Here is the recommended approach for project 2.

1) First build the skeleton for project 2, run it and be sure that you understand how it works.

2) Replace the lexical analyzer in the project 2 skeleton with your lexical analyzer from project 1. Be sure to remove the main method from your `scanner.l`. Also replace the `listing.h` and `listing.cc` files in the skeleton with the ones from your version of project 1. The add the necessary `%token` declarations for the new tokens. Note that the `tokens.h` from project 1 is no longer needed. It will be generated by bison from the `%token` declarations. Verify that the project builds correctly at this point. Then confirm that test cases `test1.txt` - `test4.txt` that were provided as test cases for the skeleton code parse properly and that the test case `syntax1.txt` produces a syntax error.

3) The simplest modification to make first is to modify the grammar to include variables and literals of the real data type. You will need to modify the `type` and `primary` productions. Use `test5.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test5.txt

   1  // Function with a Real and Character Variables and Literals
   2
   3  function main returns character;
   4      a: real is 7.8e-1;
   5  begin
   6      when a < .45, '\n' : 'A';
      end;
Compiled Successfully
```

4) Next, add the `if` statement to the `statement` production in the grammar. The `if` statement allows zero or more `elsif` clauses represented in EBNF using braces. The grammar in the specification defines the language but is not in the form needed for bison. It contains EBNF symbols, that must be removed. In that part of the grammar, the EBNF braces are used to indicate that a `if` statement contains zero or more `elsif` clauses. Because bison does not support these EBNF symbols, a recursive production must be used instead. Study how the recursive `cases` production in the skeleton defines zero or more `case` statements. You should use a similar approach.

Be sure that you understand the difference between the meaning of `;` and `';'` in the bison input file. The former is the symbol that terminates a production. The latter represents the semicolon symbol in the target language as does the `;` in the project specification. It is also important to distinguish between the `statement` and the `statement_` productions that were provided in the skeleton code. The latter incorporates the semicolon that ends the `if` statement in the target language and also provides recovery should an error occur within a statement. Use `test6.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test6.txt
```

```
   1  // Function with an If Statement
   2
   3  function main returns integer;
   4      a: integer is #A;
   5  begin
   6      if a < 10 then
   7          1;
   8      elsif a < 20 then
   9          2;
  10      elsif a < 30 then
  11          3;
  12      else
  13          4;
  14      endif;
  15  end;

Compiled Successful
```

5) The `fold` statement would be best implemented next. Note that it will require two subordinate production `direction` and `operator`. Use `test7.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test7.txt

   1  // Left and Right Fold Statement
   2
   3
   4  function main returns integer;
   5      a: list of integer is (1, 2, 3);
   6  begin
   7      switch a(1) is
   8          case 1 =>
   9              fold right - (2,3, 4) endfold;
  10          case 2 =>
  11              fold left + a endfold;
  12          others =>
  13              0;
  14      endswitch;
  15  end;

Compiled Successfully
```

6) The provided skeleton allows an optional, 0 or 1, variable declaration. The requirements state that 0 or more should be permitted. This modification again requires replacing the EBNF braces with a recursive production. Use `test8.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test8.txt

   1  // Multiple Integer Variable Initializations
   2
   3  function main returns integer;
   4      b: integer is 5 + 1 - 4;
   5      c: integer is 2 + 3;
   6  begin
   7      b + 1 - c;
   8  end;

Compiled Successfully
```

7) The next feature to add is 0 or more parameter declarations in the function header. Because the parameters require comma separators, notice that the grammar is written to indicate that the parameters are optional because the parameters production must contain at least one. Study how optional elements are included in the grammar by noticing how an optional variable declaration was implemented in the skeleton. Also study how the elements production is implemented. Like the parameters, it involves comma separators. Two test cases are provided to test this change. Before using either of them, use one of the early test cases to confirm that your parser still allows no parameters. Then proceed to use, `test9.txt`, to verify that program with one parameter declaration parses correctly. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test9.txt

   1  // Single Parameter Declaration
   2
   3  function main a: integer returns integer;
   4  begin
   5      a + 1;
   6  end;

Compiled Successfully
```

The next test case, `test10.txt` contains two parameter declarations. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test10.txt

   1  // Two Parameter Declarations
   2
   3  function main a: integer, b: real returns real;
   4      c: real is .7;
   5  begin
   6      a + b * c;
   7  end;

Compiled Successfully
```

8) The additional binary arithmetic operators for remainder and exponentiation and the unary minus operator should be added next. Notice how the existing operators are implemented in the skeleton code. There must be one production for each level of precedence. Because the remainder operator has the same precedence as the multiplying operators, it should be included as another right-hand side to the existing production. But because the exponentiation operator has higher precedence a production must be introduced. Because that operator is right associative, its production must be right recursive. Because the unary minus operator has higher precedence than all the binary arithmetic operators. In the *Course Resources* area of the classroom you will find Module 1 from CMSC 330. You may want to read section II-B if you

need a better understanding of how to construct a grammar so that it produces the correct parse tree to account for precedence and associativity.

