Here is the approach I recommend for project 4.

1) Although you have the option of keeping the interpreter from project 3 in the final project, it is a slightly 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` - `semantic6.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;
   8

Compiled Successfully
```

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

2) 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 `valid2.txt`. Shown below is the output that should result when using that test case as input:

```
$ ./compile < valid2.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;
   8
```

```
Compiled Successfully
```

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

3) Next, fully incorporate the Boolean type by assigning an attribute to the Boolean literals tokens in `scanner.l`. Also, generalize the error message that is produced when mixing Booleans with arithmetic operators and arithmetic expressions that contain real values to indicate that a numeric type is required rather than just an integer type. Use `semantic7.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic7.txt

   1  -- Using Boolean Literal with Arithmetic Operator
   2
   3  function main returns real;
   4      a: real is 8.7 + false;
Semantic Error, Numeric Type Required
   5  begin
   6      9.5 + true;
Semantic Error, Numeric Type Required
   7  end;

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

The fact that an error message has been generated confirms that Boolean literals have been given a type. Also, notice the error message has been generalized to allow for both integer and real types.

4) Because the skeleton did not include productions for all operators, you need to add semantic actions to those productions. The exponentiation operator is one of them. 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  -- Using Boolean Expression with Exponentiation Operator
   2
   3  function main returns integer;
   4  begin
   5      5 ** (1 > 0);
Semantic Error, Numeric Type Required
   6  end;
   7

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

The fact that an error message has been generated confirms that production has been successfully modified.

5) The `or` is another one of them. 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  -- Using Arithmetic Expression with Logical Or Operator
   2
   3  function main returns integer;
   4  begin
   5      9 or 6 < 3;
Semantic Error, Boolean Type Required
   6  end;

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

The fact that an error message has been generated confirms that production has been successfully modified.

6) The `not` is the last one of them. 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  -- Using Arithmetic Expression with Logical Not Operator
   2
   3  function main returns integer;
   4  begin
   5      not 3;
Semantic Error, Boolean Type Required
   6  end;

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

The fact that an error message has been generated confirms that production has been successfully modified. This change should also eliminate one of the type clashes from bison.

7) At this point you are ready to incorporate the code to check for the remaining semantic errors. A good first choice would be to implement the test that determines whether the types on the rem operator are integers. Adding a new function to `types.h` and `types.cc` is best, given that this check requires more than a single line of code. 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 Real literals with the Remainder Operator
   2
   3  function main returns real;
   4  begin
   5      4 rem 4.8;
Semantic Error, Remainder Operator Requires Integer Operands
```

```
   6  end;

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

8) The two semantic checks on the `if` statement would be a good next step. The first required check is to ensure that the type of the expression is Boolean. Both of these checks can be accomplished by adding a semantic action on the `if` statement and ideally a new function in `types.cc`. Use `semantic12.txt` to test the first of these two.. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic12.txt

   1  -- If Condition Not Boolean
   2
   3  function main returns integer;
   4      b: integer is 7 * 2;
   5  begin
   6      if 3 + 6 then
   7          b + 3;
   8      else
   9          b * 4;
  10      endif;
Semantic Error, If Expression Must Be Boolean
  11  end;

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

9) The other check is to determine whether the types on the two parts of an `if` statement match. 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  -- Conditional Expression Type Mismatch
   2
   3  function main returns integer;
   4  begin
   5      if 6 < 0 then
   6          5 + 3.0;
   7      else
   8          8 * 4;
   9      endif;
Semantic Error, If-Then Type Mismatch
  10  end;

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

10) 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 `semantic14.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 < semantic14.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
```

11) 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 `semantic15.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 < semantic15.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
```

12) The check for duplicate 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. Use `semantic16.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 < semantic16.txt
```

```
   1  -- Duplicate Identifier
   2
   3  function main returns integer;
   4      b: integer is 4 * 2;
   5      b: boolean is true;
Semantic Error, Duplicate Identifier: b
   6  begin
   7      if 8 <= 0 then
   8          b + 3;
   9      else
  10          b * 4;
  11      endif;
  12  end;

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

13) It is best to save incorporating the semantic errors for the `case` statement until last. The one to check first is the test whether the case expression is an integer. This one is the simpler of the two. Use `semantic17.txt` to test the check on the type of the case expression. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic17.txt

   1  -- Test of Non Integer Case Expression
   2
   3  function main returns integer;
   4      b: integer is 9 - 2 + 8;
   5      c: real is 9.5;
   6      d: boolean is 7 = 5;
   7  begin
   8      case c is
   9          when 1 => d;
  10          when 2 => true;
  11          others => false;
  12      endcase;
Semantic Error, Case Expression Not Integer
  13  end;
Semantic Error, Type Mismatch on Function Return

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

14) The last check to implement is the one that checks whether all the types of all the `when` clauses of the `case` statement match. The implementation is similar to what was needed in project 3. In this case, a sentinel value again is needed. This time it indicates whether an initial type has been determined. Adding another enumeration literal to the `Types` enumeration will provide one. It should become the attribute of the empty production in the definition of the list of `when` clauses. No inherited attributes are needed to implement this check. Be sure all productions connected to this statement are declared in the `%type` declaration. At this point, all type clashes

should be eliminated. Use `semantic18.txt` to test this semantic error. Shown below is the output that should result when using that test case as input:

```
$ ./compile < semantic18.txt

   1  -- Test of Case Types Mismatch
   2
   3  function main returns integer;
   4      b: integer is 9 - 2 + 8;
   5      c: real is 9.5;
   6      d: boolean is 7 = 3;
   7  begin
   8      case b is
   9          when 1 => c;
  10          when 2 => true;
Semantic Error, Case Types Mismatch
  11          others => false;
  12      endcase;
  13  end;

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

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  -- Test of Multiple Semantic Errors
   2
   3  function main returns integer;
   4      b: integer is
   5          if 6 + 5 then
   6              2;
   7          else
   8              5;
   9          endif;
Semantic Error, If Expression Must Be Boolean
  10      c: real is 9.8 - 2 + 8;
  11      d: boolean is 7 = f;
Semantic Error, Undeclared f
  12  begin
  13      case b is
  14          when 1 => 4.5 + c;
  15          when 2 => b;
Semantic Error, Case Types Mismatch
  16          others => c;
  17      endcase;
  18  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 3
```

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 I recommend that you choose some different test cases as a part of your test plan. I may use a comparable but different set of test cases when I test your projects.