Reverse Polish Notation Translator

or: Adventures in porting legacy Fortran
by Ryan Wilson-Perkin

I must preface this report with an apology.

Forgive me Michael Feathers for I have sinned. For years I have looked down on legacy software with disdain. I have cracked jokes from my position of assumed programming superiority, as I wrote yet-another-python-script.

Only now do I realize the folly of my ways: Fortran 95 is actually quite nice!
Fortran IV however...

Migration Process

Migrating the existing Reverse Polish Notation (RPN) program to a modern dialect of Fortran proved fairly simple. The process was undertaken using git for version control, which allowed for discrete units of change in each commit. I'll document the migration process in terms of the major changes that took place, and reference commits as appropriate.

Look and Feel

During this stage, the actual structure of the code was left largely untouched. Instead of trying to jump right into untangling the logic of the program, I focussed on introducing some sanity into the use of variables, constant, and comments. In the very first commit, comments were changed from the old format (prefixed with C's) to the modern standard (prefixed with !'s).

Next, in accordance with best practices, I introduced the "implicit none" line at the beginning of the program. This line instructs the compiler to disallow implicit variables, allowing the compiler to inform you of potential errors before runtime. This promptly broke everything, since the program relied on implicit variables for index variables I, J, K, L, and M. That's fine, the variables should have been declared anyways. Adding declarations for them informs the user of their exact type.

The array variables were using the old notation of "INTEGER*1 id(length)". It took some research to determine that the number following the asterisk indicated the "kind" of the variable. Once I knew the term for it, I was able to find that the current way of representing "kind" can be done in several ways. I opted to enclose the value in parentheses next to the type specifier (ie. "INTEGER(1)"). Furthermore, the new way of declaring arrays is to specify a "dimension" attribute to the variables, listing how long the array should be.

```
integer(1), dimension(40) :: SOURCE, SHIER, OPSTCK, OHIER, POLISH
integer(1) :: BLANK, LPAREN, RPAREN, PLUS, MINUS, ASTRSK, SLASH
```

Since early versions of Fortran had no character type, programmers had to rely on Hollerith constants to specify character data. Now we can do away with the Hollerith constants. I replaced references to Hollerith constants with the ICHAR subroutine, that returns the integer value of a given character.

Finally, certainly my favourite commit, “Replace uppercase to lowercase”. It no longer feels like Fortran is yelling at me.

Structure

Structural re-engineering of the program consisted mainly of replacing “GO TO”s with loops and conditional statements. This was by far the most difficult part of the process because it involved tracing the various logical routes that the program could take. To determine the logic, I relied on the flow diagrams provided with the assignment to trace the conditions that are evaluated before each jump.

Once I understood the logic of the program, it was trivial to map the label numbers and the GO TO statements to their respective structures. This resulted in one large DO loop that wrapped the entirety of the reading and writing section. Within the large DO loop, there were several iterative loops and IF statements that performed specific checks to either continue or exit the inner and outer loops.

Improvement

The final steps were mostly housekeeping:

- The comments were re-written to reflect the new structure of the program.
- Array instantiation was updated to the modern form to avoid an unnecessary loop.
- If statements were changed to select-case statements where appropriate.
- Labelled format lines were replaced with inline format specifiers.
- Indentation was changed from “fixed form” to “free form”
- Long lines were broken up wherever possible.
- The “program” keyword was introduced to encapsulate the code.
- Prompts were improved.

Many of these steps were recommended in the unofficial [“Fortran Best Practices” guide](#).

Going Beyond (adding exponentiation)

Once the code had been understood and re-engineered, the process of adding the exponentiation operator was as simple as two lines of code.

```
            case ('*', '/')
                shier(m) = 4
+           case ('^')
+               shier(m) = 5
            case default
                shier(m) = 0
```

Additional Discussion

Re-engineering vs. Re-implementing

A question to be addressed is, “Would it have been easier to re-write the program from scratch in a language such as C?”. This question is common when dealing with legacy code, since most programmers would rather spend the effort to re-implement a program in a language they are more familiar with to gain more control over it. I have done this in the past when re-implementing Matlab code in C++ to avoid the overhead of maintaining a Matlab library.

In this case, the program is small enough, and simple enough, that it does not bear repeating in another language. The program works well, does not have external dependencies, and was small enough to be re-written in an afternoon. Were I to re-implement this, I would likely turn to a scripting language such as python which can take advantage of some functional programming paradigms to reduce the code size significantly (at the cost of additional complexity).

Problems Encountered

As previously mentioned in the migration discussion, the biggest problem was deciphering the paths that the “GO TO”s took throughout the code, and determining the equivalent looping structure. This reduces to a compilation process: translating from one language (of “GO TO”s and labels) to an analogous language (of DO loops and IF statements). The problem was mainly one of time consumption; while not a particularly hard process, it took a while to determine the edge cases of the conditional jumps.

Program Length

The re-written program is only marginally shorter than the original program at a meager 106 lines (vs. 114 lines originally). However, a simple line count fails to take into account the change in comment density as well. The original program relied heavily on the use of comments to explain the substructures and jumps, whereas the redesign is largely self-documenting. Since the new code is simpler, it needs significantly less comments to help the reader understand the logic.

Next Steps

This new program attempts to match the exact logic of the previous program as possible. The only visible differences encountered in black box testing is that the nature of the input prompts have changed to be more descriptive. To further improve this program, I would recommend divorcing the input/output from the translation, and encapsulating the latter in a subroutine. This would allow the program to act as a library and be used by other programs. Any program that is unable to be integrated elsewhere is doomed to become yet more “legacy” code.