Here is the recommended approach for project 4.

1) Although you have the option of keeping the interpreter from project 3 in the final project, it is a more complicated approach. Unless you are an experienced C++ programmer, it will be simpler to take your project 2 implementation and begin by incorporating what is in the project 4 skeleton, much like what you should have done for project 3. Because your project 2 has additional productions, be sure to include them in your `%type` declaration. You are likely to have some type clash warnings at this point. You initially should be able to ignore them. Once you complete the project, they should all be resolved, however.

At this point, you should verify that the test cases provided with the project 4 skeleton, `semantic1.txt` – `semantic7.txt`, produce the same output as before.

A good starting point would be to incorporate real types. To accomplish that you will need to modify the `Types` enumeration in `types.h`, assign an attribute to real literal tokens in `scanner.l` and modify the `type` production in `parser.y`. To verify that your program accepts real types, use `valid1.txt`. Shown below is the output that should result when using that test case as input:

```
$ ./compile < valid1.txt

   1  -- Program with a Real Variable
   2
   3  function main returns real;
   4      a: real is 4.5;
   5  begin
   6      a;
   7  end;

Compiled Successfully
```

The fact that it compiles successfully confirms that real types have been added.

2) Next ensure that hexadecimal literals are identified as type integer. Assigning an attribute to hexadecimal integer literal in `scanner.l` is required. To verify that your program accepts hexadecimal literals as integers, use `valid2.txt`. Shown below is the output that should result when using that test case as input:

```
$ ./compile < valid2.txt

   1  -- Program with a Hexadecimal Literals
   2
   3  function main returns integer;
   4      a: integer is #A;
   5  begin
   6      a + #a;
   7  end;

Compiled Successfully
```

The fact that it compiles successfully confirms that hexadecimal literals are treated as integers.

3) The next requirement to implement is to ensure that type coercion from an integer to a real type is performed within arithmetic expressions. That will require that you modify the `checkArithmetic` function so that it returns a real type when one operand is integer and the other is real. It should, of course, still return an integer type when both types are integer. To verify that your program performs type coercion, use `valid3.txt`. Shown below is the output that should result when using that test case as input:

```
$ ./compile < valid3.txt

   1  -- Program with a Real Variable
   2
   3  function main returns real;
   4      a: real is 4 + 4.5;
   5  begin
   6      a;
   7  end;

Compiled Successfully
```

The fact that it compiles successfully confirms that type coercion has been incorporated.

4) Next, include a check to ensure that lists contain elements that are all the same type. Adding a semantic action to the `expressions` production is required. Creating a new function in `types.cc` is a good idea. This check is similar to the check made to ensure all `case` statements have the same type, but is somewhat simpler because all lists must contain at least one clement. Use `semantic8.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic8.txt

   1  // List with Elements of Different Types
   2
   3  function main returns integer;
   4      aList: list of integer is (1, 2, 3.5);
Semantic Error, List Element Types Do Not Match
   5  begin
   6      aList(1);
   7  end;

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

5) Next, include a check to ensure that the type of a list variable matches the type of its elements. Adding a semantic action to the `variable` production whose right-hand side declares lists is required. Creating a new function in `types.cc` is a good idea. Use `semantic9.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic9.txt

   1  // List Type Does Not Match Element Types
   2
   3  function main returns character;
   4      aList: list of character is (1, 2, 3);
Semantic Error, List Type Does Not Match Element Types
   5  begin
   6      aList(1);
   7  end;

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

6) Include a check to ensure that the type of a list subscript is an integer expression. Adding a semantic action to the `primary` production whose right-hand side defines subscripted lists is required. Creating a new function in `types.cc` is a good idea. Use `semantic10.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic10.txt

   1  // List Subscript is not Integer
   2
   3  function main returns integer;
   4      aList: list of integer is (1, 2, 3);
   5  begin
   6      aList(1.5);
Semantic Error, List Subscript Must Be Integer
   7  end;

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

7) Include a check to ensure that character literals can be compared to other character literals but cannot be compared to numeric expressions. Adding a semantic action to the `relation` is required. Creating a new function in `types.cc` is a good idea. Use `semantic11.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic11.txt

   1  -- Mixing Numeric and Character Types with Relational Operator
   2
   3  function main returns integer;
   4  begin
   5      if 'b' < 'c' then
   6          1;
   7      elsif 'b' < 1 then
Semantic Error, Character Literals Cannot be Compared to Numeric Expressions
   8          2;
   9      else
  10          3;
  11      endif;
  12  end;

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

8) Because the skeleton did not include productions for all operators, you need to add semantic actions to those productions. The exponentiation operator and the arithmetic negation operators both require additional checks. Use `semantic12.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic12.txt

   1  // Using Character Literal with Exponentiation Operator
   2  //     and Negation Operator
   3
   4  function main returns integer;
   5      c: character is ~'c';
Semantic Error, Arithmetic Operator Requires Numeric Types
   6  begin
   7      5 ^ 'P';
Semantic Error, Arithmetic Operator Requires Numeric Types
   8  end;
   9

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

The fact that two error messages have been generated confirms that the productions have been successfully modified.

