loop using a label
		• Replaced it using a modern do loop featuring “end do” statement

IF (SOURCE(M) .EQ. LPAREN) SHIER(M) = 1	if(source(m) == LPAREN) SHIER(m) = 1	• Occurs at lines 42, 46-50, 60,73,76 and 101 • Legacy code using .EQ. relational operator • Replaced with modernized “==” operator
IF (OHIER(J-1) .GE. SHIER(I))	if(operatorHierarchy(j-1) >= inputHierarchy(i))	• Occurs at line 98 • Legacy code using .GE. relational operator • Replaced with modernized “>=” operator
IF (SOURCE(M) .EQ. BLANK) GO TO 60 60 IF (M .EQ. 1) STOP	if(source(m) == BLANK) then exit	• Occurs at line 42, 57, 73, 76, 85, 90, 98, 101, 102, 108 and 113 • Legacy code using go to statements

		• Used modern if statements blocks to deal with the go to statements
IF (M .EQ. 1) STOP	if(source(m) == BLANK) then exit	• Occurs at line 60 • Legacy code used to exit the program • Replace the stop with exit to stop the do loop and check if flag was set to exit program

I = 1
J = 2
K = 1
	Implicit none
	Integer :: i, j ,k
		• Occurs at line 68-70
		• Legacy code assumed that these variables are to be integer type
		• Use of the implicit none ensures correct variable declaration is used

<u>Questions and Answers</u>

1. For a program of this size, I believe you could have easily re-written this program from scratch by using the flow chart with any language of your choice. By inspecting the flow chart, you can easily start grouping things together and start coding from there. However, if you were tasked to re-write a program that has more than 10,000+ lines of code, I don't think it would be efficient. It would be plausible to do but with no documentation you would not know how the program should execute. Having the legacy code allows you to compile the program and run it for yourself to see how it works. From this you can start planning how to code the program.

2. The toughest challenge during the re-engineering process was dealing with the go to statements and the spaghetti mess of unstructured code. It made it difficult for me to follow what was happening in the code. With all the go to statements had me jumping from one page of the code to another, I found it irritating. In addition, the unstructured nature of the code was hard to look at because I like to have clean structured code that is easy to read. This allows me to gain a better understanding of what is happening in the

program rather than being frustrated at reading unreadable code. Overall, I had no problems adjusting to the language.

3. My program is a bit longer than the original legacy code. Compared to the legacy code, my program is much more structured. While the legacy code relies on the go to statements, this allows the code to be shorter than mine. I would have to structure the code using loops to get rid of all the go to statements which adds to the length of the program. Also with my program I used subprograms and modules to make the program more readable.

4. In my personal opinion, I believe this was a good approach of re-engineering the legacy code. When you have code to look at you can gain a better knowledge of what the program might be trying to accomplish, rather than starting from scratch. However, it depends on the size of the program. For this assignment, you could have easily started from scratch and just used the flow chart to code this program. The flow chart was very well done and easy to follow which is always important. On the other hand, trying to re-engineer legacy code that is 10,000+ lines would be a difficult task to accomplish. Having the legacy code and a flow chart allows for better understanding and makes it simpler to tackle the problem of re-engineering the legacy code.

Being faced with learning a new language can sometimes be difficult. Having to learn the syntax of the language, how to declare variables, how to create functions and I/O can be difficult to grasp. However, at no point while learning Fortran did I find it difficult to understand. Having a solid foundation in the C language allowed me to grasp Fortran very quickly. It also helps that C was influenced by Fortran and have many similarities to each other. Overall the experience was fun and I enjoyed learning/coding in Fortran.