CMSC 430 Project 2

The second project involves modifying the syntactic analyzer for the attached compiler by adding to the existing grammar. The full grammar of the language is shown below. The highlighted portions of the grammar show what you must either modify or add to the existing grammar.

```
function:
  function_header {variable} body

function_header:
  FUNCTION IDENTIFIER [parameters] RETURNS type ;

variable:
  IDENTIFIER : type IS statement

parameters:
  parameter {, parameter}

parameter:
  IDENTIFIER : type

type:
  INTEGER | REAL | BOOLEAN

body:
  BEGIN statement END ;

statement:
  expression ; |
  REDUCE operator {statement} ENDREDUCE ; |
  IF expression THEN statement ELSE statement ENDIF ; |
  CASE expression IS {case} OTHERS ARROW statement ; ENDCASE ;

operator:
  ADDOP | MULOP

case:
  WHEN INT_LITERAL ARROW statement

expression:
  ( expression ) |
  expression binary_operator expression |
  NOT expression |
  INT_LITERAL | REAL_LITERAL | BOOL_LITERAL |
  IDENTIFIER

binary_operator: ADDOP | MULOP | REMOP | EXPOP | RELOP | ANDOP | OROP
```

In the above grammar, the red symbols are nonterminals, the blue symbols are terminals and the black punctuation are EBNF metasymbols. The braces denote repetition 0 or more times and the brackets denote optional.

You must rewrite the grammar to eliminate the EBNF brace and bracket metasymbols and to incorporate the significance of parentheses, operator precedence and associativity for all operators. Among arithmetic operators the exponentiation operator has highest precedence following by the multiplying operators and then the adding operators. All relational operators have the same precedence. Among the binary logical operators, `and` has higher precedence than `or`. Of the categories of operators, the unary logical operator has highest precedence, the arithmetic operators have next highest precedence, followed by the relational operators and finally the binary logical operators. All operators except the exponentiation operator are left associative. The directives to specify precedence and associativity, such as `%prec` and `%left`, may not be used

Your parser should be able to correctly parse any syntactically correct program without any problem.

You must modify the syntactic analyzer to detect and recover from additional syntax errors using the semicolon as the synchronization token. To accomplish detecting additional errors an error production must be added to the function header and another to the variable declaration.

Your bison input file should not produce any shift/reduce or reduce/reduce errors. Eliminating them can be difficult so the best strategy is not introduce any. That is best achieved by making small incremental additions to the grammar and ensuring that no addition introduces any such errors.

An example of compilation listing output containing syntax errors is shown below:

```
   1  -- Multiple errors
   2
   3  function main a integer returns real;
Syntax Error, Unexpected INTEGER, expecting ':'
   4      b: integer is * 2;
Syntax Error, Unexpected MULOP
   5      c: real is 6.0;
   6  begin
   7      if a > c then
   8          b  3.0;
Syntax Error, Unexpected REAL_LITERAL, expecting ';'
   9      else
  10          b = 4.;
  11      endif;
  12  ;
Syntax Error, Unexpected ';', expecting END

Lexical Errors 0
Syntax Errors 4
Semantic Errors 0
```

You are to submit two files.

- The first is a `.zip` file that contains all the source code for the project. The `.zip` file should contain the flex input file, which should be a `.l` file, the bison file, which should be a `.y` file, all `.cc` and `.h` files and a `makefile` that builds the project.
- The second is a Word document (PDF or RTF is also acceptable) that contains the documentation for the project, which should include the following:
    a. A discussion of how you approached the project
    b. A test plan that includes test cases that you have created indicating what aspects of the program each one is testing and a screen shot of your compiler run on that test case
    c. A discussion of lessons learned from the project and any improvements that could be made

Grading Rubric

Criteria	Meets	Does Not Meet
Functionality	**70 points**	**0 points**
	Parses all syntactically correct programs (25)	Does not parse all syntactically correct programs (0)
	Productions correctly implement precedence and associativity (10)	Productions do not correctly implement precedence and associativity (0)
	Grammar contains no shift/reduce or reduce/reduce errors (5)	Grammar contains shift/reduce or reduce/reduce errors (0)
	Detects and recovers from all programs with single syntax errors (20)	Does not detect and recover from errors in the function header (0)
	Detects and recovers from a program with multiple syntax errors (10)	Does not detect and recover from multiple errors (0)
Test Cases	**15 points**	**0 points**
	Includes test cases that test all grammar productions (6)	Does not include test cases that test all grammar productions (0)
	Includes test cases that test errors in all productions (6)	Does not include test cases that test errors in all productions (0)
	Includes test case with multiple errors (3)	Does not include test case with multiple errors (0)
Documentation	**15 points**	**0 points**
	Discussion of approach included (5)	Discussion of approach not included (0)
	Lessons learned included (5)	Lessons learned not included (0)
	Comment blocks with student name, project, date and code description included in each file (5)	Comment blocks with student name, project, date and code description not included in each file (0)