9) Because the remainder operator was not in the skeleton, it is necessary to implement the test that determines whether the types on the `%` operator are integers. Adding a new function to `types.cc` is again a good idea, given that this check requires more than a single line of code. Use `semantic13.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic13.txt

   1  // Mixing Real Literals with the Remainder Operator
   2
   3  function main returns integer;
   4  begin
   5      4 % 4.8;
Semantic Error, Remainder Operator Requires Integer Operands
   6  end;

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

10) The semantic check on the `if` statement would be a good next step. This check can be accomplished by adding a semantic actions on the various productions that define the `if` statement and ideally adding a new function in `types.cc`. Use `semantic13.txt` to test this semantic check. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic14.txt
```

```
   1  -- If Elsif Else Mismatch
   2
   3  function main returns integer;
   4  begin
   5      if 9 < 10 then
   6          1;
   7      elsif 8 = 1 then
   8          2;
   9      else
  10          3.7;
Semantic Error, If-Elsif-Else Type Mismatch
  11      endif;
  12  end;

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

11) Next, implement the rule associated with the `fold` statement that requires that the type of the list be numeric. Use `semantic15.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic15.txt

   1  // Folding a nonnumeric List
   2
   3  function main returns integer;
   4  begin
   5      fold left + ('a', 'b', 'c') endfold;
Semantic Error, Fold Requires A Numeric List
   6  end;

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

12) The next best pair of checks to implement are the ones that check whether a variable initialization or function return are narrowing. Both require changing the `checkAssignment` function. Use `semantic16.txt` to test the check for narrowing variable initialization. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic16.txt

   1  -- Narrowing Variable Initialization
   2
   3  function main returns integer;
   4      b: integer is 5 * 2.5;
Semantic Error, Illegal Narrowing Variable Initialization
   5  begin
   6      b + 1;
   7  end;

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

13) You should be able to use the same function to check a narrowing function return by adding a similar semantic action on the top level production for the whole function. Be sure that you have declared that the `function_header` and `body` productions carry a type attribute if your receive an error indicating that they have no declared type. In addition both require semantic actions that pass the type information up the parse tree. Use `semantic17.txt` to test the check for a narrowing function return. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic17.txt

   1  -- Narrowing Function Return
   2
   3  function main returns integer;
   4      b: integer is 6 * 2;
   5  begin
   6      if 8 < 0 then
   7          b + 3.0;
   8      else
   9          b * 4.6;
  10      endif;
  11  end;
Semantic Error, Illegal Narrowing Function Return

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

14) The check for duplicate scalar and list variables is a good one to incorporate next. Because the symbol table already contains a function to look up identifiers, no modification to the symbol table code is required. Instead the semantic action for variable declarations can be modified to first check whether the identifier is in the symbol table, and if so, generate a duplicate identifier error. Writing a separate function that can be passed which symbol table to use will avoid duplicating code. Use `semantic18.txt` to test the check for a duplicate identifier. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic18.txt

   1  -- Duplicate Scalar and List Variables
   2
   3  function main returns integer;
   4      scalar: integer is 4 * 2;
   5      scalar: character is 'b';
Semantic Error, Duplicate Scalar scalar
   6      a_list: list of integer is (4, 2);
   7      a_list: list of real is (2.3, 4.4);
Semantic Error, Duplicate List a_list
   8  begin
   9      1;
  10  end;

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

15) As a final test, use `semantic19.txt` to test a program that contains multiple semantic errors. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic19.txt

   1  // Multiple Semantic Errors
   2
   3  function main returns integer;
   4      value: integer is 4.5;
Semantic Error, Illegal Narrowing Variable Initialization
   5      numbers: list of real is (1, 2, 3);
Semantic Error, List Type Does Not Match Element Types
   6      one: integer is '1';
Semantic Error, Type Mismatch on Variable Initialization
   7  begin
   8      if value > 0 then
   9          fold left + ('a', 'b') endfold;
Semantic Error, Fold Requires A Numeric List
  10      elsif name = 'N' then
Semantic Error, Undeclared Scalar name
  11          fold right * (1, 2.5) endfold;
Semantic Error, List Element Types Do Not Match
  12      else
  13          when value < 10, 1 : 1.5;
Semantic Error, When Types Mismatch
  14      endif;
  15  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 7
```

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 semantic error for all programs that contain them, 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.