Modernizing Legacy Fortran

William (Aidan) Maher

University of Guelph

CIS*3190 Legacy Software

M. Wirth

February 5, 2015

1 The Plan/Re-engineering overview

Analyze the problem the code was intended to solve: Turn an input string from infix mathematical notation to Reverse Polish Notation. Analyze the problem from a computer science perspective: What kind of data types, libraries and functionalities from a computer am I going to use to solve this problem? Can I port over the legacy code directly to newer version of the language?

To be honest where I started was to attempt to copy/paste over Legacy code and attempt to compile with the Fortran 95 compiler and see what I would have to change just to get it running and I thought I would go from there. This was not possible, too many GOTO statements and loops with labels. Again I attempted to re-engineer the code with the given flowchart, which was next to impossible since the Legacy code was using integers to compare characters and it was awful to try to think like that when in Fortran 95 I knew we had nice character types and string types. The final time I re-built it, I did it from scratch. I watched videos on how to turn an equation into Reverse Polish notation. I was able to break down the problem very simply in my head (this is pseudocode).

```
while(currChar->next!=null):
    if(currChar==operand):
        //Put it in polish string
    elif(currChar!=rightBracket)
        //check it's precedence and put it in polish or operand
```

There would be a few other conditions and one other operation, specifically unloading the stack of operands, but that's all easy and just details.

So first the basics of the overall task: input a string, manipulate it, output the string. Inputting a string in Fortran 95 was fairly simple, since I got a hint that perhaps the program would be graded by redirecting stdin doing this in Fortran was fairly simple. You simply open a file stream to the number 5 for stdin. And if you wanted to output to a text file the print statement already goes to stdout so you can easily redirect the stream if necessary.

In the Legacy code, it was coded to read 30 characters from the keyboard. It then stored it in an integer array. This makes no sense to me, why you would want to think or code this way is beyond me. Obviously they had to back then, but I didn't need to now with F95. So I used F95 strings and characters to do the string manipulation and comparisons. This was easier for me to think and code this way since I'm use to strings being dealt with like characters. Also in F95 you can use len(string) which returns the length of the string which is a handy feature.

When I attempted to re-engineer from the Legacy code, I noticed how all the variables were in all-caps with a few weird names I did not understand. Therefore you can see in my code the names are properly camelCased and with not too many vague names. OHIER and SHIER were the hierarchy tables for comparing operator precedence, but I didn't really care about how they worked and was too confusing in F77.

To rectify the hierarchy problem I made a function called precedence which returned an integer value for the precedence of the operator. I think this is a lot easier to understand especially when you're reading the code. The legacy code was broken up in archaic GOTO statements to other IF statements to other GOTO statements... what a headache. My code is very procedural with a couple helper functions to make it easier to read.

Back to solving the problem. After the string was read in, it was simply comparing and manipulating the string from there. If the character was an operand simply put it in the polish string.

More Simply put:

1. Read source string character by character

2a. If character is an operand, put it in Polish stack, iterate polish index

2b. If it's an operator AND it's precedence is higher, put in operator stack

2c. If it's an operator AND it's precedence is lower, empty operator stack into Polish of higher OR equal precedence, until a lower precedence is reached. Then put the character into the operator stack

2d. If it's a RIGHT BRACKET, empty the operator stack

3. End of string?

4. Write Polish string

I call it an operator stack but it's really just a string that I keep track of like a stack.

2 F77 VS F95

Comparing the two sets of code, the first thing I notice is the variable declarations. IN F77 they use a DATA statement to input the values into their variables. I do not need to store the characters in any variable since I use a function to determine if it's an operand or not. Also in F95 I can simply compare a character, i.e

```
          if(isOperand(currChar)) then          // code
```

instead of

```
C       CHECK FOR OPERAND
70      IF ( SHIER(I) .EQ. 0 ) GO TO 90
           //code
```

Hurts the eyes doesn't it? And mine is self documenting, if you can read English and are an inexperienced programmer, you can read that if statement and understand to a degree what it is doing. I will give credit that the F77 code is well documented enough to understand, but it's an eyesore.

