Here is the recommended approach for project 3.

1) Study the skeleton project provided, build it and run it so that you understand how semantic actions are used to perform the evaluation needed to interpret programs in our language.

2) You should incorporate the new features that are in the skeleton into your version of project 2 and be sure that it builds and runs just as the skeleton did. Then confirm that test cases `test1.txt - test4.txt` that were provided as test cases for the skeleton code produce the correct output. Note that changes are required to both `parser.y` and `scanner.l`. You are likely to get type clash warnings from bison initially. These will be resolved once the entire project is completed.

3) Make additions as defined by the specification incrementally. Start with adding the code to evaluate real literals and the hexadecimal integer literals. These changes involve modifications to `scanner.l`. There is a predefined C++ function `atof` that will convert a string containing a real literal to a `double`. But you will need to write a function that converts a string containing a hexadecimal integer to an `int`. Once you have made these modifications use `test5.txt` test them. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test5.txt

   1  // Function with Arithmetic Expression using Real Literals
   2  //     and Hexadecimal Integer Literals
   3
   4  function main returns real;
   5  begin
   6      .83e+2 + 2.5E-1 + (4.3E2 + #aF2) * .01;
   7  end;

Compiled Successfully

Result = 115.57
```

4) Next, add the code to `scanner.l` to evaluate character literals that includes both ordinary character literals and the escape characters. As with the hexadecimal integer literals, you will need to write a function to perform this conversion. Once you have made this modification use `test6.txt` test it. Shown below is the output that should result when using that test case as input:

```
   1  // Function with Character Literal Escape Characters
   2
   3  function main returns character;
   4      lines: integer is 60;
   5  begin
   6      when lines < 60, '\n' : '\f';
   7  end;

Compiled Successfully

Result = 12
```

5) Next add the necessary semantic actions for each of the new arithmetic operators. Create individual test cases to test each new operator. Once you have verified that the operators are evaluated correctly in those cases, use `test7.txt`, which will test all of them along with their precedence. Modifications to `scanner.l`, `parser.y`, `values`.h and `values.cc` are required Shown below is the output that should result when using that test case as input:

```
$ ./compile < test7.txt

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

Compiled Successfully

Result = 5
```

6). Do the relational operators next. As before it is a good idea to create individual test cases to test each new operator. Finally the two logical operators `|` and `!` testing each with separate test cases. Once you have added all the new operators use `test8.txt` to test them. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test8.txt

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

Compiled Successfully

Result = 0
```

7) The `if` statement would be a good next step. Study how the `when` statement is implemented using the conditional expression operator. A similar approach should be used here. Also study how the case statements are implemented. The `elsif` statements should be done in a similar fashion. Use `test9.txt` to test it. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test9.txt

   1  // Function with an If Statement
   2
   3  function main returns integer;
   4      a: integer is 28;
   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 Successfully

Result = 3
```

To ensure all cases work correctly, modify the value of the variable a, so each case is tested.

8) Test multiple variable declarations next. No modifications should be needed provided you are using the grammar from project 2. Use `test10.txt` to confirm that they work correctly. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test10.txt

   1  -- Multiple Variable Initializations
   2
   3  function main returns character;
   4      b: integer is 5 + 1 - 4;
   5      c: real is 2 + 3.7;
   6      d: character is 'A';
   7  begin
   8      if b + 1 - c > 0 then
   9          d;
  10      else
  11          '\n';
  12      endif;
  13  end;

Compiled Successfully

Result = 10
```

9) Next make the changes necessary for programs that contain parameters. The parameters are in the command line arguments, `argv`. The prototoype of main is:

```
int main(int argc, char *argv[])
```

Declare a global array, which is dynamically allocated based on `argc`, at the top of `parser.y`. In main convert each command line argument to a `double` and store it into that global array. The function `atof` will do the conversion of a `char*` to a `double`. In the semantic action for the `parameter` production, retrieve the value of the corresponding parameter from the global array and store its value in the symbol table.

Use `test11.txt` and `test12.txt` to test that change. Shown below is the output that should result when using both test cases as input:

```
$ ./compile < test11.txt 6.8

   1  // Single Parameter Declaration
   2
   3  function main a: real returns real;
   4  begin
   5      a + 1.5;
   6  end;

Compiled Successfully

Result = 8.3

$ ./compile < test12.txt 16 15.9

   1  // Two parameter declarations
   2
   3  function main a: integer, b: real returns real;
   4  begin
   5      if a < #A then
   6          b + 1;
   7      else
   8          b - 1;
   9      endif;
  10  end;

Compiled Successfully

Result = 14.9
```

10) Save the `fold` statement for last. It is likely the most unfamiliar statement. Be sure to read the description in the requirements on how such statements are evaluated. Creating a new function in `values.cc` to evaluate the `fold` statement is the best approach.

Use `test13.txt` to test the right `fold` statement using a list variable.. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test13.txt

   1  // Test Right Fold
   2
   3  function main returns integer;
   4      values: list of integer is (3, 2, 1);
   5  begin
   6      fold right - values endfold;
   7  end;

Compiled Successfully

Result = 2
```

Use `test14.txt` to test the left `fold` statement using a list literal. . Shown below is the output that should result when using that test case as input:

```
$ ./compile < test14.txt

   1  // Test Left Fold
   2
   3  function main returns integer;
   4  begin
   5      fold left - (3, 2, 1) endfold;
   6  end;

Compiled Successfully

Result = 0
```

11) The final test cases, `test15.txt` contains all the statements in the language. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test15.txt 1 2.5 65

   1  // Test that Includes All Statements
   2
   3  function main a: integer, b: real, c: character returns real;
   4      d: integer is when a > 0, #A: #A0;
   5      e: list of integer is (3, 2, 1);
   6      f: integer is
   7             switch a is
   8                 case 1 => fold left - e endfold;
   9                 case 2 => fold right - e endfold;
  10                 others => 0;
  11             endswitch;
  12      g: integer is
  13             if c = 'A' & b > 0 then
  14                 a * 2;
  15             elsif c = 'B' | b < 0 then
  16                 (a ^ 2) * 10;
  17             else
  18                 0;
  19             endif;
  20  begin
  21      f + (d - 1) % g;
  22  end;

Compiled Successfully

Result = 1
```

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 compiler should produce the correct output for 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.