Then proceed to use, `test11.txt`, to verify that program containing every arithmetic operator parses correctly. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test11.txt

   1  // Arithmetic Operators
   2
   3  function main returns integer;
   4  begin
   5      9 + ~2 - (5 - 1) / 2 % 3 * 3 ^ 1 ^ 2;
   6  end;

Compiled Successfully
```

9) The additional logical operators, which include the `|` and `!` operators should be added next. Each has a separate precedence level. The `|` operator has the lowest precedence of all operators and `!` has the highest. So, two new productions must be added to the grammar. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test12.txt

   1  // Relational and Logical Operators
   2
   3  function main returns integer;
   4  begin
   5      when 5 > 8 & 3 = 3 | 9 < 1 & !(3 <> 7) | 6 <= 7 & 3 >= 9, 1 : 0;
   6  end;

Compiled Successfully
```

10) At this point your parser should contain the complete grammar for the language. As a final check, use test cases `test13.txt`, which contains a nested `if` together with most elements of the language and `test14.txt`, which contains a nested `switch` statement. Shown below is the output that should result when using `test13.txt` as input:

```
$ ./compile < test13.txt

   1  // Comprehensive Test with Nested If
   2
   3  function main a: integer, b: character, c: real returns real;
   4      d: integer is #8e;
   5      e: real is 3.75;
   6      f: character is 'A';
   7      g: list of integer is (1, 3, 5);
   8  begin
   9      if ~a > 5 & a < 1 & !(c = 5.8 | c <> .7E4) then
  10          if c >= 7.5E-2 & c <= 5.2 then
  11              when a >= d, a + 2 - 7.9E+2 / 9 * 4 : 8.9;
  12          elsif g(1) = a ^ 2 % 3 then
  13              a % 2 - 5 / c;
  14          else
  15              fold left + (1, 2, 3) endfold;
  16          endif;
  17       else
  18           fold right - g endfold;
  19       endif;
  20  end;

Compiled Successfully
```

Shown below is the output that should result when using `test14.txt` as input:

```
$ ./compile < test14.txt

   1  // Comprehensive Test with Nested Switch
   2
   3  function main a: integer, b: real returns real;
   4      c: integer is 8;
   5      d: real is .7E2;
   6  begin
   7      switch a is
   8           case 1 => a * 2 / d ^ 2;
   9           case 2 => a + 5.3E+2 - b;
  10           case 3 =>
  11               switch d is
  12                   case 1 => a % 2;
  13                   others => 9.1E-1;
  14               endswitch;
  15           case 4 => a / 2 - c;
  16           others => a + 4.7 * b;
  17      endswitch;
  18  end;

Compiled Successfully
```

11) At this point, you are ready to implement the error recovery portion of the program. Verify first that test case `syntax1.txt` that was include with the project 2 test data produces the correct output. Then modify the grammar so that errors in the function header will be properly recovered from. You will need to introduce an additional production that includes the `error` token. You should use the `statement_` production in the skeleton code as a model. Once you have implemented the error recovery production for the function header, use test case `syntax2.txt` to test it. Shown below is the output that should result when using `syntax2.txt` as input:

```
$ ./compile < syntax2.txt

   1 // Error in Function Header, Missing Colon
   2
   3  function main a integer returns integer;
Syntax Error, Unexpected INTEGER, expecting ':'
   4      b: integer is 3 * 2;
   5  begin
   6      if a <= 0 then
   7          b + 3;
   8      else
   9          b * 4;
  10      endif;
  11  end;

Lexical Errors 0
Syntax Errors 1
Semantic Errors 0
```

The fact that it continued to parse the remainder of the program is confirmation that it has been properly implemented.

12) Next implement error recovery in variable declarations. As before, an additional production is required. Once you have implemented the error recovery production for variable declarations, use test case `syntax3.txt` to test it. Shown below is the output that should result when using that test case as input:

```
$ ./compile < syntax3.txt

   1  // Error in Variable Declaration
   2  
   3  function main a: integer returns integer;
   4      b integer is
Syntax Error, Unexpected INTEGER, expecting ':'
   5          if a > 5 then
   6              a * 3;
   7          else
   8              2 + a;
   9          endif;
  10      c: real is 3.5;
  11  begin
  12      if a <= 0 then
  13          b + 3;
  14      else
  15          b * 4;
  16      endif;
  17  end;

Lexical Errors 0
Syntax Errors 1
Semantic Errors 0
```

Confirmation that error recovery has occurred is that no extraneous error messages are generated after the variable declaration.

13) The final addition to the grammar to enhance error recovery is to an error recovery to the `case` clause of the `switch` statement. As before, an additional production is required. Use test case `syntax4.txt` to test it. Shown below is the output that should result when using that test case as input:

```
$ ./compile < syntax4.txt

   1  // Multiple Errors, Error in Case Clause and Missing Others Clause
   2  
   3  function main a: integer returns integer;
   4  begin
   5      switch a is
   6           case 1 => a 2;
Syntax Error, Unexpected INT_LITERAL, expecting ';'
   7           case 2 => 5;
   8      endswitch;
Syntax Error, Unexpected ENDSWITCH, expecting CASE or OTHERS
   9  end;

Lexical Errors 0
Syntax Errors 2
Semantic Errors 0
```

The fact that the second error, the one indicating that the `others` clause is missing, confirms that recovery from the first error message has occurred.

14) As a final test use test case `syntax5.txt`, which contains a variety of syntax errors. Shown below is the output that should result when using that test case as input:

```
$ ./compile < syntax5.txt

   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 + / 4.7;
Syntax Error, Unexpected MULOP
   9      else
  10          switch b is
  11              case => 2;
Syntax Error, Unexpected ARROW, expecting INT_LITERAL
  12              case 2 => c;
  13          endswitch;
Syntax Error, Unexpected ENDSWITCH, expecting CASE or OTHERS
  14      endif;
  15  end;

Lexical Errors 0
Syntax Errors 5
Semantic Errors 0
```

If you have implemented all the required error recovery, your compiler should detect all five syntax errors.

All of the test cases discussed above are included in the attached .zip file.

You are certainly encouraged to create any other test cases that you wish to incorporate in your test plan. Keep in mind that your parser should parse all syntactically correct programs, so it is recommended that you choose some different test cases as a part of your test plan. Your instructor may use a comparable but different set of test cases when testing your project.