COSC561 – M3.10.3 Programming Assignment 2 - cexpr

Giuseppe Schintu

University of Tennessee, Knoxville

COSC561 – M3.10.3 Programming Assignment 2 - cexpr

With some previous experience on yacc and bottom-up parsing lecture from Module 4, I did quick research on some ideas for yacc and arithmetic operations. I was able to figure out that a good case scenario would be to break down the grammar into self-sufficient productions with simple rules. I set the value [factor] and [term] rules quickly. The alphabet, dump and clear I had to research a bit and was able to find some articles and videos explaining how to go about it. Since an algebraic expression in its simplest form has numbers, variables, and prefix/postfix operands, I opted to use production rules for factors, terms, and ultimately arithmetical expressions. I had some issues getting yacc to process rules correctly when using productions like term '/' term; for some reason it forced me to declare a token of DIVD with a value of '/'. Ultimately, I had to create tokens for all the operators and assign-operators' symbols as the [expr] production was finding the '/*%=' and '/*%' used in [term] ambiguous.

It took me some trial and error and some additional research to figure out and clean out the [logical_expression] production connected to the [arith_expr] production. I had the [logical_expression] rule working with numbers but not with [arith_expr].

I believe that I was fortunate to lock my attention over the value, factor, and term composition rules, because the alternative would have been to code each case in one expr production with the risk of having to spend much more time over ambiguous rules as many of the symbols were repeated.

Overall, I am pleased that it looks clean and simple enough. Of course, I am not sure if there is a much better way to do this (aside from using associativity and precedence of operators).