Here is the approach I recommend for project 3.

1) Study the skeleton project provided, built 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`.

3) Make additions as defined by the specification incrementally. Start with adding the literals of the real and Boolean data types. These changes involve modifications to `scanner.l`. . Once you have made these modifications use `test5.txt` and `test6.txt` to test them. Shown below is the output that should result when using both test cases as input:

```
$ ./compile < test5.txt

   1  -- Function with arithmetic expression using real literals
   2
   3  function main returns real;
   4  begin
   5      8.3e+2 + 2.E-1 * (4.3E2 + 2.8) * 3.;
   6  end;

Compiled Successfully

Result = 1089.68

$ ./compile < test6.txt

   1  // Function containing Boolean literals
   2
   3  function main returns boolean;
   4  begin
   5      true and false;
   6  end;
   7

Compiled Successfully

Result = 0
```

4) 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. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test7.txt

   1  // Arithmetic operators
   2
   3  function main returns integer;
```

```
   4  begin
   5      9 + 2 - (5 - 1) / 2 rem 3 * 3 ** 1 ** 2;
   6  end;

Compiled Successfully

Result = 5
```

5). 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 `or` and `not` 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  -- Relational and logical operators
   2
   3  function main returns boolean;
   4  begin
   5      (5 + 3 > 8 or 9 / 3 = 3) or (7 - 9 < 1) and (not (3 /= 1 * 7) or 6
<= 7 and 3 >= 9);
   6  end;

Compiled Successfully

Result = 1
```

6) The `if` statement would be a good next step. It can be done in one line using the conditional expression operator. 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  -- Conditional expression
   2
   3  function main returns integer;
   4  begin
   5      if not (5 + 4 >= 9) then
   6          6 + 9 * 3;
   7      else
   8          8 - 9 rem 7;
   9      endif;
  10  end;

Compiled Successfully

Result = 6
```

7) Do multiple variable declarations next. Use `test10.txt` and `test11.txt` to test that change. Shown below is the output that should result when using both test cases as input:

```
$ ./compile < test10.txt
```

```
-- Multiple integer variable initialization

function main returns integer;
    b: integer is 5 + 1 - 4;
    c: integer is 2 + 3;
begin
    b + 1 - c;
end;
```

```
Compiled Successfully

Result = -2

$ ./compile < test11.txt
```

```
-- Variable initialization with real and Boolean variables

function main returns real;
    b: real is 5.3 + 1. - 4 / 2.0E-2;
    c: boolean is b > 3.e2 and true or false;
begin
    if c then
        b * 6.4 + 1.5;
    else
        2.e+3;
    endif;
end;
```

```
Compiled Successfully

Result = 2000
```

8) 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 `test12.txt` and `test13.txt` to test that change. Shown below is the output that should result when using both test cases as input:

```
$ ./compile < test12.txt 3.6
```

```
-- Single parameter declaration

function main a: real returns real;
begin
    a + 1.5;
```

```
end;

Compiled Successfully

Result = 5.1

$ ./compile < test13.txt 1 8.3

-- Two parameter declarations

function main a: boolean, b: real returns real;
begin
    if a then
        b + 1;
    else
        b - 1;
    endif;
end;

Compiled Successfully

Result = 9.3
```

9) Save the `case` statement for last. It is the most challenging. The approach that I recommend is using an inherited attribute. Study how an inherited attribute is used in the skeleton for the reductions. A similar approach can be used with the `case` statement.

Here is pseudo-code for semantic actions for handling the `case` statement. It uses the sentinel `NAN`, not-a-number as the attribute carried up the parse tree to indicate a match has not yet been found. In that way, the decision can be made at the top level when the value of the case should be what is in the `OTHERS` clause:

```
statement:
        CASE expression IS cases OTHERS ARROW statement_ ENDCASE
                {If the attribute of cases, is a number then
                return it as the attribute otherwise return the
                attribute of the OTHERS clause};
cases:
        cases case
                {if the attribute of cases is a number then return it as the
                attribute otherwise return the attribute of case}  |
        %empty
                {Set the attribute to the sentinel NAN} ;

case:
        WHEN INT_LITERAL ARROW statement_
                {$-2 contains the value of the expression after CASE.
                It must be compared with the attribute of INT_LITERAL.
                If they match the attribute of this production
                should become the attribute of statement_
                If they don't match, the attribute should be set to the
                sentinel value NAN} ;
```

`NAN` is a constant defined in `cmath` that represents not-a-number. The function `isnan` checks whether a double is `NAN`.

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

```
$ ./compile < test14.txt

   1  // Case selection
   2
   3  function main returns integer;
   4      a: integer is 4 + 2;
   5  begin
   6      case a is
   7          when 1 => 3;
   8          when 2 => (3 + 5 - 5 - 4) * 2;
   9          others => 4;
  10      endcase;
  11  end;

Compiled Successfully

Result = 4
```

10) The final two test cases, `test15.txt` and `test.16.txt`, involve nested statements and provide a further test of both the `if` and `case` statements. Shown below is the output that should result when using both test cases as input:

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

   1  -- Nested if
   2
   3  function main a: integer returns integer;
   4      b: integer is 8;
   5  begin
   6      if a >= 0 then
   7          if b > 5 then
   8              a * 2;
   9          else
  10              a + 5;
  11          endif;
  12       else
  13           a / 2;
  14       endif;
  15  end;

Compiled Successfully

Result = 2

$ ./compile < test16.txt 3 1

   1  -- Nested case
   2
```

```
function main a: integer,  b: integer returns integer;
    c: integer is 8;
begin
    case a is
         when 1 => a * c;
         when 2 => a + 5;
         when 3 =>
             case b is
                 when 1 => 2;
                 others => 19;
             endcase;
         when 4 => a / 2;
         others => a + 4;
    endcase;
end;
```

```
Compiled Successfully

Result = 2
```

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 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.