Back to Fortran feature differences I had to deal with: Hollerith constants. In F77 Hollerith constants are used to represent ASCII characters as integers, they're almost like strings but not. In the legacy code they used the data statement with Hollerith characters to assign ASCII codes to the corresponding variable names. In F95 character types are allowed, so this solved a lot of my problems and was much easier to code.

Another little note on strings is that in F95 you can concatenate strings with the double slash (//) operator. This was useful because when I declare re-initialize stack pointers and strings, I pad the string with spaces to clear it. Therefore when I used len(polish) it was always full, so I had to use len(trim(polish)) which would trim the polish string without padded spaces and then I could (//) the rest of a the operator stack onto the string when necessary.

Labels are not necessary in loops. The F77 loops and labels and goto statement remind me of assembly. I don't want to code in assembly, my code you can read top to bottom and understand what's going on by reading it once or twice. You would have to trace your finger all over the screen to understand what's going on in the F77 control flow, i.e spaghetti code. And for FORMAT statements for output, I did not use format statements because I used the print statement and would send the format as the first argument to print as a string. This type of output is similar to C which I am used to. I.e specifying a format, then giving the list of variables.

3 Questions from Assignment

Would it have been easier to re-write the program from scratch in a language such as C?

There are probably a couple better functions for tokenizing strings, applying a regex etc. But from another point of view, I used the most basic principles of the programming language (since I had to learn it in a couple weeks). In my opinion it would have been easier to write it in C simply using libraries that I am aware of like regex.h, although it is a third party library not standard with GCC. So if I couldn't use any libraries it might be about the same length

I'm sure there's huge efficiency differences since C is well known to have very optimized code. I learned that Fortran and C do not allow aliasing i.e 2 pointers pointing to the same location means they are aliased. This helps prevents overlapping memory for pointers and pointer allocations.

What were the greatest problems faced during the re-engineering process?

Attempting to port over the code directly and try trial and error copy pasting, moving around statements, changing a few things here or there just to make the compiler happy which ended up breaking the program did not work at all. Attempting to code the flowchart was a bit difficult too, it only made sense for the logic of the old code, and you can place arrows going back in many directions jumping around all over the place which was a nightmare to attempt to code.

Was your program shorter or longer? Why?

My program was (50 lines longer without/without comments) longer because I did not use GOTO statements. GOTO statements are useful for deeply nested logic in some cases which makes the legacy code shorter. They are also useful for breaking out of deeply nested loops. The logic is a tad complicated for this problem with cases and subcases, so you have to code it in such a way that the code will eventually exit out the loop and IF block. But at some points while I was coding, I did see the uses of GOTO statements, but also understood that using them hampers the debugging process (for yourself or others).

Is there a better way of writing the program?

In Fortran? Maybe if you had custom defined ADT's (some kind of Priority Queue) with operations that you could use. Thus the main logic of all the code would be shorter. Which you could do with custom Types in F95, and functions that deal with those types, you could hide the code in other files and work with that. You can do this with gfortran, it will link the files similiar to C:

gfortran -o executable source1.f90

source2.f90 - https://gcc.gnu.org/wiki/GFortranUsage

Same goes for if I was attempting to do it the best way in C, with ADT's and libraries.

4 Going Beyond

To implement the exponential operator, I simply added it to my precedence hierarchy as the highest precedence, it still follows the same rules for RPN from there. I ran a couple test cases on my own from converters or examples online, and it worked.

5 Program Usage

To run the program type:

gfortran rpn.f95

./a.out < polish.text

You may redirect the input stream from a text file, it will open the file and read to the end, spitting out the output. **You may also run the program simply by typing ./a.out**

At this point the stdin will be open and you may input a regular mathematical equation you want into reverse polish. Type it in and press enter. ***To quit, press q (LOWER CASE) and press enter. DO NOT put q as the first character in any string or the program will terminate***

There is no prompt because I assumed we would just be running these through with text files for quick marking.