Compiler Construction

BPDC

(Lab - 03)

1 Extending the Syntax Analyser for your mini-compiler

To begin with, finish off the implimentations of *insert*() and *lookup*() functions as instructed in your previous experiment and test your implementations for the given test cases in the given extended version of the problem.

You may have to use *conflict resolution rules* to handle the following exercises. Further, modify your lex program in tandem whenever required.

In the following problems, you could assume the *operands to be variables*. Further, refer previous programs, code provided with, and tutorials to pick the right construct to use. The key in designing the grammar is to identify the recursive structure associated with the target syntactic construct for which the grammar being build for.

1. The program given in the previous lab does type checking (for data types *int* and *char*), given you implement the functions *insert*() and *lookup*() properly. Modify the given program to incorporate the same for $float$ and *double* data types too.
2. Incorporate binary operator ˆ (exponent) into your compiler. Note that the exponent operator is a right associative operator and has higher precedence than all those arithmetic operators $(+, -, *, /)$ that are already defined in your compiler (refer to these definitions and incorporate the modifications).
3. Incorporate unary increment $++$ and decrement $--$ operators too (are of highest precedence).
4. Do the same for modulo operator (%), has the same precedence as * and / operators. In this case, further, we are supposed to ensure that both operands are of type "int" (you have to modify the action part of the respective grammar rule, referring to the code provided with).
5. In the given implementation, the input C file is expected to have all declaration statements in the beginning, followed by program statements. Rewrite your grammar to let the user to have declarative and program statements in any arbitrary interleaved order in their input C program.
6. Update your rules section to incorporate logical expressions in your program. A logical expression (composite) could be defined as a composition of *simple logical expressions* connected by any of the binary connectives && or ||, or the unary connective !(not). Further, each *simple logical expression* is one of the "binary" comaprison operators in the set $\{\geq, \leq, <, >, !=, ==\}$ together with operands (Eg: $(x \geq y)\&\&!(y == z)||(ptr! = NULL)$).
7. Incorporate $if(logical\_expresion)\{...\}else\{...\}$ construct in to your compiler.